Raspberry Pi Based LCD Clock

Contents

Introduction

This document describes the hardware and software for a Raspberry Pi (RPi) based clock that uses six 320x240 LCD displays.

List of Hardware Components

The major hardware components, with Amazon links, are listed below.

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.

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 has been factored out and placed into a third project. The common code, third project, 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.

List of Software Files

```
+--spiClock                     --+ ALL CODE RESIDES IN THIS ROOT DIRECTORY.
|    client.py                    | These 4 files are the only ones that need to be on
|    clientCustomize              | the machine running the client, command line of gui.
|    gui.py                       | The last 2 (cfg.*) also need to be on the machine
|    cfg.py                       | running the server (RPi). cfg.cfg, read by cfg.py,
|    cfg.cfg                      | contains IP addresses and passwords that need to be
|                                 | shared between the client and the server. Files below
|                                 | here only need to be on the RPi.
|                                 |
|    server.py                    | Clients send commands via a socket to the server. The
|    serverCustomize.py           | The command is looked up in a table where the worker
|    cmdVectors.py                | function for that command is found and vectored to.
|                                 |
|    startStopClock.py            | Worker functions for the Start Clock & Stop Clock
|      clockProcess.py            | commands reside in startStopClock.py. These worker
|      lcdProcess.py              | functions spawn/terminate two separate processes,
|                                 | running concurrently on separate cores, that
|                                 | increment the clock counter and push data out to
|                                 | the LCDs respectively.
|                                 |
|    makeScreen.py                | Screens pushed to the LCD need to be made before the
|    styleMgmtRoutines.py         | clock starts.  Making a screen results in a file.
|                                 | When that file is loaded and pushed to an LCD it
|                                 | results in, for example, a white "4" character being
|                                 | displayed on a black background.  Management Routines
|                                 | allow for changing styles, choosing a nighttime style,
|                                 | setting the times to automatically switch styles, etc.
|                                 |
|    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 style and can't be deleted.
|      greyOnBlack.pickle         | All other styles can be deleted. New styles can be
|      orangeOnTurquoise.pickle   | created at will.  Each file contains an image for
|      turquoiseOnOrange.pickle   | all digits 1 through 9.
|      whiteOnBlack.pickle      --+
|
+----fonts                      --+ ALL FONTS RESIDE IN THIS SUBDIRECTORY
|      Font00.ttf               --+ This file contains the font that used for all digits.
|
+----pics                       --+ ALL PICTURES RESIDE IN THIS SUBDIRECTORY
|      240x320b.jpg               | Using the appropriate command the LCDs can be forced
|      240x320c.jpg               | to momentarily display a set of six 240x320 jpg
|      240x320d.jpg               | images.
|      240x320e.jpg               |
+      240x320f.jpg             --+
```

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

Memory Utilization

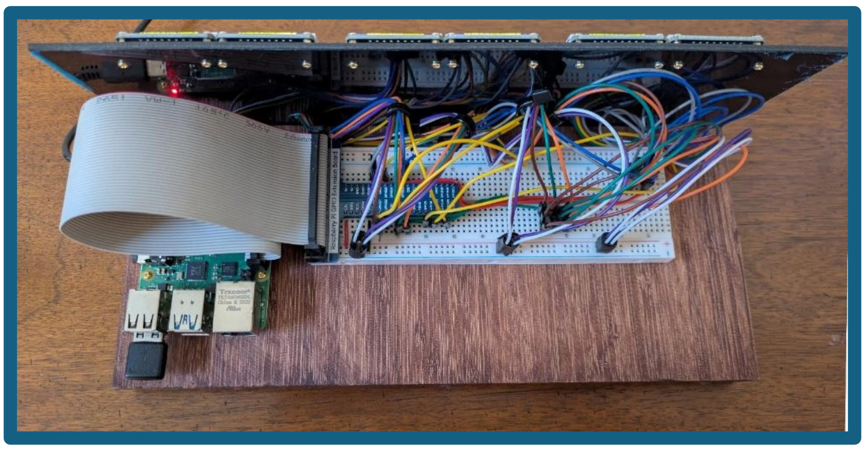
The Amazon Link provided for the RPi earlier in this document is a link to a 4GB (RAM) version of the RPi. However the application can run on a 1GB version. Below is a screen capture of the output of four issuances of the `free --mega` Linux command. The first command was issued right after boot where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

