A Python Implementation of a Client-Server Architecture

21-Jan-2025

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.

This document and its associated python code can be found here:
https://github.com/sgarrow/sprinkler2

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 RELATIONSHIP

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.

A Python Implementation of a Client-Server Architecture

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 Python Implementation of a Client-Server Architecture

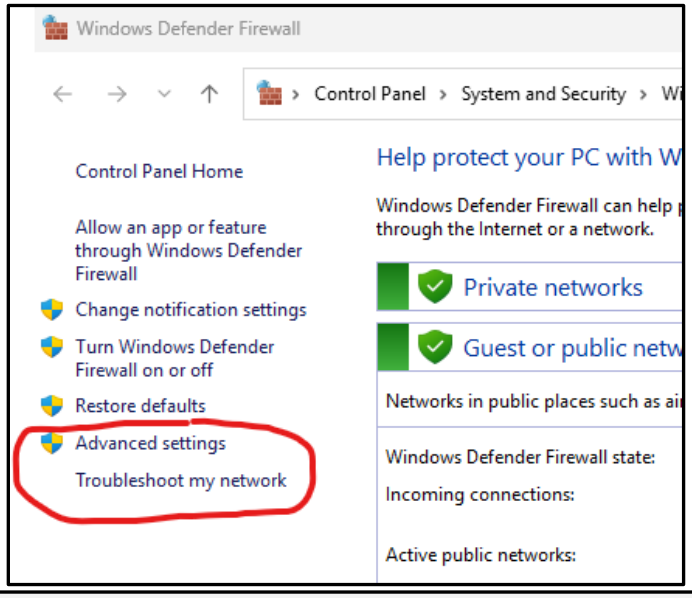
A POTENTIAL PITFALL – FIREWALL

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.

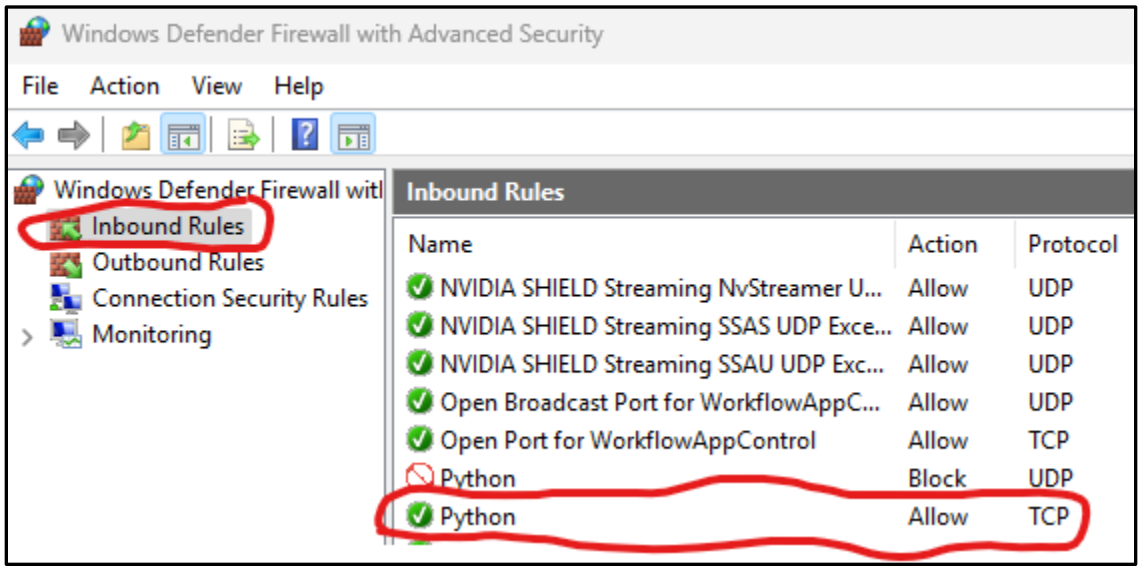
On windows to get all connection types working, specifically connection type 3, the Windows firewall will need to be changed to allow incoming python TCP connections.

These are the basic steps:

