Raspberry Pi Based LCD Clock

User’s Manual

13-November-2025

Table of Contents

Table of Figures

Table of Photographs

1 Introduction

This document describes the hardware and software for a Raspberry Pi (RPi) based clock that uses six 240x320 LCD displays.

1.1 List of Hardware Components

Hardware components, with Amazon links, are listed below.

	Item	Link	Cost	Comment
1.	Raspberry Pi 4B (RPi):	RPi	~ $65	-
2.	RPi SD card (OS):	SD_Card	~ $14	Download SD OS imager here
3.	RPi Power Supply:	PWR	~ $10	-
4.	LCD Displays:	LCD	~ $15	Each, six required
5.	Mini Breadboard:	MiniBB	~$ 8	-
6.	Breadboard Jumpers:	JMP	~$13	Breadboard-to Breadboard
7.	RPi Jumpers:	JMP2	~$ 6	RPi-to-Breadboard

Total cost: ~$206.

1.2 Location of Software Components

This document and (most of) the clock's software can be found at this GitHub repository:

https://github.com/sgarrow/spiClock

Some of the clock's code is common with another, second, project. To prevent having to copy/paste changes/bug-fixes back and forth between projects, this common code resides in a third project. The common code can be found at this GitHub repository:

https://github.com/sgarrow/sharedClientServerCode

Just as a matter of completeness the above-mentioned second project that also uses the shared code is an RPi sprinkler controller. Its documentation and (most of) its software can be found at this GitHub repository:

https://github.com/sgarrow/sprinkler2

Both the clock and sprinkler run within a client/server architecture. It is the Client and Server files/functionality that are common.

1.3 List of Software Files

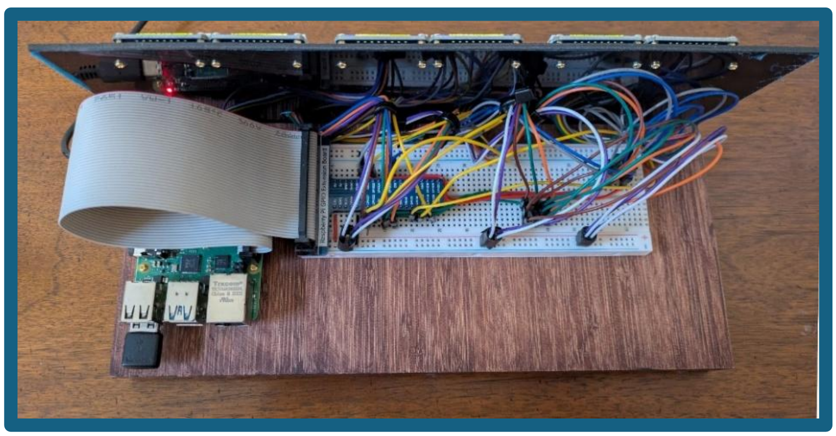
```
+--spiClock                 --+ ALL CODE RESIDES IN THIS ROOT DIRECTORY.
|    client.py                | Only these files need to be on the client machine.
|    clientCustomize          | The last 2 (cfg.*) also need to be on the server
|    gui.py                   | machine (RPi). Cfg.cfg contains IP addresses, etc.,
|    cfg.py                   | that need to be shared between the client/server
|    cfg.cfg                  | Files below only need to be on the RPi.
|                             |
|    server.py                | Clients send commands, via a socket, to the server.
|    serverCustomize.py       | The server looks up the command in a dictionary and
|    cmdVectors.py            | vectors to the worker function.  In the dictionary,
|                             | the command string serves as the "key" and the
|                             | associated "value" is the worker function.
|                             |
|    startStopClock.py        | Worker functions for the Clock Stop/Stop commands
|      clockProcess.py        | reside in startStopClock.py. These functions spawn/
|      lcdProcess.py          | terminate separate processes, running concurrently
|                             | on separate cores, that increment the clock counter
|                             | & push data out to the LCDs respectively.
|                             |
|    makeScreen.py            | Screens pushed to LCDs need to be made before the
|    styleMgmtRoutines.py     | clock starts.  Making a screen creates a file. When
|                             | that file is loaded and pushed to an LCD it results
|                             | in, e.g., a white 4 character being displayed on a
|                             | black background. Management Routines allow for
|                             | for changing styles, choosing a nighttime style,
|                             | setting the times to automatically switch styles.
|                             |
|    spiRoutines.py           | These files contain the worker functions for the
|    swUpdate.py              | remaining commands and various helper routines.
|    utils.py                 |
|    cmds.py                  |
|    testRoutines.py          |
|    fileIO.py                |
|    rpiShellCmds.py        --+
|
+----digitScreenStyles      --+ ALL SCREEN STYLES RESIDE IN THIS SUBDIRECTORY
|     blackOnWhite.pickle     | blackOnWhite is the default and can't be deleted -
|     greyOnBlack.pickle      | other styles can be. New styles can be created at
|     orangeOnTurquoise.pickle| will.  Each file contains an image for all digits.
|     turquoiseOnOrange.pickle|
|     whiteOnBlack.pickle   --+
|
+----fonts                  --+ ALL FONTS RESIDE IN THIS SUBDIRECTORY
|      Font00.ttf           --+ This file contains the font used for all digits.
|
+----pics                   --+ ALL PICTURES RESIDE IN THIS SUBDIRECTORY
|      240x320b.jpg           | Using the appropriate command the LCDs will
|      240x320c.jpg           | momentarily display a set of six 240x320 jpg
|      240x320d.jpg         --+ images.
```

Note: the **bold underlined** files are in the shared GitHub Repository, the remaining files are in the clock repository.