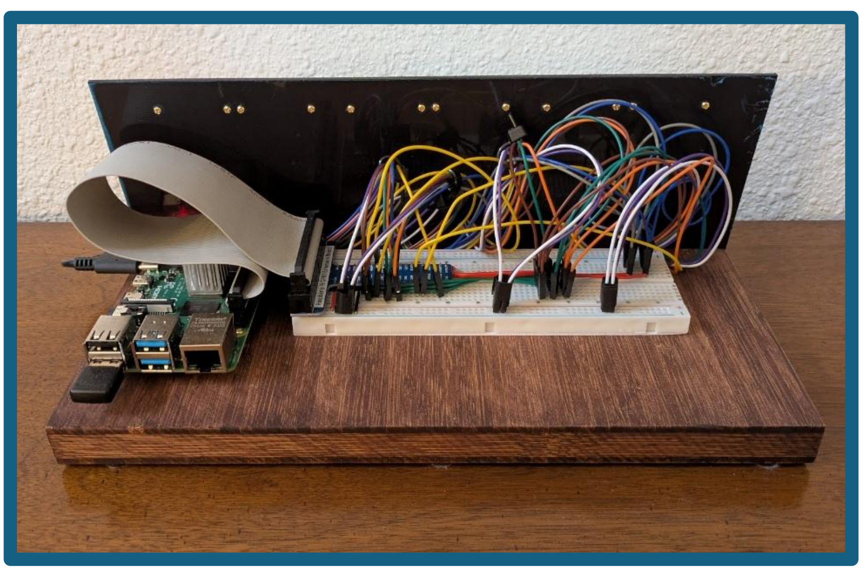
```
pi@rasp3:~ $ free --mega   # After Boot

        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:~ $
```