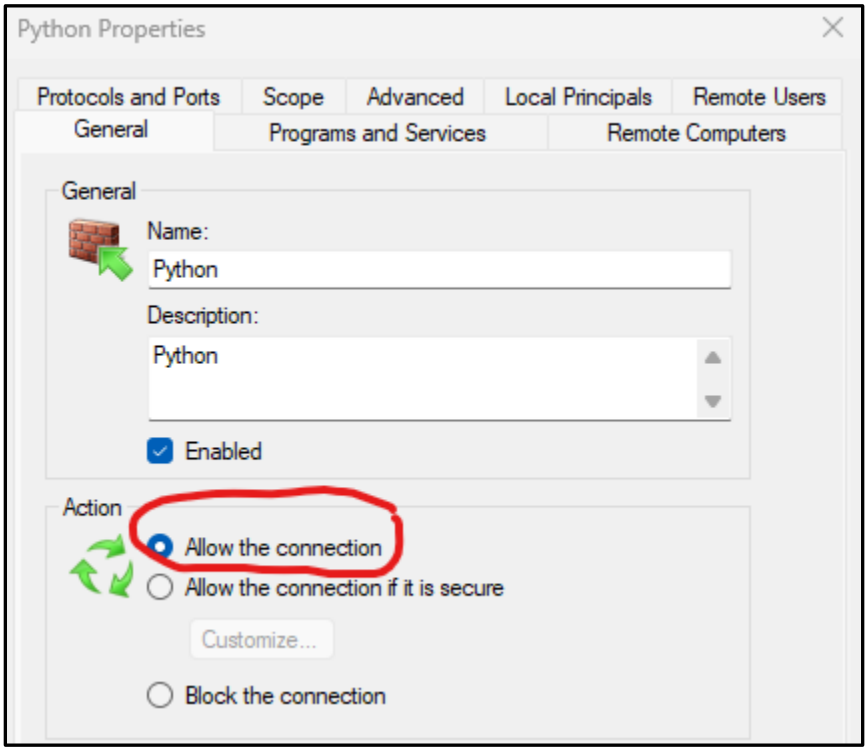
Control Panel\System and Security\Windows Defender Firewall ---> Advanced settings ---> Connection Security Rules ---> Inbound Rules



A Python Implementation of a Client-Server Architecture



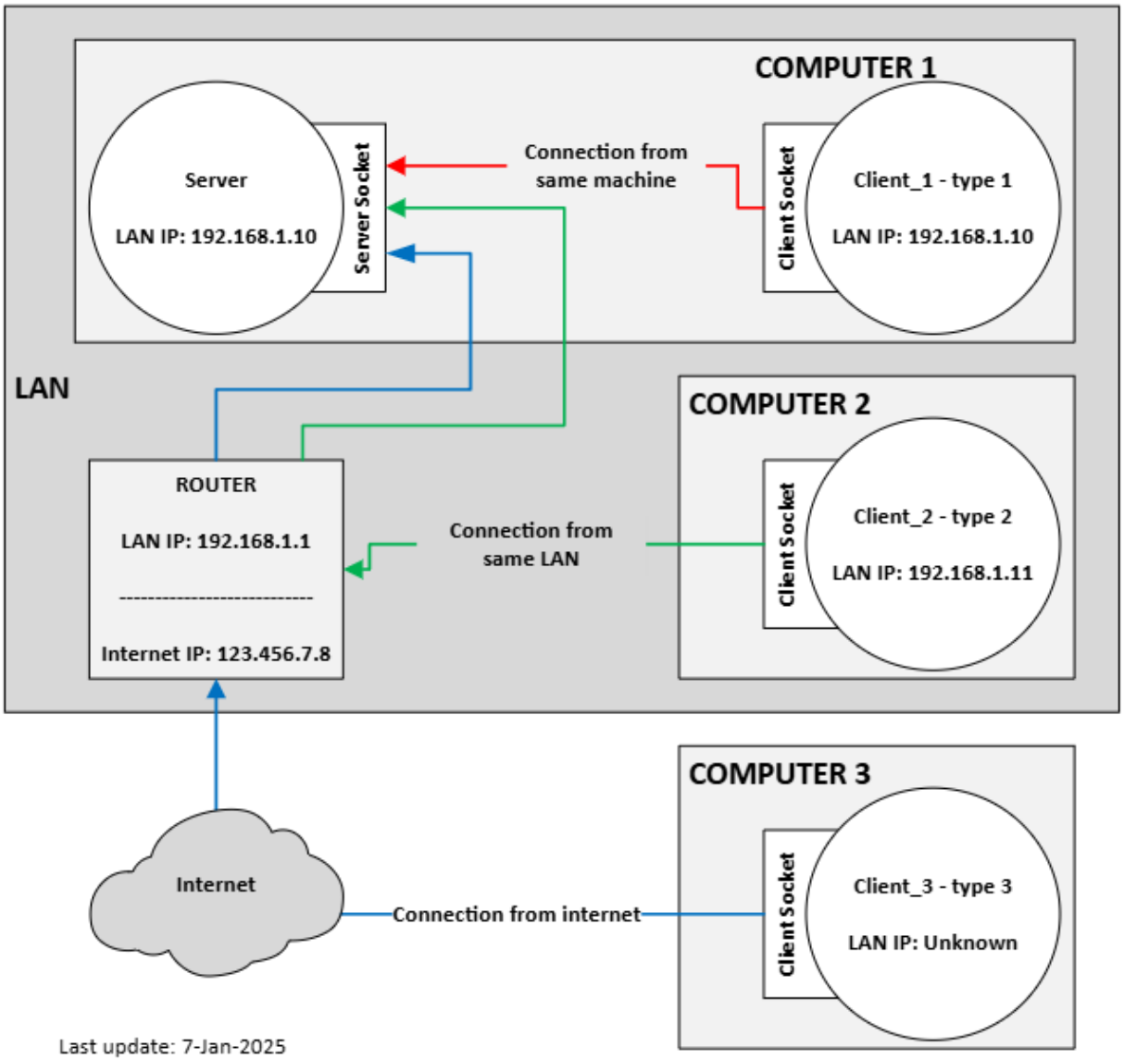
Right click on the Incoming Rule for Python TCP Protocol, select the General Tab and change to Allow the connection.