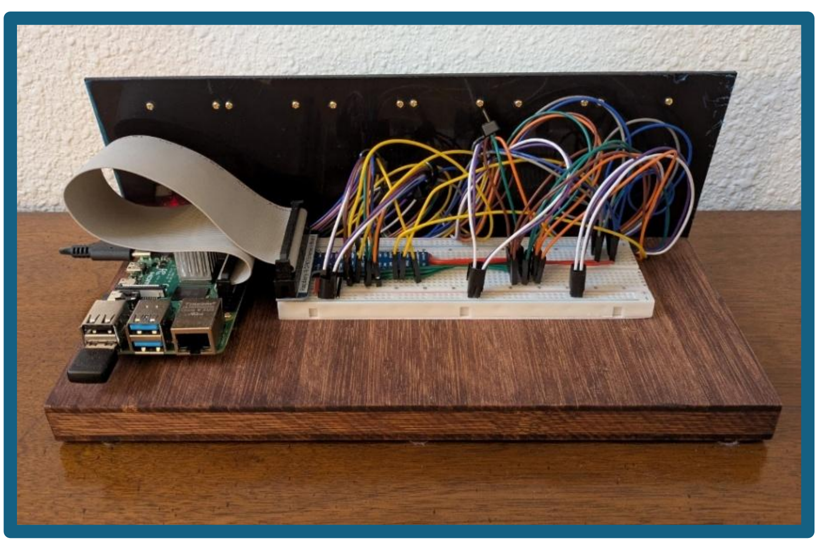
2 Photographs of Hardware Components

2.1 Photographs of Prototype Unit

Photograph 1 - Prototype Front View



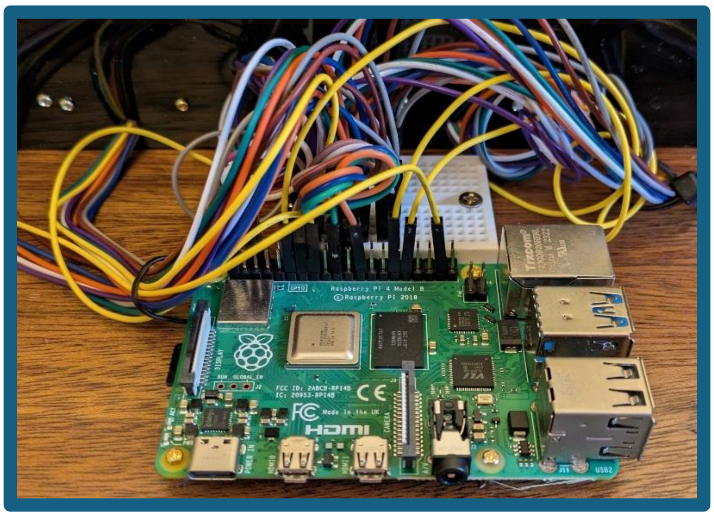
Photograph 2 - Prototype Back View 1



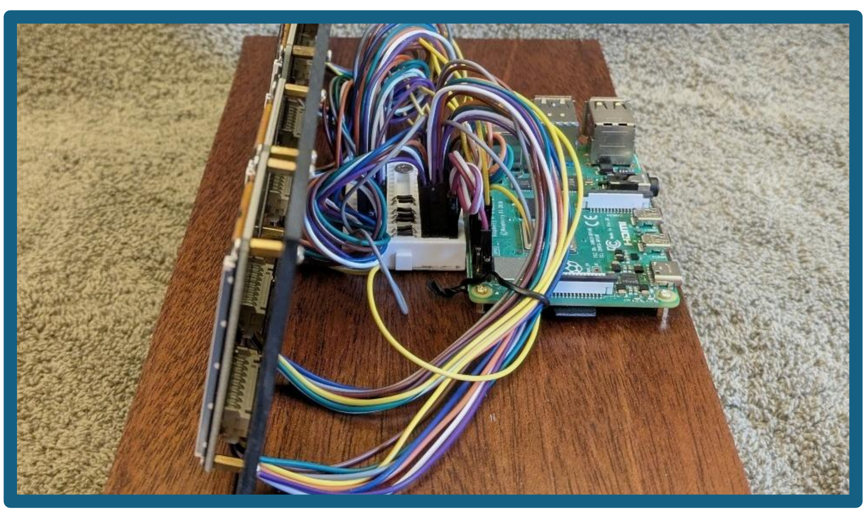
Photograph 3 - Prototype Back View 2

2.2 Photographs of Production Unit

Photograph 4 - Production Front View



Photograph 5 - Production Back View 1



Photograph 6 - Production Back View 2

3 Wiring Diagrams

The RPi talks to the LCDs over an SPI interface. An introduction to SPI can be found here: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface

Each LCD has eight connection points. Seven connection points are common to all LCDs. For example, pin 19 on the RPi (the Data In pin) is connected to all eight LCDs. So, when the RPi is pumping out data on pin 19 it is going to ALL LCDs. That said, the only LCD that is "listening" is that LCD whose Chip Select pin (CS) is "low". The CS points are NOT all common. The RPi will only drive one of the six LCD Chip Select signals low at a time.

Functional and Physical wiring diagrams are provided in Figures 1 and 2, respectively.