Photographs of Hardware Components

Photograph 1 – Front View



Photograph 2 – Back View 1



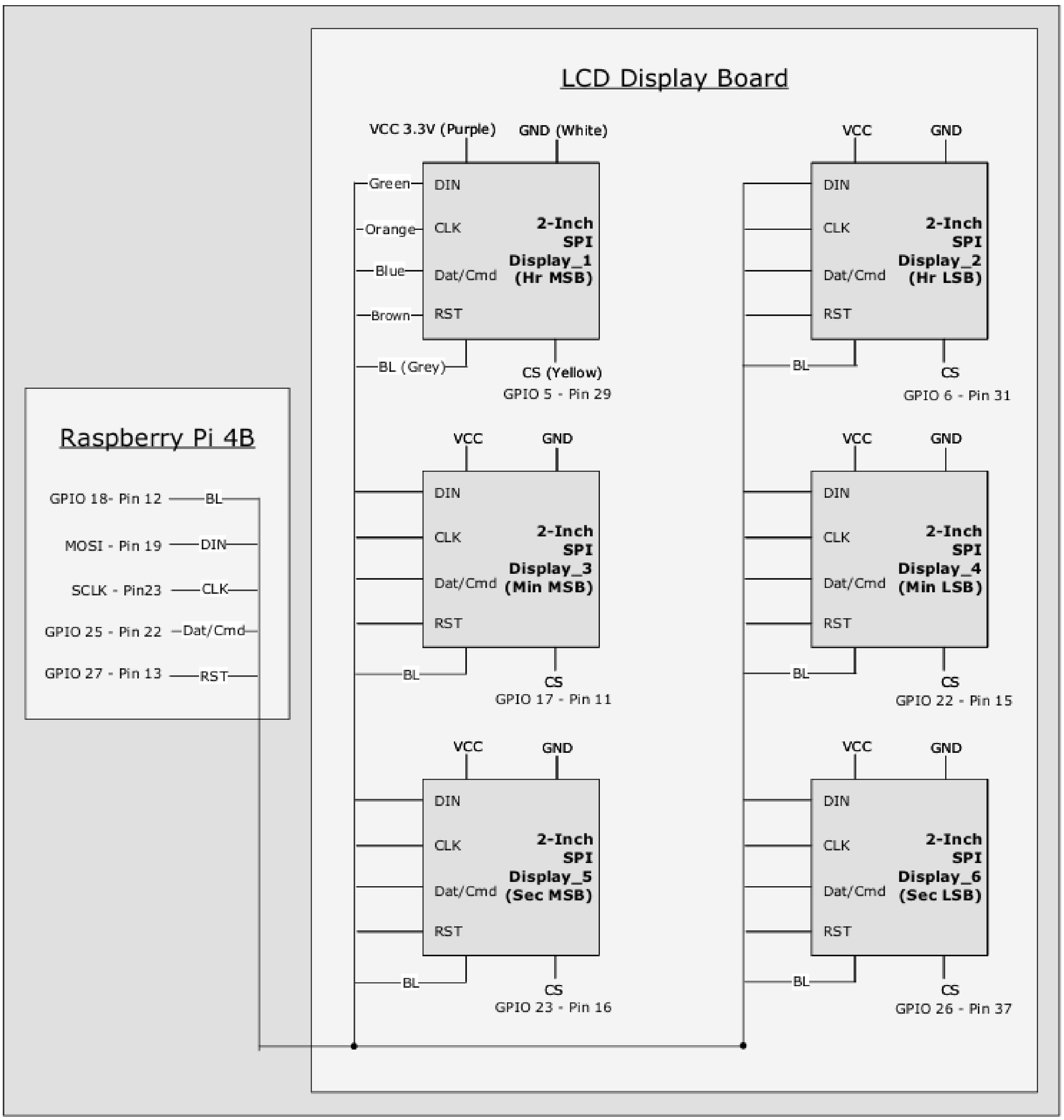
Photograph 2 – Back View 1

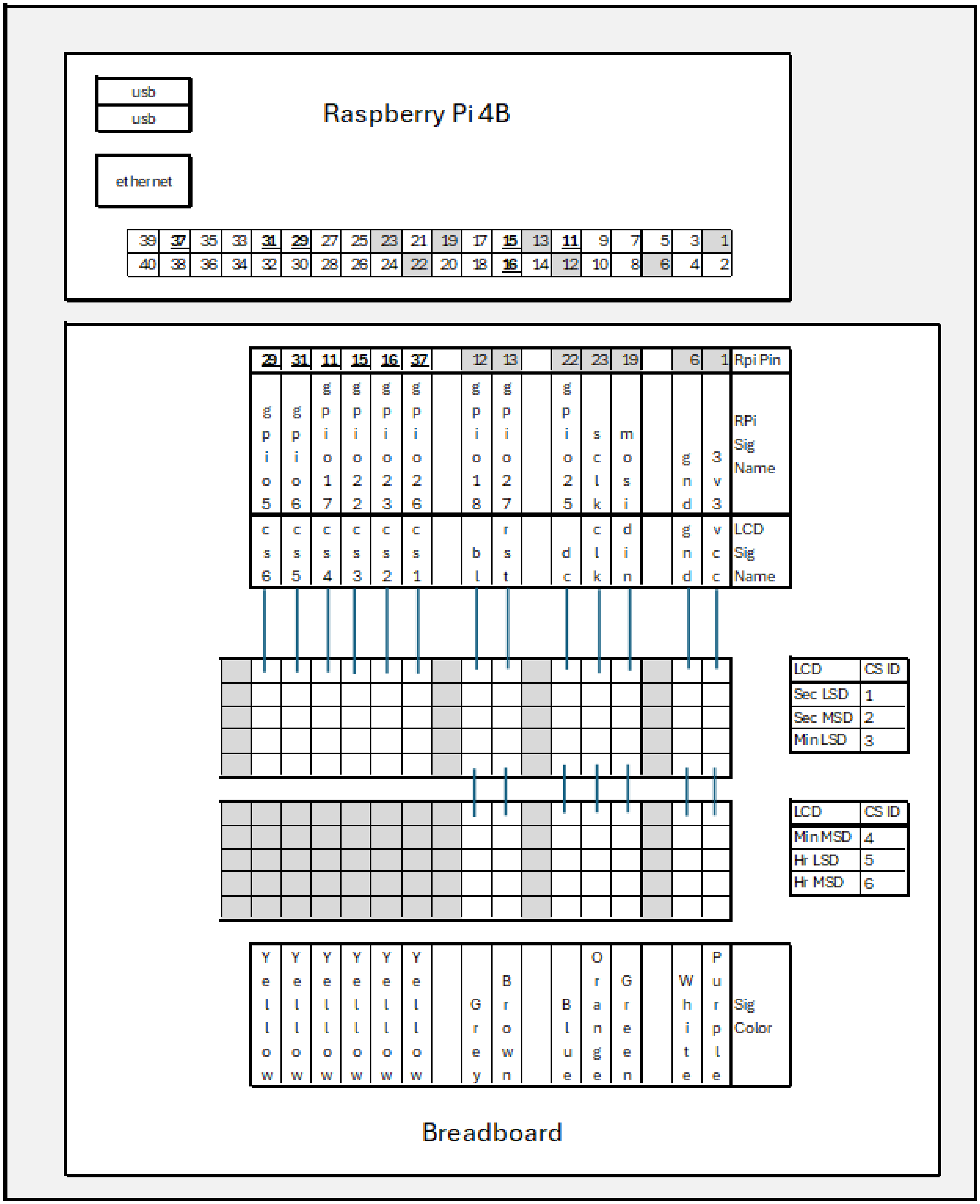
Wiring Diagrams

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 8 connection points and 7 of these are ALL common to each LCD. For example, pin 19 on the RPi (the Data In pin) is connected to all 8 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 1 of the 6 LCD CS signals low at a time.

Functional and Physical wiring diagrams are provided, respectively, in the following two diagrams.