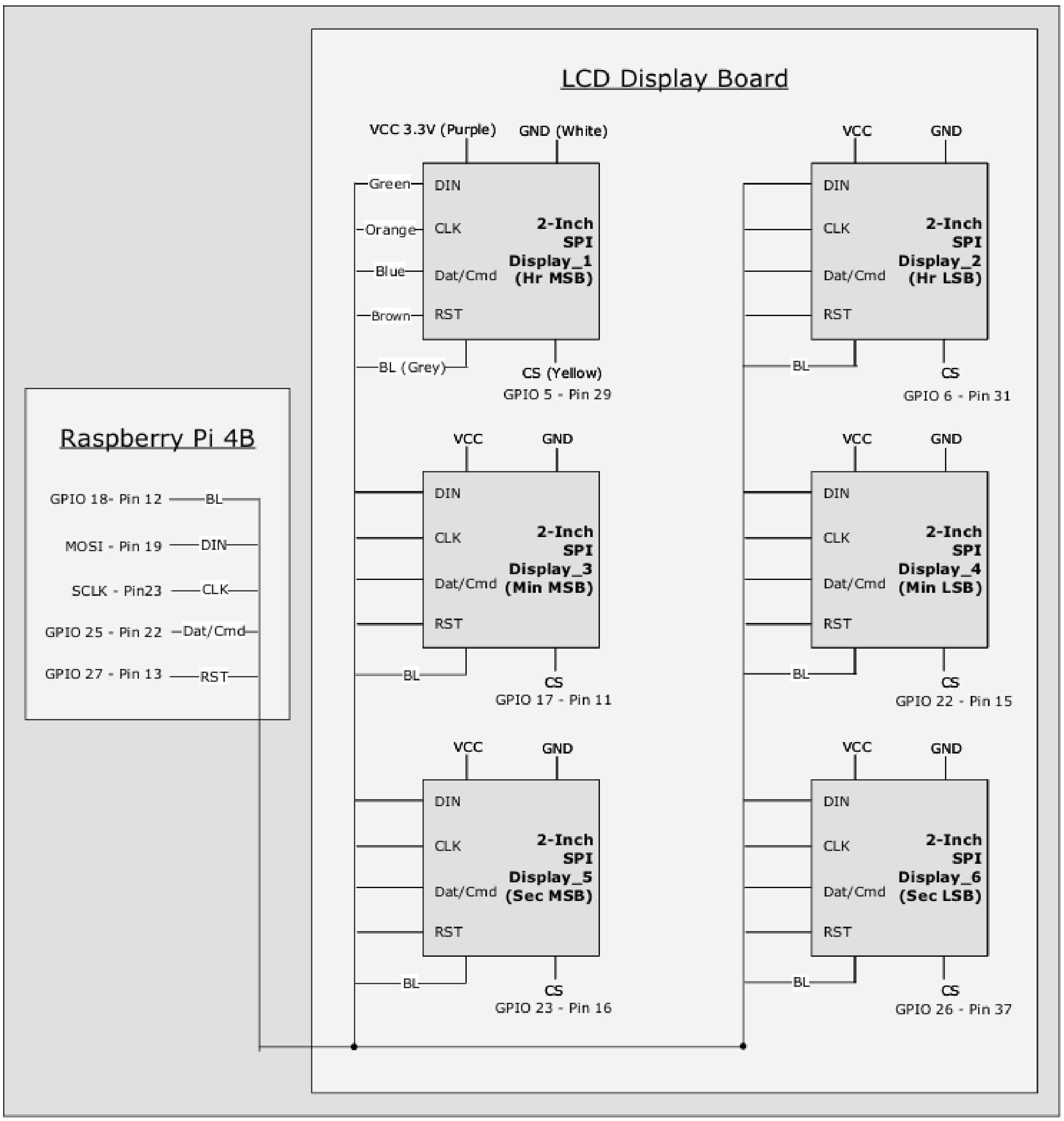


Figure 1. Functional Wiring Diagram

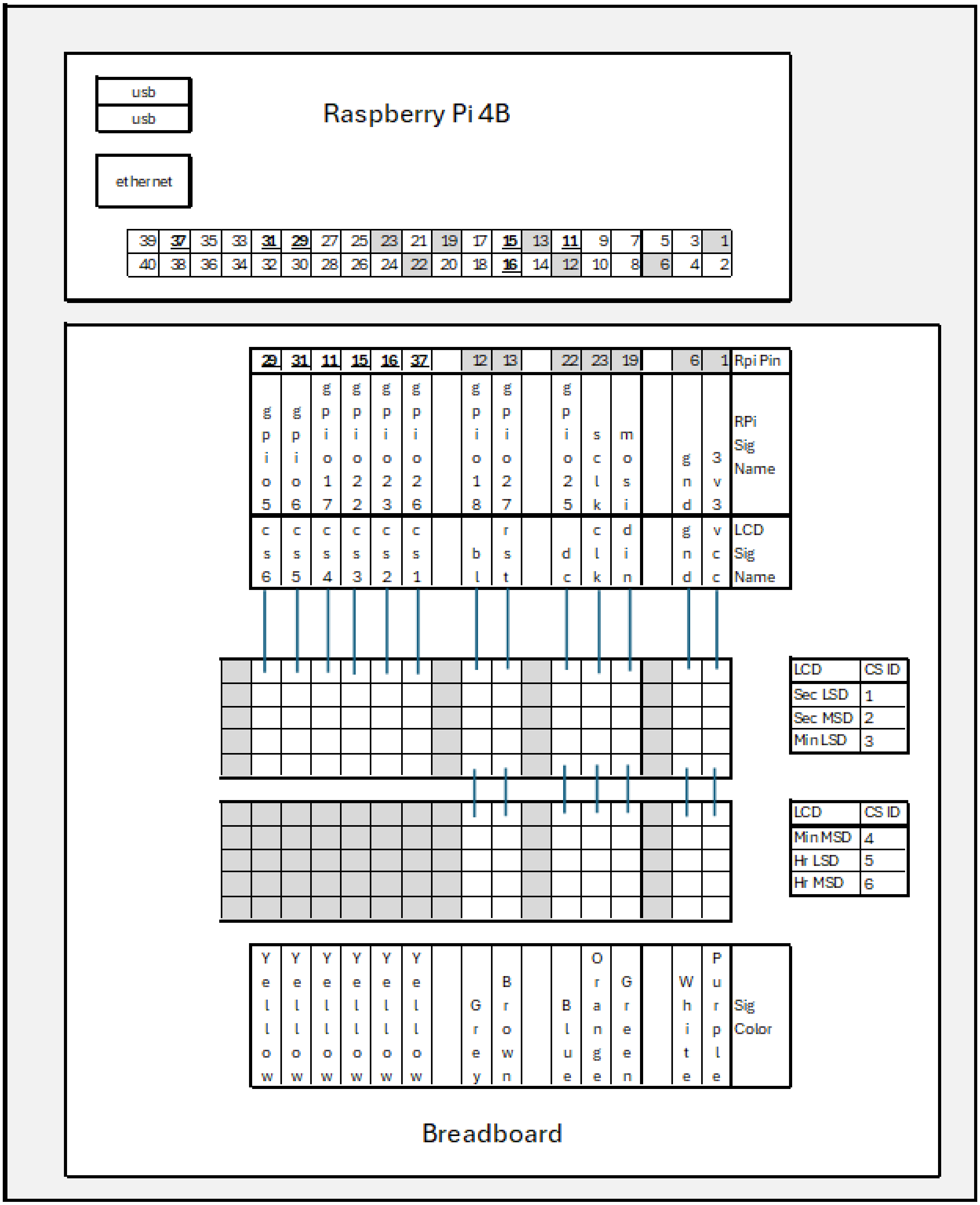


Figure 2. Physical Wiring Diagram

4 Software Operational Overview

Conceptually, there are two programs running on the RPi – a "server" and a "clock". The server can accept commands from a "client" (running on a different, remote device – a PC or a phone) over a "socket" that exists either over a Local Wireless LAN or over the internet. Control of the clock is accomplished by running the client. The client sends commands to the server who, in turn, forwards it to the clock. Information on this client/server implementation can be found in Appendix 1.

4.1 Software Processes and Communication Queues

The clock runs on three separate cores, one core runs the server (Main Process), another runs the clock counter (Clock Process) and the third controls the displays (LCD Process). These processes communicate using four multiprocessing-communication-queues. Two queues for are for sending commands and two are for receiving responses. A simplified communication diagram (for the 'stop' command) is presented in Figure 3.

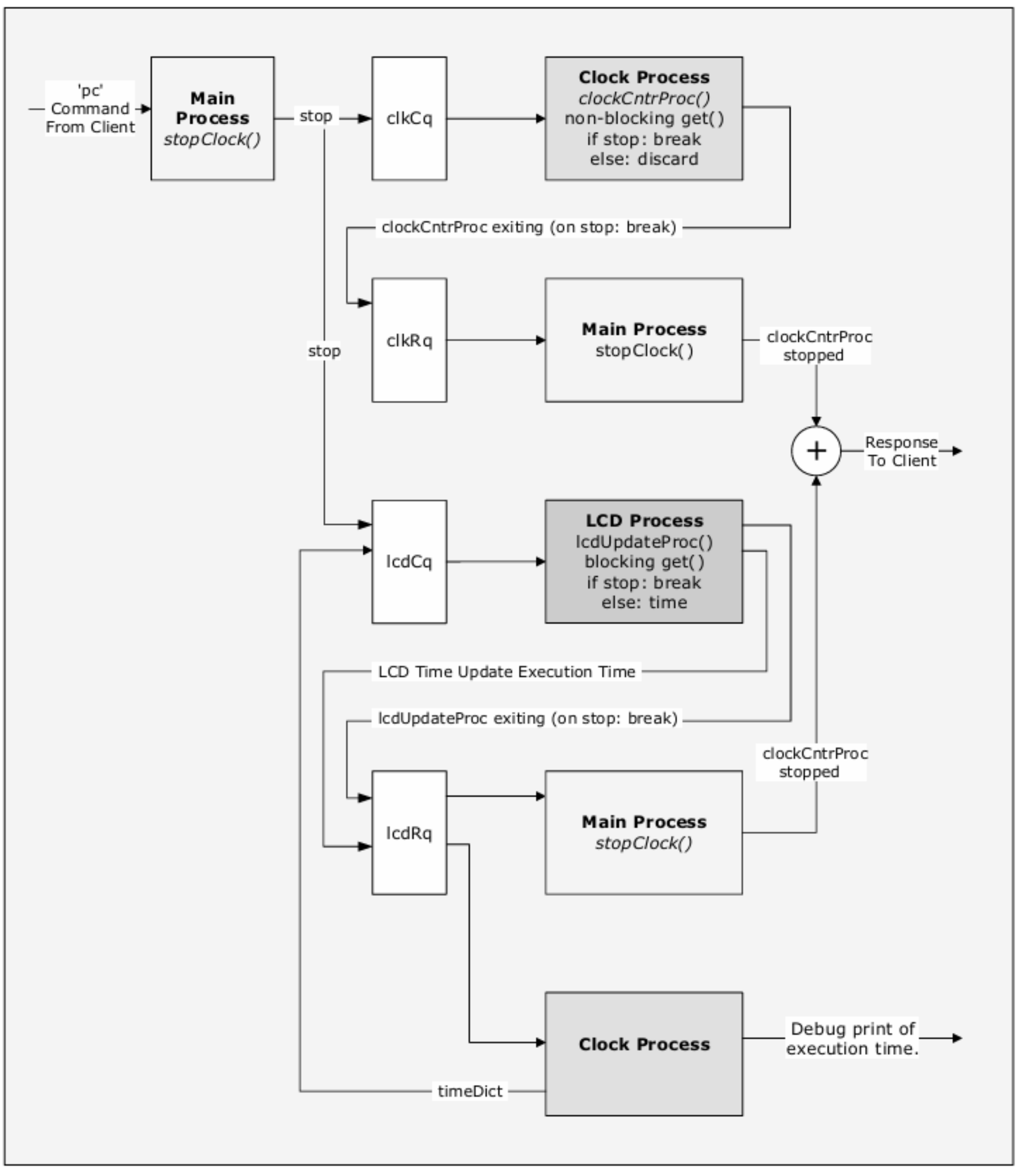


Figure 3. Communication Queues

4.2 Configuration File Setup

For the client and server to communicate they must share certain information - IP address, port numbers, etc. Users must determine this information and enter it into the file named cfg.cfg.

4.2.1 Accessing the RPi Desktop and Connecting to a Wi-Fi Network

The file cfg.cfg resides on the Solid-State Drive on the RPi. To edit the file, the RPi Desktop must first be accessed. Connect an HDMI display and a wireless (or wired) keyboard/mouse to the RPi. The connection points are designated in Figure 4. Once these connections are made, apply power to the RPi.

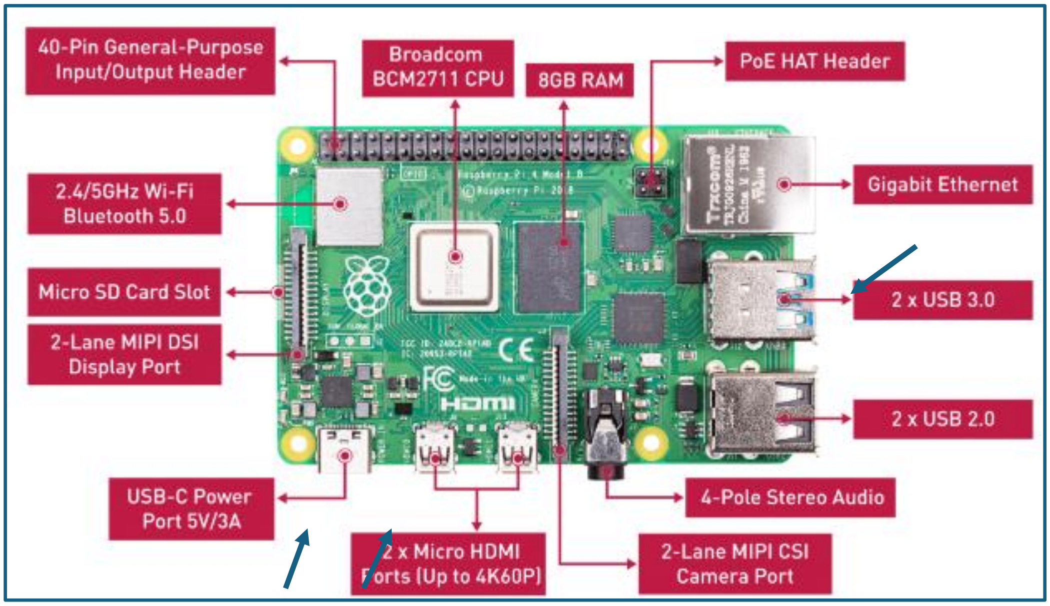


Figure 4. Raspberry Pi Connection Points

Once the display and mouse/keyboard are connected apply power via the USB-C connector. Once the RPi has booted the desktop shown in Figure 5 will be displayed.

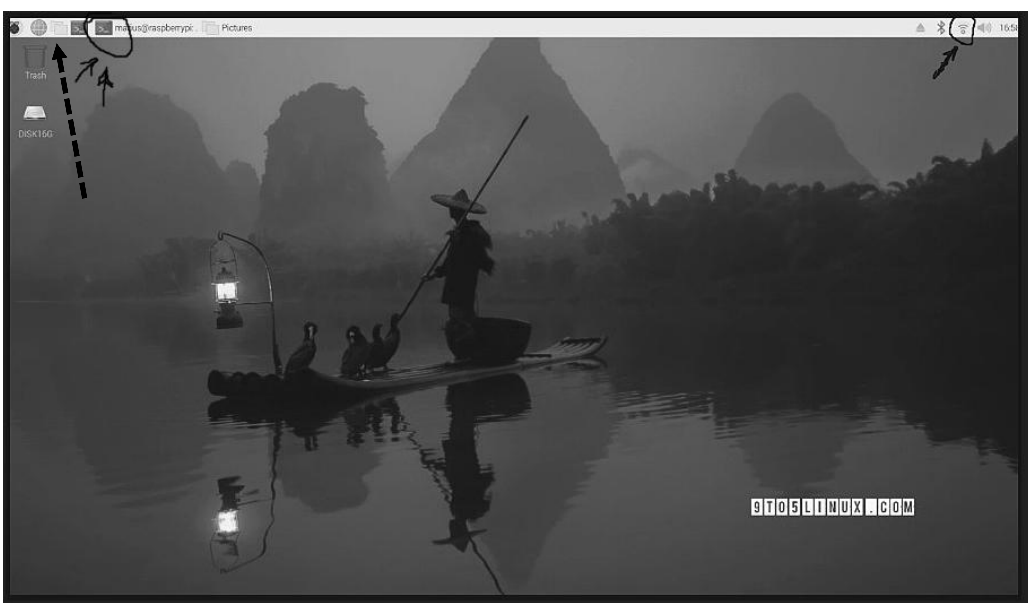
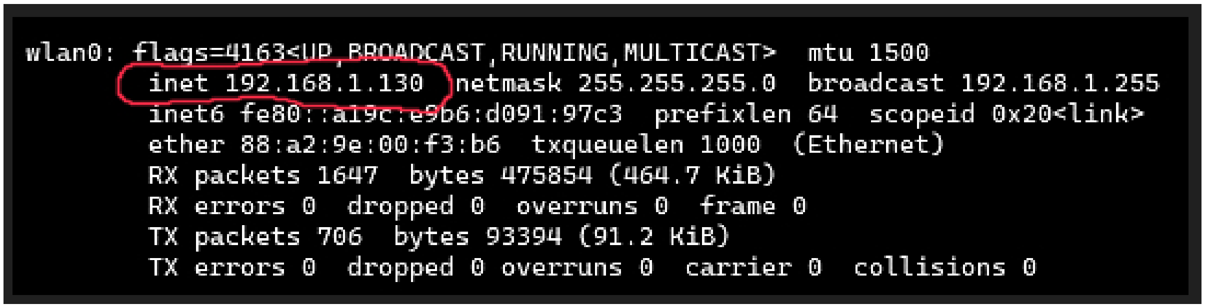


Figure 5. Raspberry Desktop

Click the Wi-Fi icon, designated above by a single arrow, select the Wi-Fi network and enter the password. Henceforth the RPi will automatically connect to this network whenever it is available.

4.2.2 Accessing the RPi Terminal Window

After connecting to the LAN, open a terminal by clicking on the icon, designated above by the double arrow, and enter the command **ifconfig**. An example output of the ifconfig command is shown in Figure 6. Take note of the inet address as it will be added to cfg.cfg.



```
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.130  netmask 255.255.255.0  broadcast 192.168.1.255
        inet6 fe80::a19c:e9b6:d091:97c3  prefixlen 64  scopeid 0x20<link>
        ether 88:a2:9e:00:f3:b6  txqueuelen 1000  (Ethernet)
        RX packets 1647  bytes 475854 (464.7 KiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 706  bytes 93394 (91.2 KiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Figure 6. ifconfig Command Output Screenshot

4.2.3 Editing the Configuration File with the Built-In Nano Editor.

In a terminal window, enter the following command to navigate to the appropriate RPi directory:
cd python/spiClock

The following command opens cfg.cfg in the nano editor (mouse doesn't work use arrow keys):
nano -e cgf.cfg Note: An example cfg.cfg is provided in Figure 7.

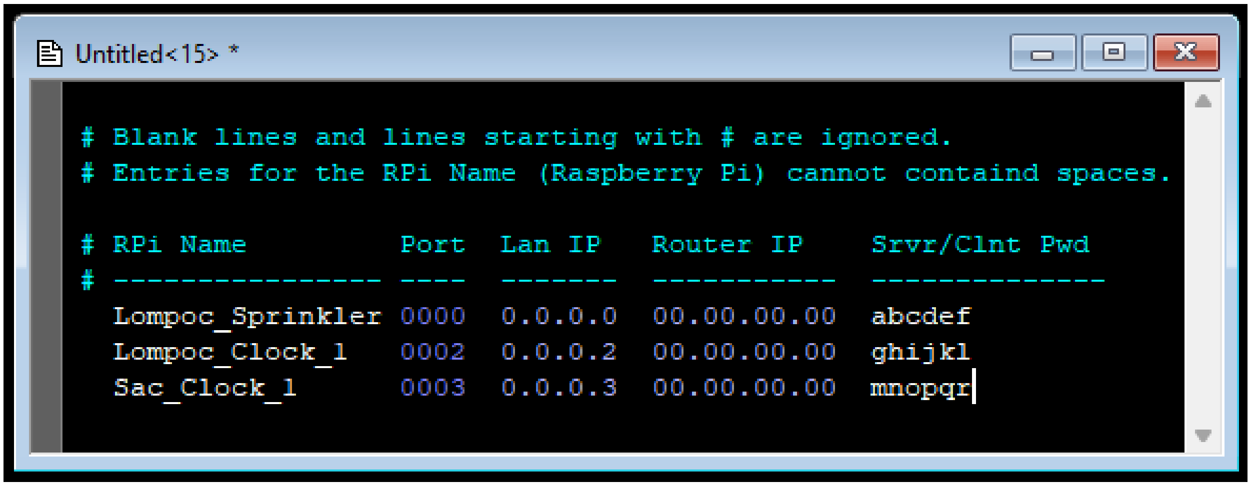
Edit an existing line or add a new line. The RPi name will be entered when starting the server (on the RPi) and when starting the client (on a remote machine, either a PC or a phone).

Enter a port number. Ports between 8000 and 9000 are safe to use. Enter the Lan address obtained by the ifconfig command. Enter your router's IP address. The router IP address can be obtained via website www.whatismyip.com. Choose an enter your desired password.

Note that the password contained in the cfg.cfg file is the password that the client will send to the server when attempting to establish a connection. It is not the "user" password for the RPi itself. The RPi password for the RPi itself is used when, for example attempting, to establish an SSH connection to the RPi.

Save the file by entering the following commands:

ctrl-x	# To begin the editor exiting process.
Y	# Answer to the "save-file" prompt.
<RETURN>	# Answer to the "save as cfg.cfg" (default) prompt.



Untitled<15> *

```
# Blank lines and lines starting with # are ignored.
# Entries for the RPi Name (Raspberry Pi) cannot containd spaces.