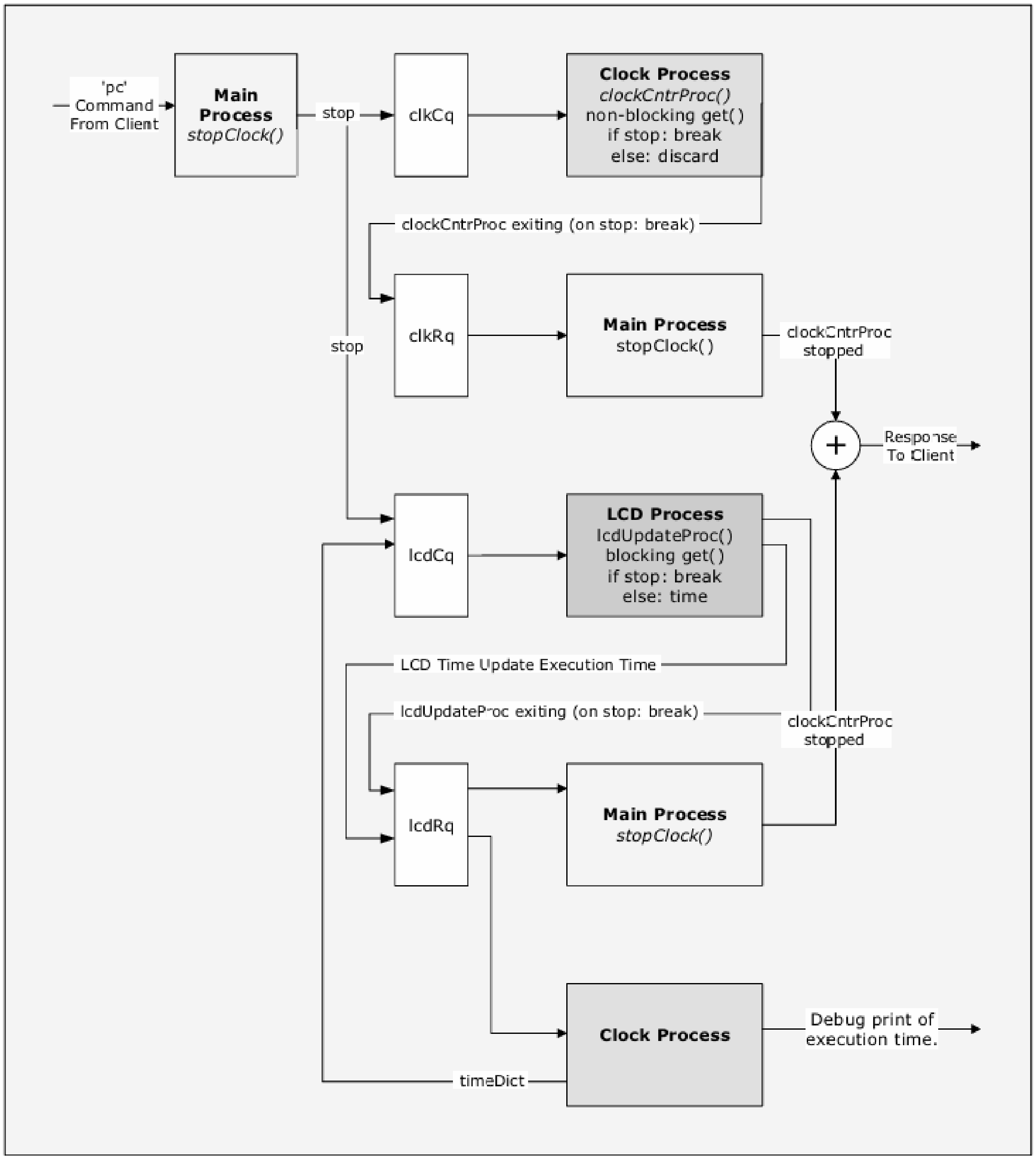



usb
usb
ethernet
Raspberry Pi 4B
39 37 35 33 31 29 27 25 23 21 19 17 15 13 11 9 7 5 3 1
40 38 36 34 32 30 28 26 24 22 20 18 16 14 12 10 8 6 4 2
29 31 11 15 16 37 12 13 22 23 19 6 1
Rpi Pin
gpio5 gpio6 gpio17 gpio22 gpio23 gpio26 gpio18 gpio27 gpio25 sclk mosi gnd 3v3
RPi Sig Name
cs6 cs5 cs4 cs3 cs2 cs1 bl rst dc clk din gnd vcc
LCD Sig Name
LCD CS ID
Sec LSD 1
Sec MSD 2
Min LSD 3
LCD CS ID
Min MSD 4
Hr LSD 5
Hr MSD 6
Yellow Yellow Yellow Yellow Yellow Yellow Grey Brown Blue Orange Green White Purple
Sig Color
Breadboard

Communication Queues

Control of the clock is accomplished by running a python script (client.py) on a remote machine (PC, Phone). The clock on the RPi runs within a server on the RPi and the server will communicate with a remote machine running a client. Information on this client/server implementation can be found in Appendix 1.