# RPi Name          Port  Lan IP   Router IP    Srvr/Clnt Pwd
# ----------------- ----  -------  -----------  --------------
  Lompoc_Sprinkler 0000  0.0.0.0  00.00.00.00  abcdef
  Lompoc_Clock_1   0002  0.0.0.2  00.00.00.00  ghijkl
  Sac_Clock_1      0003  0.0.0.3  00.00.00.00  mnopqr
```

Figure 7. Example cfg.cfg File Screenshot

4.2.4 Copying Files from the RPi to a Flash Drive.

As mentioned in section 1.3, there are six files that need to be placed on the machine that will run the client. Insert a Flash Drive into a USB Port on the RPi and, using the RPi File Explorer (click on the File Explorer Icon designated be the dotted arrow in Figure 5. Raspberry Desktop), copy the following files to it:

`client.py, clientCustomize.py, gui.py, cfg.py, cfg.cfg`

4.3 Starting the Server Manually

Start the server on the RPi by entering the following command in a terminal window:

```
Python3    server.py   <RPi Name>
```

Replace <RPi Name> with the appropriate name from the cfg.cfg file.

4.4 Starting the Server Automatically at Boot Time

Cron is a shell command for scheduling a job (i.e. command or shell script) to run periodically at a fixed time, date, interval, or at boot-time.

Open the cron scheduling file for editing on the RPi enter the following command in a terminal window:

```
Crontab  -e
```

Add the line shown at the bottom of the screenshot provided in Figure 8.

Save the file by entering the following commands:

`ctrl-x`	# To begin the editor exiting process.
`Y`	# Answer to the "save-file" prompt.
`<RETURN>`	# Answer to the "save as" prompt.

Cycle power to the RPi and the clock will start after about 30 seconds. The 30 second sleep is required to ensure that the RPi has completely booted (including connecting to the Wi-Fi) before the server script is started.

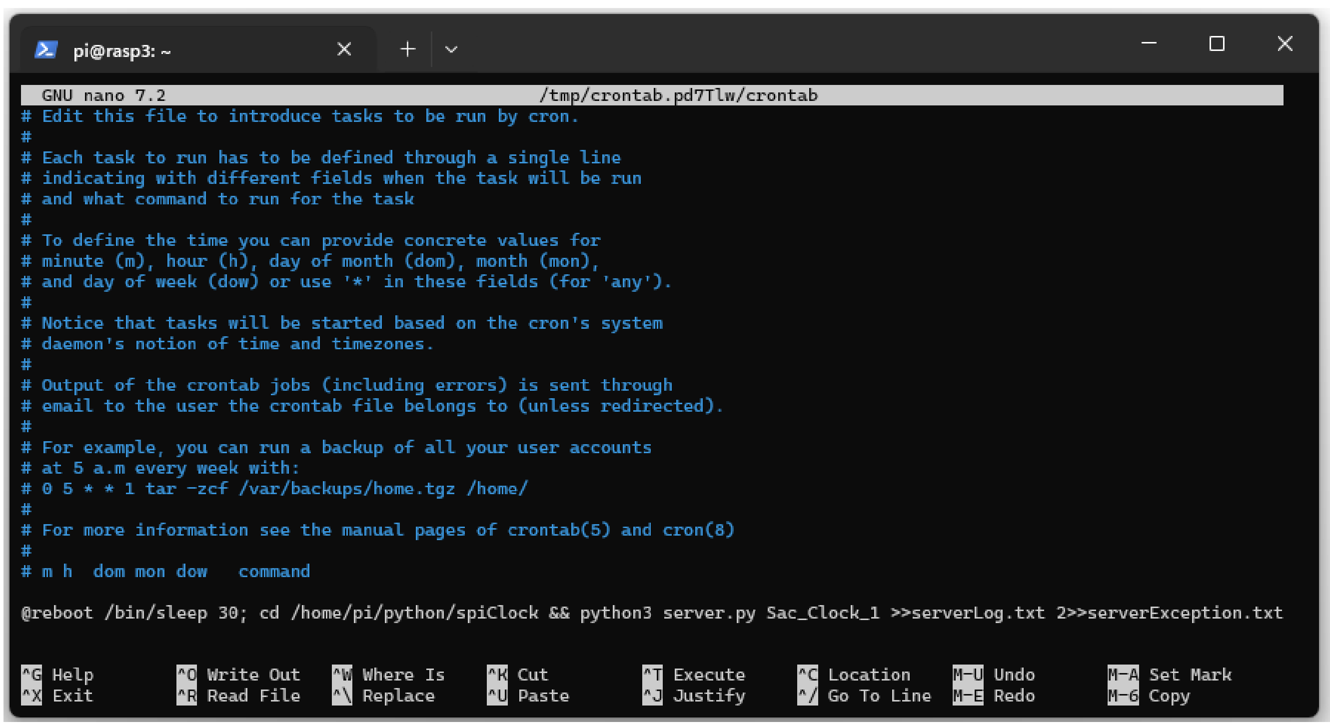


Figure 8. Crontab Screenshot

4.5 Starting the Command Line Client

Start the command line client on a remote machine by entering the following command:

`python  client.py  <id>` Replace **`<id>`** as appropriate, see Section "Configuration File", below

Once the server accepts the connection a prompt is presented. Enter 'm' at the prompt to get a list of available commands. A screen shot of this is provided below.

```
PS C:\01-home\14-python\gitTrackedCode\spiClock> python .\client.py clk2
 same, lan, internet (s,l,i) -> l
 sndBufSize 65536
 rcvBufSize 65536

 Accepted connection from: ('192.168.1.110', 65386)


 Choice (m=menu, close) -> m

 === GET   COMMANDS ===
 gas - Get Active Style
 gds - Get Day Style
 gns - Get Night Style
 gAs - Get ALL Styles
 gdt - Get Day Time
 gnt - Get Night Time
 gat - Get Active Threads
 gvn - Get Version Number

 === SET   COMMANDS ===
 sas - Set Active Style
 sds - Set Day Style
 sns - Set Night Style
 sdt - Set Day Time
 snt - Set Night Time

 === FILE  COMMANDS ===
 ral - Read App Log File
 rsl - Read Srvr Log File
 rse - Read Srvr Exc File
 cal - Clr App Log File
 csl - Clr Srvr Log File
 cse - Clr Srvr Except File

 === OTHER COMMANDS ===
 sc  - Start Clock
 pc  - Stop Clock
 mus - Make User Style
 dus - Delete User Style
 dp  - Display Pics
 up  - Upload Pic
 rp  - Remove Pic
 us  - Update SW
 hlp - Help
 close - Disconnect
 ks  - Kill Server
 rb  - Reboor RPi

 === TEST  COMMANDS ===
 rt1 - Run Test 1
 rt2 - Run Test 2
 rh  - Reset LCD HW Test
 rs  - Reset LCD SW Test
 sb  - LCD Backlight Test
 lc  - List Commands Test


 Choice (m=menu, close) ->
```

Figure 9. Command Line Client Screenshot

4.6 Starting the GUI Client

Start the GUI client with the following command: `python gui.py` .Then, when the following screen is displayed click on the appropriate UUT (this is the equivalent of the <id> parameter specified when starting the Command Line Client).

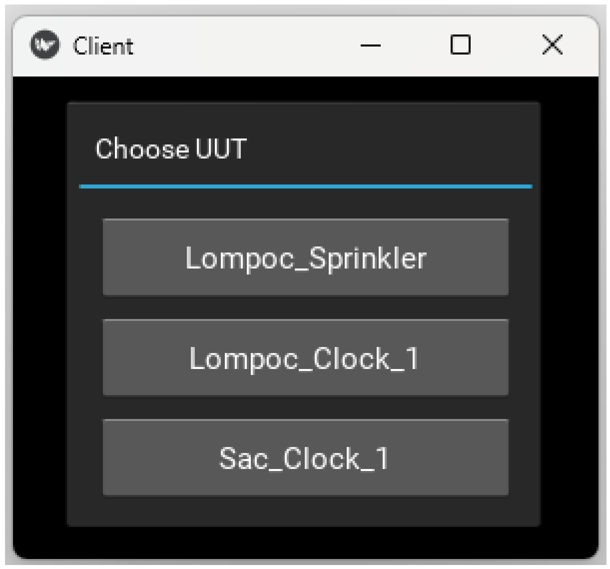


Figure 10. GUI Client Screenshot 1

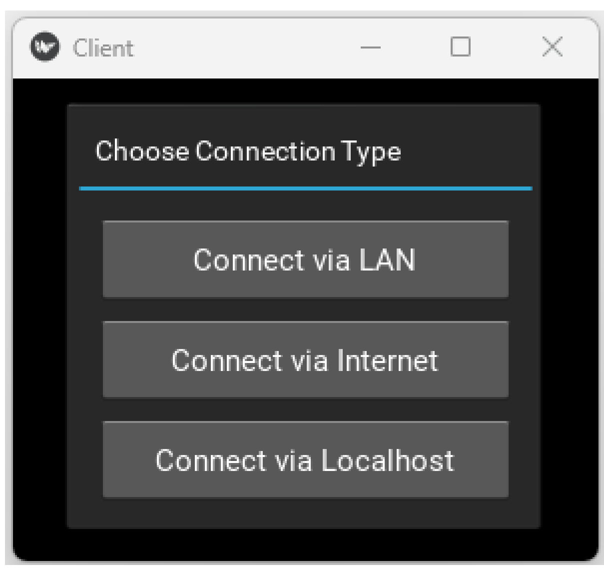


Figure 11. GUI Client Screenshot 2

When the GUI client is started and a connection is accepted by the server the 'm' command is automatically issued. The response is used to populate the GUIs buttons (GUI is 'built' at run time). A screen shot of the GUI is provided below.

The 'build at run time' architecture is what allows the sprinkler and clock applications to use the same GUI.

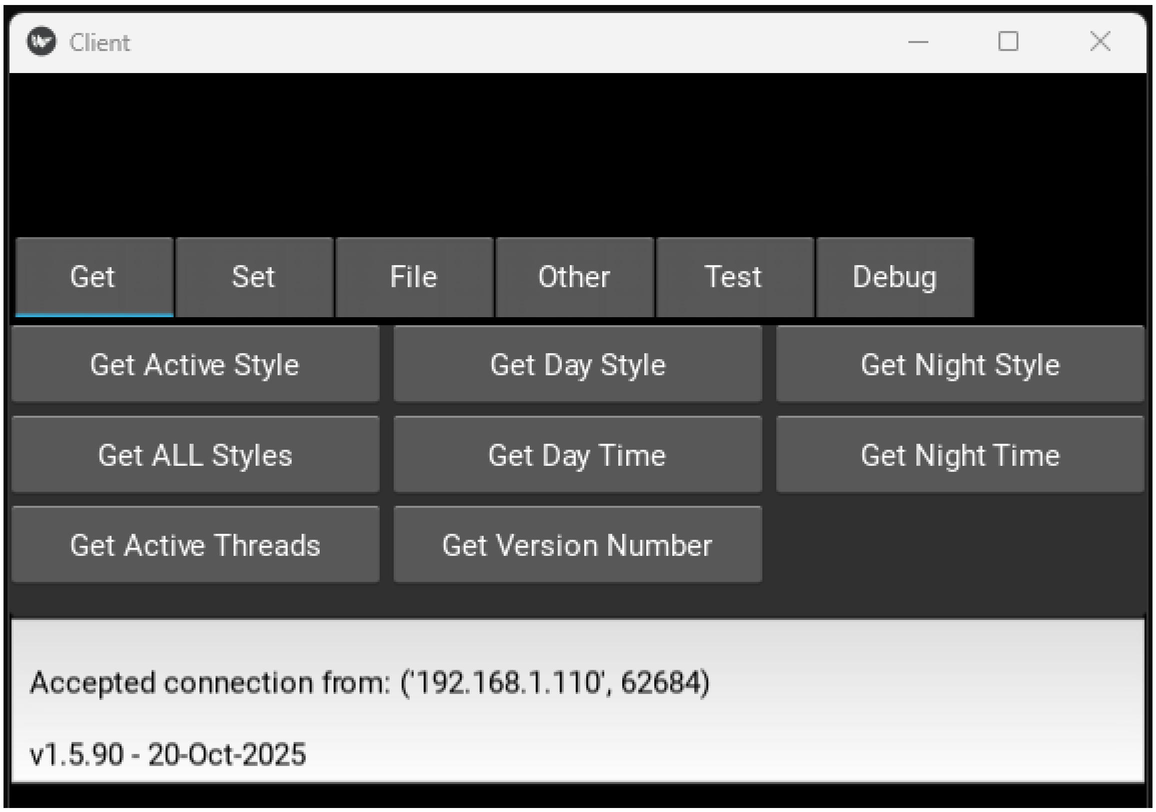


Figure 12. GUI Client Screenshot 3

4.7 Memory Utilization

The Amazon Link provided earlier for the RPi earlier is a link to a 4GB (RAM) version of the RPi. However, the application can run on a 1GB version. Below is a screen capture of the output of four issuances of the `free --mega` Linux command. The first command was issued right after boot where only the OS is running, the second after the server was started, the third after a client was connected and the fourth after the clock was started.

The data shows that after everything is up and running there is still over 76MB of free RAM. Plenty for dozens of more clients to connect.

```
pi@rasp3:~ $ free --mega    # After Boot

          total        used        free      shared  buff/cache   available
Mem:        950         359         152          24         520         591
Swap:       536           0         536

pi@rasp3:~ $ free --mega    # After Starting Server

          total        used        free      shared  buff/cache   available
Mem:        950         389         100          24         542         560
Swap:       536           0         536

pi@rasp3:~ $ free --mega    # After Connecting Client

          total        used        free      shared  buff/cache   available
Mem:        950         387         102          24         542         562
Swap:       536           0         536

pi@rasp3:~ $ free --mega    # After Starting Clock

          total        used        free      shared  buff/cache   available
Mem:        950         410          76          24         545         539
Swap:       536           0         536