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 three processes communicate with each other using four multiprocessing-communication-queues. Two queues are used for sending commands and the other two are used for receiving responses. A simplified communication diagram (for the 'stop' command) is presented below.



Running the Command Line Client

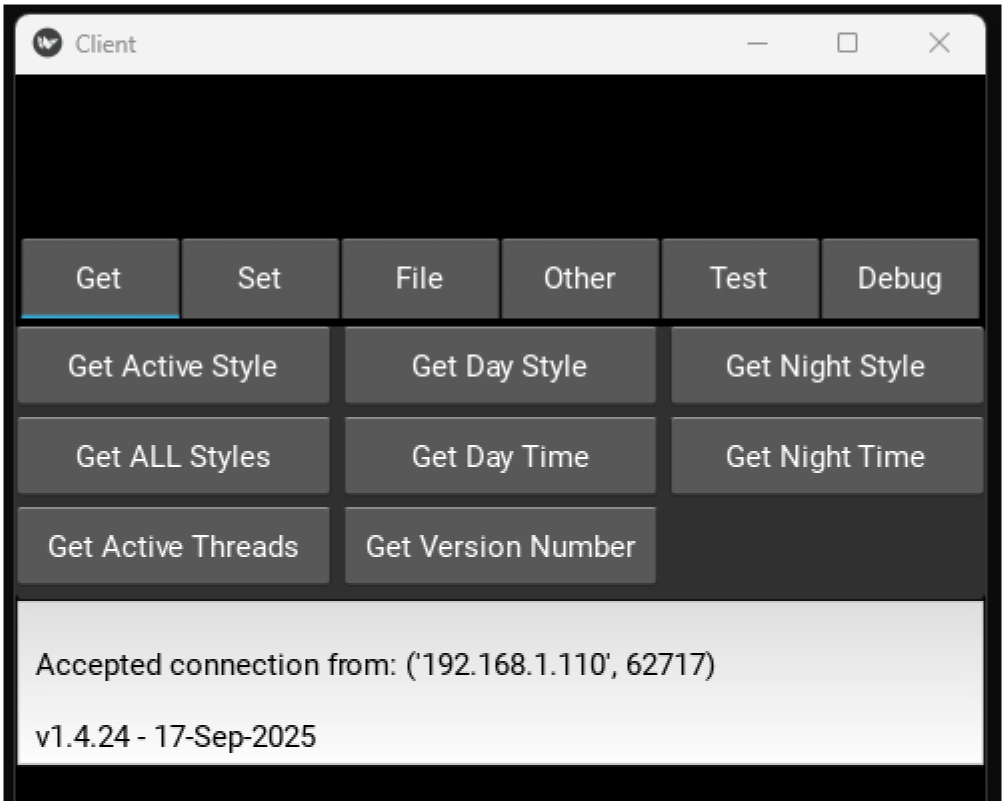
When a command line client is started and a connection is accepted by the server a prompt is presented. When ‘m’ is entered at the prompt a list of available commands is provided. A screen shot of this presentation is provided below.

```
PS C:\01-home\14-python\gitTrackedCode\spiClock> python .\client.py
 sndBufSize 65536
 rcvBufSize 65536

 Accepted connection from: ('192.168.1.110', 58479)


 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) ->
```

Running the GUI Client

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.



Appendix 1. A Python Based Client-Server Architecture

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

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.

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.

The worker functions associated with all commands are contained within file cmdWorkers.py with the exceptions of the close and ks commands. The work associated with the close and ks commands is performed in file server.py directly.

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.

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.

Some Assembly Required

In file client.py on approximately lines 60 and 61 the following two lines of code are present:

```
connectDict={'s':'localhost','l':'00.00.00.00','i':'00.00.00.00'}
PORT =
```

Likewise in file server.py on approximately line 164 the following line of code is present:

```
port =
```

For all connection types (refer to Figure 1) a port number needs to be specified. The number used must be the same in both the client and the server files. Use a number greater than1024 – between 5,000 and 50,000 is safe.