pi@rasp3:~ $
```

5 Appendix 1. A Python Based Client-Server Architecture

5.1 Introduction

This document describes a client-server architecture written in the Python programming language. A client-server architecture is a structure where multiple clients request services from a centralized server and separates user interaction from data processing.

To start the server type "python server.py <device>" on the command line.

To connect a command line client to the server type "python client.py <device>" on the command line.

To connect a GUI client to the server type "python gui.py <device>" on the command line.

A client can be run on the same machine as the server (in a different command window) or on a different machine entirely.

5.2 A Well-Known Client Server

Servers are things that respond to requests. Clients are things that make requests.

A web browser is a type of client that can connect to servers that "serve" web pages - like the Google server.

When a web browser (a client) connects to the Google Server and sends it a request (e.g., send me a web page containing a bunch of links related to "I'm searching this") the Google Server will respond to the request by sending back a web page.

Requests are sent in "packets" over connections called "sockets". Included in the request is the IP address of the client making it - that's how the server knows where to send the response back to. A given machine has one IP address, so if more than one instance of a web browser is open on a single machine how is it that the response ends up in the "right" web browser and not the other browser? Port number.

5.3 Closing a Client and Stopping the Server

Every client has a unique (IP, port) tuple. The server tracks every client by (IP, port). The server maintains a list of (IP, port) for all active clients.

Each client has two unique things associated with it - (1) a socket and (2) an instance (a thread) running the client's handling function

When a client issues a "close" command, its (IP, port) is removed from the list and as a result the handleClient's infinite loop is exited thereby causing its socket to be closed and its thread to terminate.

When a client issues a "ks" (kill server) command, not only does that client terminate but all other clients terminate as well. Furthermore the "ks" command causes the server itself (it's still waiting for other clients to possibly connect) to terminate.

Upon receipt of the ks command the server (1) sends a message to all clients (including the one sent the command) indicating that the server is shutting down so that the client will exit gracefully, (2) terminates all clients and then finally (3) the server itself exits.

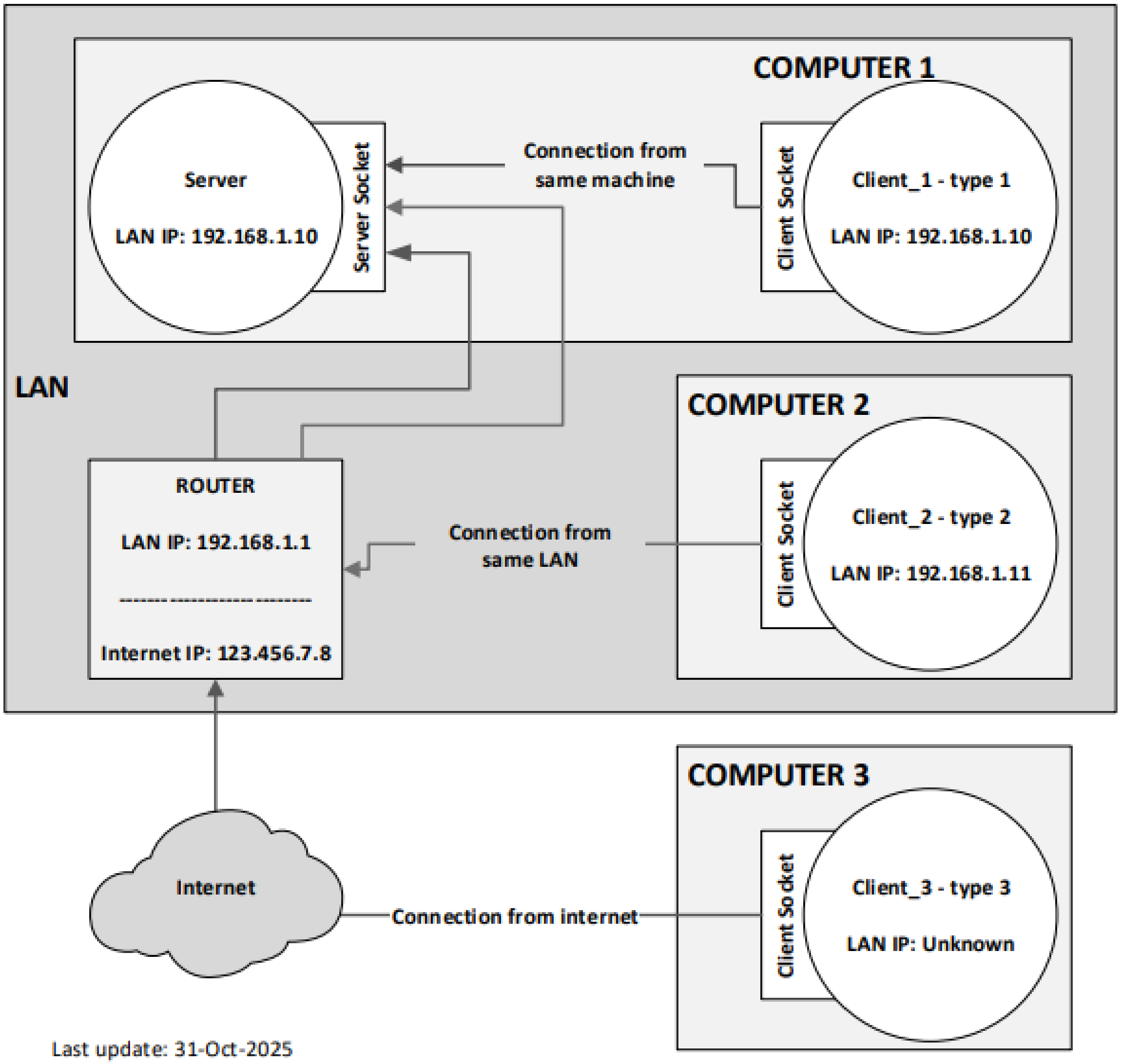
Additional details related to client connection types and to function calling sequences are provided in figures 1 and 2.

5.4 Server's Handling of Unexpected Events

If a user clicks the red X in the client window (closes the window) that client unexpectedly (from the server's viewpoint) terminates. This contrasts with the client issuing the close or ks command where the server is explicitly notified of the client's termination. An unexpected termination results in a sort of unattached thread and socket that may continue to exist even when the server exits. This situation is rectified by two try/except blocks in function handleClient. Two are needed because it was empirically determined the Window and Linux systems seem to block (waiting for a command from the associated client) in different places.

5.5 Connection Types

Clients can connect to the sever under three different scenarios as described in Figure A-1.



Computers 1 & 2 & the router are all on the same Local Area Network (LAN). Computer 3 in not on the same LAN but is connected to the Internet (an Interconnected set of Local Area networks).

When a client is started it prompts for the desired connection type and offers 3 choices: 's', 'l', 'i'.
s = same machine = type 1. l = same lan = type 2. i = internet = type 3.

Computer 1 is running both the server and the client so the obvious connection type is type 1, as shown. However, computer 1 could also connect over the LAN or over the internet (neither shown).

Computer 2 is running only the client but is on the same LAN as the computer running the server so the obvious connection type is type 2, as shown. However, computer 2 could also connect over internet (not shown). Computer 2 can not connect to the server via a type 1 connection.

Computer 3 can only connect to the server via a type 3 connection.

The server can handle multiple connections of type 1,2 and 3 simultaneously.

Figure A1. Connection Types

For connection type 2, in addition to the port number, the IP of the server needs to be known. The address can be found via the ifconfig command in a command window open on the machine that will be running the server.