For connection type 2, in addition to the port number, the IP of the server needs to be entered (value for key `'l'`) needs to be entered. The address can be found via the ipconfig 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 entered (value for the key `'i'`) needs to be entered. 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 number you entered client.py and server.py) 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

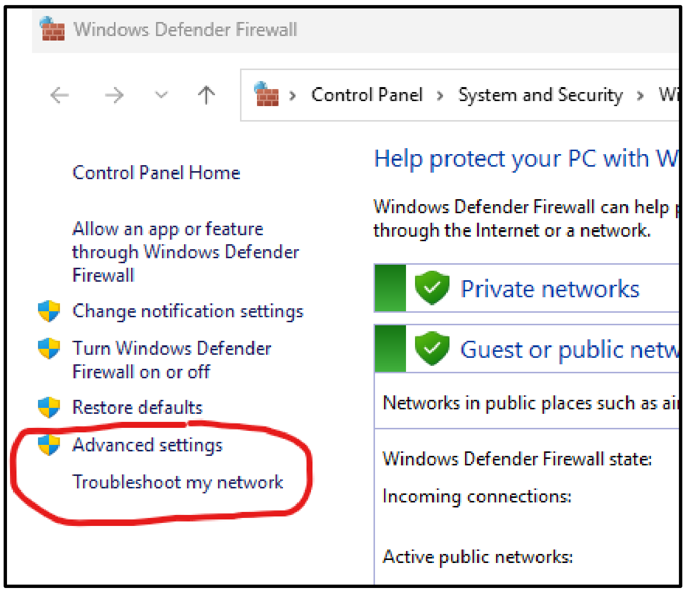
A Potential Pitfall – Firewall

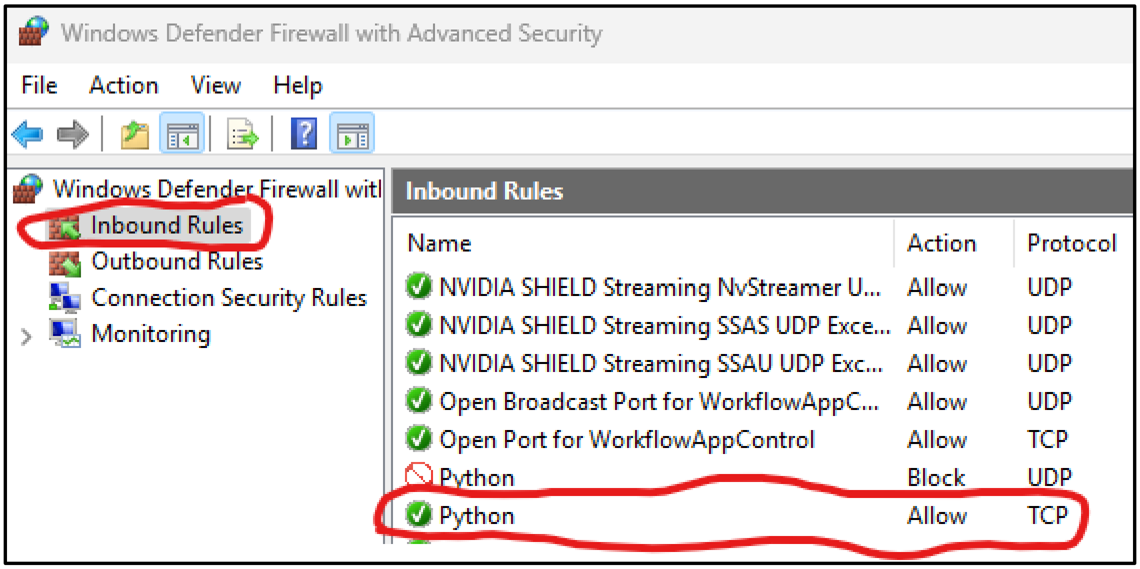
The above was all initially done with the server running on a Raspberry Pi. The Raspberry Pi runs Linux and by default its firewall is disabled. As such, all connection types worked only by performing the "SOME ASSEMBLY REQUIRED" steps outlined above.

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.

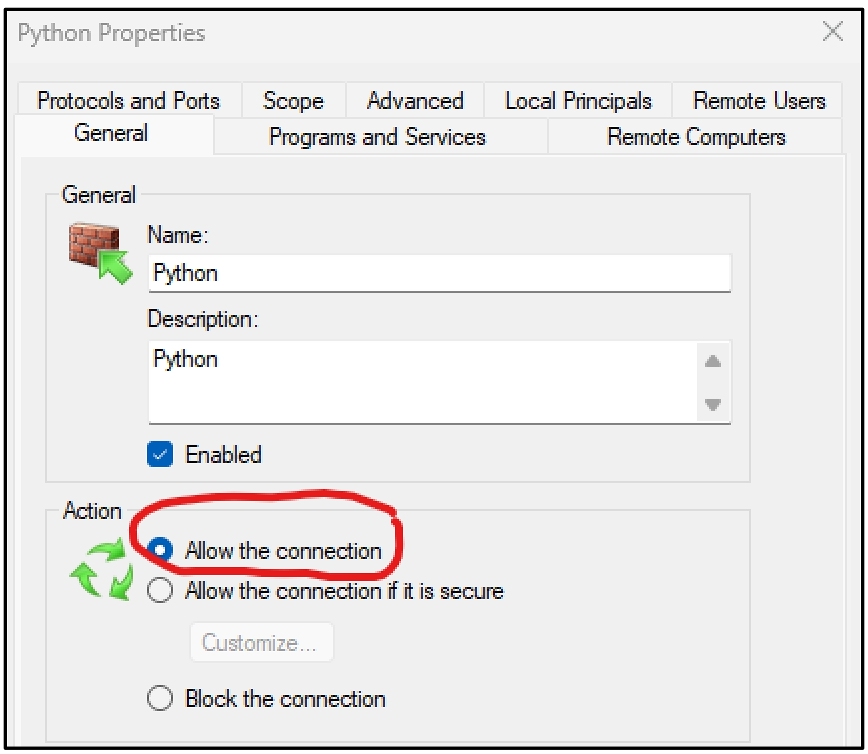
These are the basic steps:

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



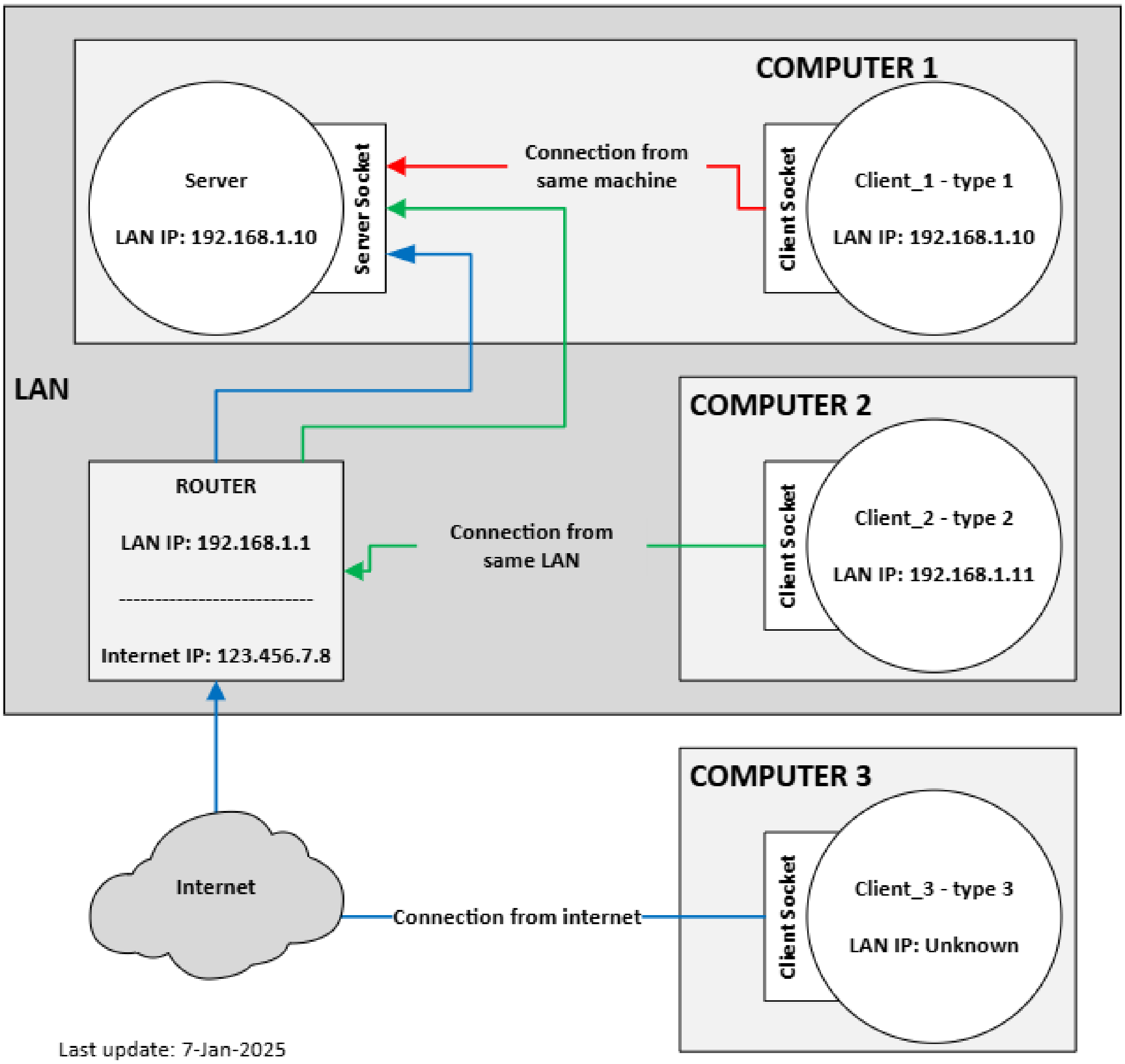


Right click on the Incoming Rule for Python TCP Protocol, select the General Tab and change to Allow the connection.