For connection type 3, in addition to the port number, the external IP of the router needs to be known. The router's external IP address can be found using by going to the following web page on a browser: **https://whatismyipaddress.com/**

The use of connection type 3 also requires port forwarding to be set up on the router. An example is shown below. The example shows forwarding port 1234 (substitute 1234 with whatever port is configured) to port 1234 for IP address 192.168.1.10. Substitute 192.168.1.10 with whatever the IP address of the machine running the server is. Again, this address can be obtained via use of the ipconfig command. Since only one port number needs to be forwarded the start and end port numbers are the same.

It seems weird that a port number needs to be forwarded to that same number, but it does.

Service Name	External Start Port	External End Port	Internal Start Port	Internal End Port	Internal IP address
My Service Name	1234	1234	1234	1234	192.168.1.10

5.6 A Potential Pitfall – Firewall

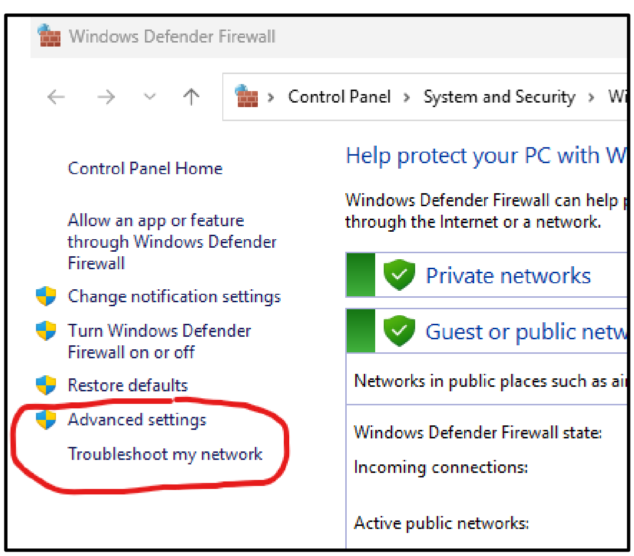
The above was all initially done with the server running on a Raspberry Pi. The Raspberry Pi runs Linux and by default its firewall is disabled. As such, all connection types worked only by performing the "SOME ASSEMBLY REQUIRED" steps outlined above.

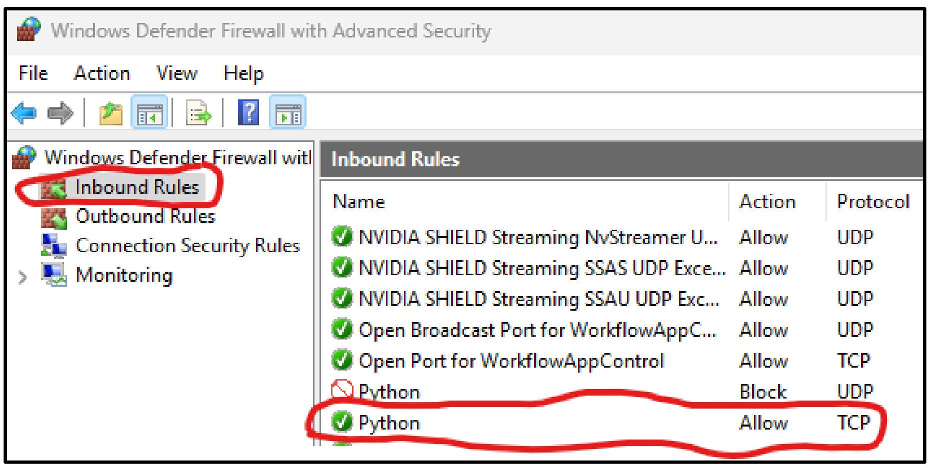
On windows to get all connection types working, specifically connection type 3, the Windows firewall will need to be changed to allow incoming python TCP connections.

These are the basic steps:

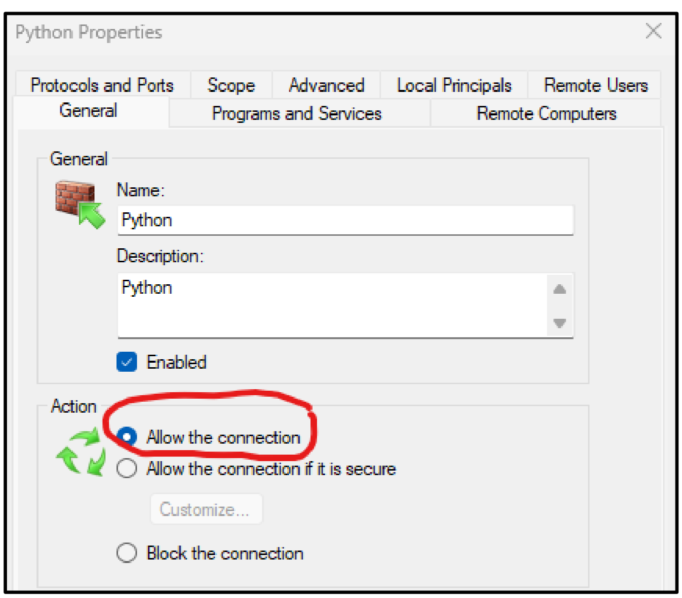
Control Panel\System and Security\Windows Defender Firewall ---> Advanced settings ---> Connection Security Rules ---> Inbound Rules

Screen shots for these various are provided below.

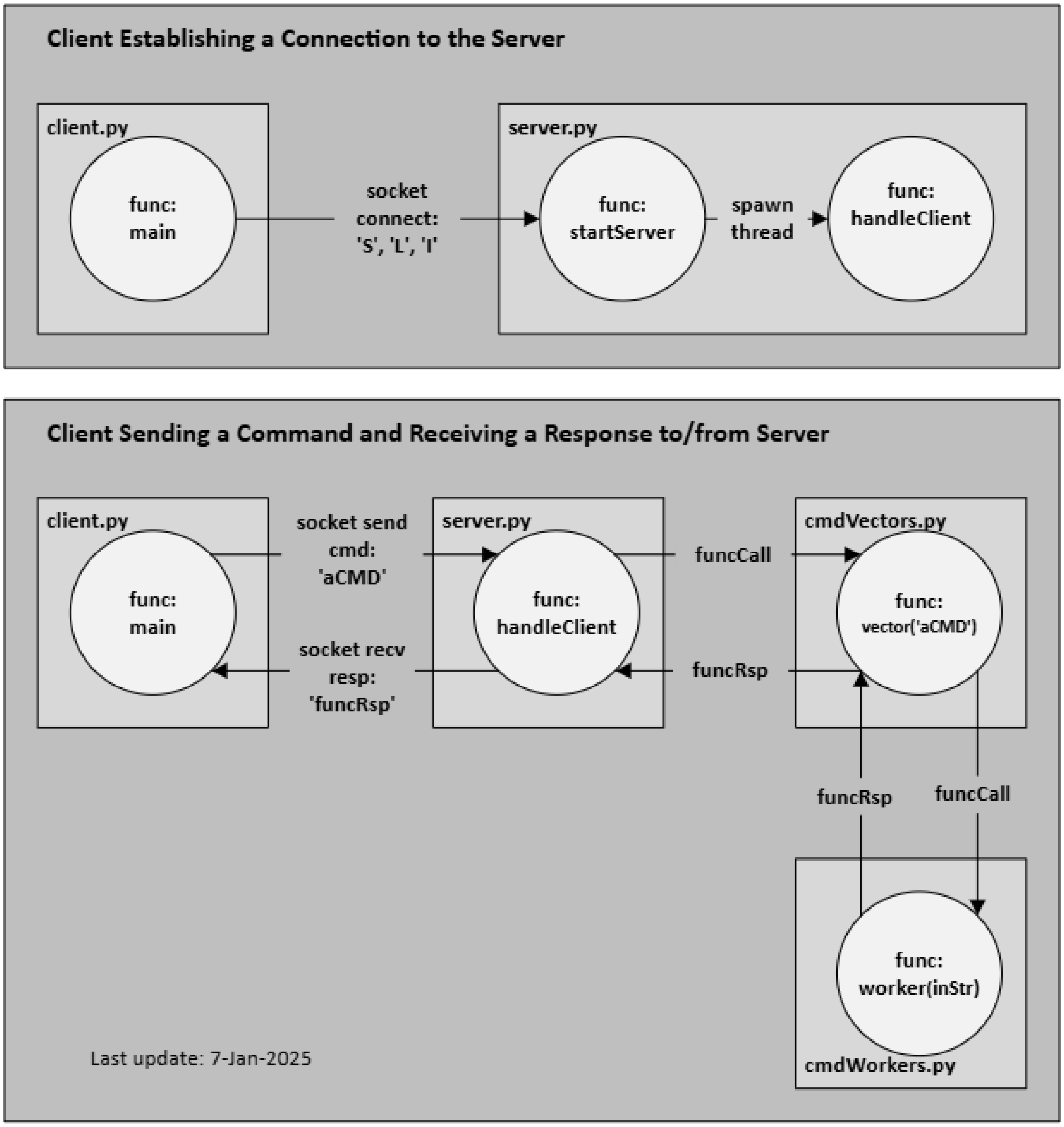




Right click on the Incoming Rule for Python TCP Protocol, select the General Tab and change to Allow the connection.



5.7 Functional Call Tree



A client's connection request gets transmitted to the server over a socket where it is recieved by function startServer. When the server accepts the connection it spawns a thread that runs function handleClient that is then dedicated to servicing commands recieved from that client.

A client's command (a text string) is recieved by function handleClient who in turn forwards it to function vector. Function vector looks up the worker function associated with the command and subsequently calls it. The worker function performs the associated processing and return the response (also a text string). The response is passes back up the call tree where it is eventually recieved by the client.

Figure A-2. Functional Call Tree

6 Appendix 2. SSH, SCP and other Handy RPi Commands

Installing Python3

1 sudo apt update
2 sudo apt install python3 idle3

Installing Various Python Packages

yaml: sudo apt install python3-yaml
upgrade pip: python3 -m pip install --upgrade pip

To enable SSH connections on Rpi:

1 settings -> pref -> config -> interfaces -> en SSH
2 sudo reboot
3 ifconfig on terminal
4 get ipaddr, wlan0 section
5 whoami "user"
6 get password

Establishing/Closing SSH connections to Rpi:

Over direct ethernet cable: ssh pi@raspberrypi.local
Over LAN IP: ssh pi@12.34.56.78
Closing connection: ~.

Using SCP to copy files to RPi:

Copy a file from PC to specified RPi directory:

scp .\sprinkler.py pi@12.34.56.78: ~/python/sprinkler

Copy all .py files PC directory to specified RPi directory:

scp *.py pi@12.34.56.78: ~/python/spiClock

To set up an SSH pass key (eliminates having to supply password for SSH and SCP commands):

First, on the PC, create a key and copy it to rpi:

ssh-keygen
scp -v "C:\Users\stang\.ssh*.pub" pi@192.168.1.120:~/.

Second, on the RPi put the key in the right place:

mkdir -p ~/.ssh
chmod 700 ~/.ssh
mv ~/id_ed25519.pub ~/.ssh/
cat ~/.ssh/id_ed25519.pub >> ~/.ssh/authorized_keys
chmod 600 ~/.ssh/authorized_keys
rm ~/.ssh/id_ed25519.pub # Clean up (optional)

Then restart SSH on the RPi:

sudo systemctl restart ssh

Now when SSH-ing you will be prompted for the passphrase you used when you created the key so to not have to enter that every time (essentially, we're no better off than before):

Solution: Run PowerShell as Administrator

1. Click on the **Start Menu** and search for **PowerShell**.
2. Right-click on **Windows PowerShell** and select **Run as administrator**.
3. Now, try running:

   Set-Service ssh-agent -StartupType Automatic
   Start-Service ssh-agent
4. If no errors appear, run: ssh-add ~/.ssh/id_ed25519
5. Verify the key is loaded using: ssh-add -l

<u>To enable NTP synchronization:</u>

timedatectl set-ntp true

==== AUTHENTICATING FOR org.freedesktop.timedate1.set-ntp ====
Authentication is required to control whether network time synchronization shall be enabled.
Authenticating as: ,,, (pi)
Password:
==== AUTHENTICATION COMPLETE ====

<u>To see what scripts are running:</u>

ps -aef | grep python (aux?)

<u>To see what ports are open:</u>

ss -lntp
netstat |grep -E 'tcp|State'