Explanatory Figures

Figures 1 and 2 illustrate the various connection types and the functional call tree, respectably. This client-server architecture was used in the design of a Raspberry Pi sprinkler controller and a functional block diagram and a wiring diagram for that are provided in figures 3 and 4.



Last update: 7-Jan-2025

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.

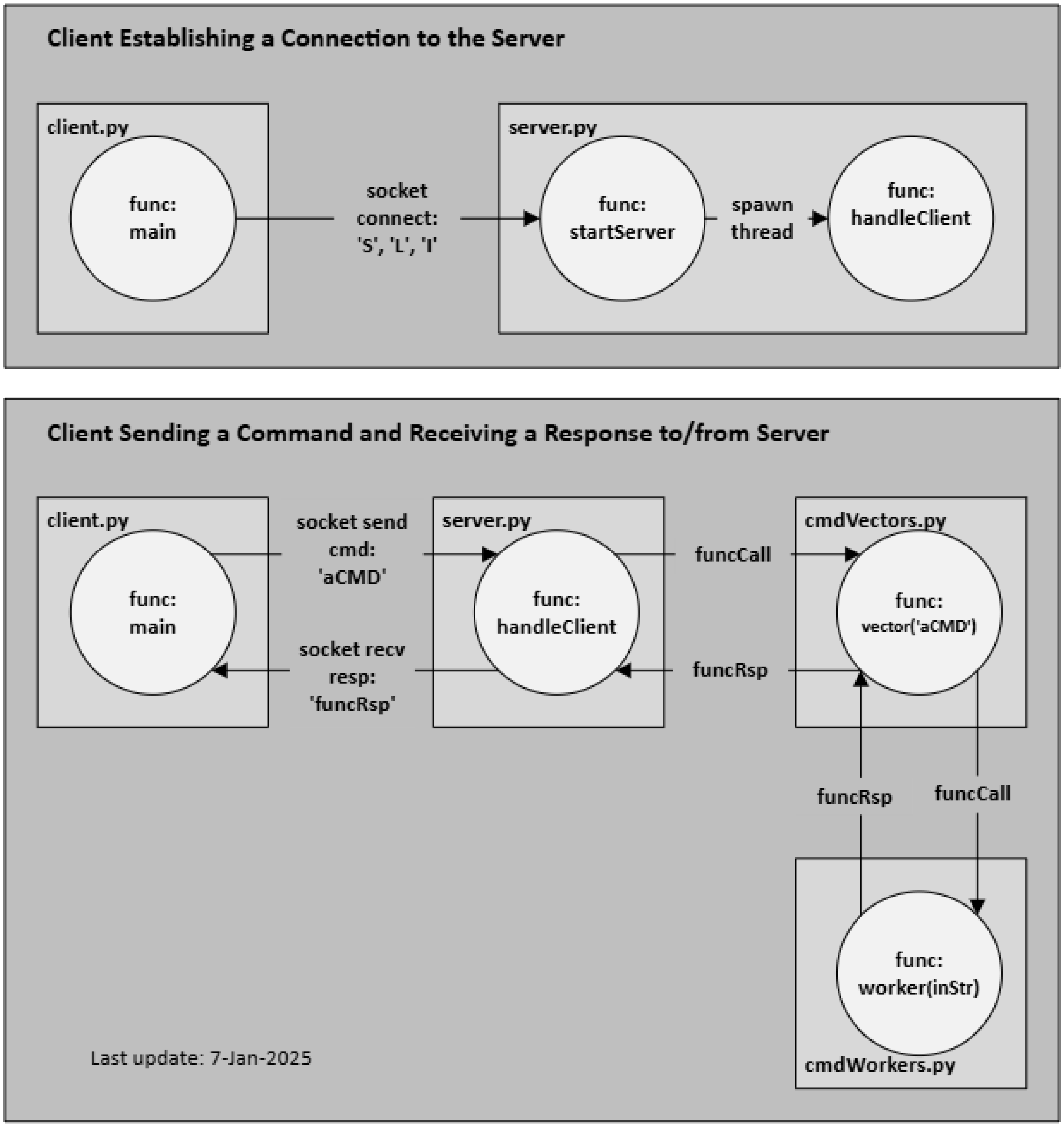
Computer 1 is running both the server and the client so the obvious connection type is type 1, as shown. However, computer 1 could also connect over the LAN or over the internet (neither shown).

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 1. Connection Types



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 2. Function Call Tree