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.

A Python Implementation of a Client-Server Architecture



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).

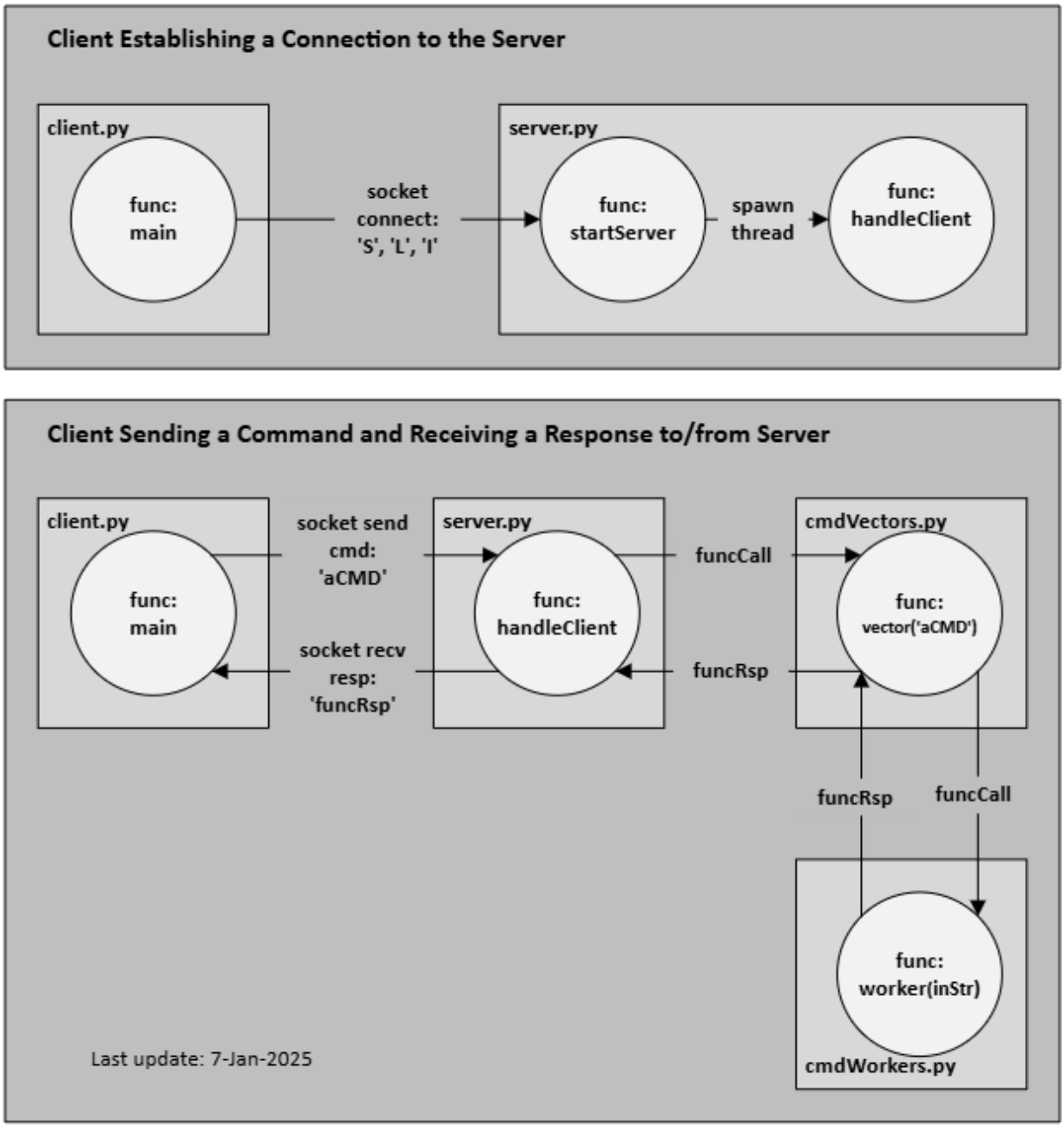
Computer 2 is running only the client but is on the same LAN as the computer running the server so the obvious connection type is type 2, as shown. However, computer 2 could also connect over internet (not shown). Computer 2 can not connect to the server via a type 1 connection.

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 Python Implementation of a Client-Server Architecture

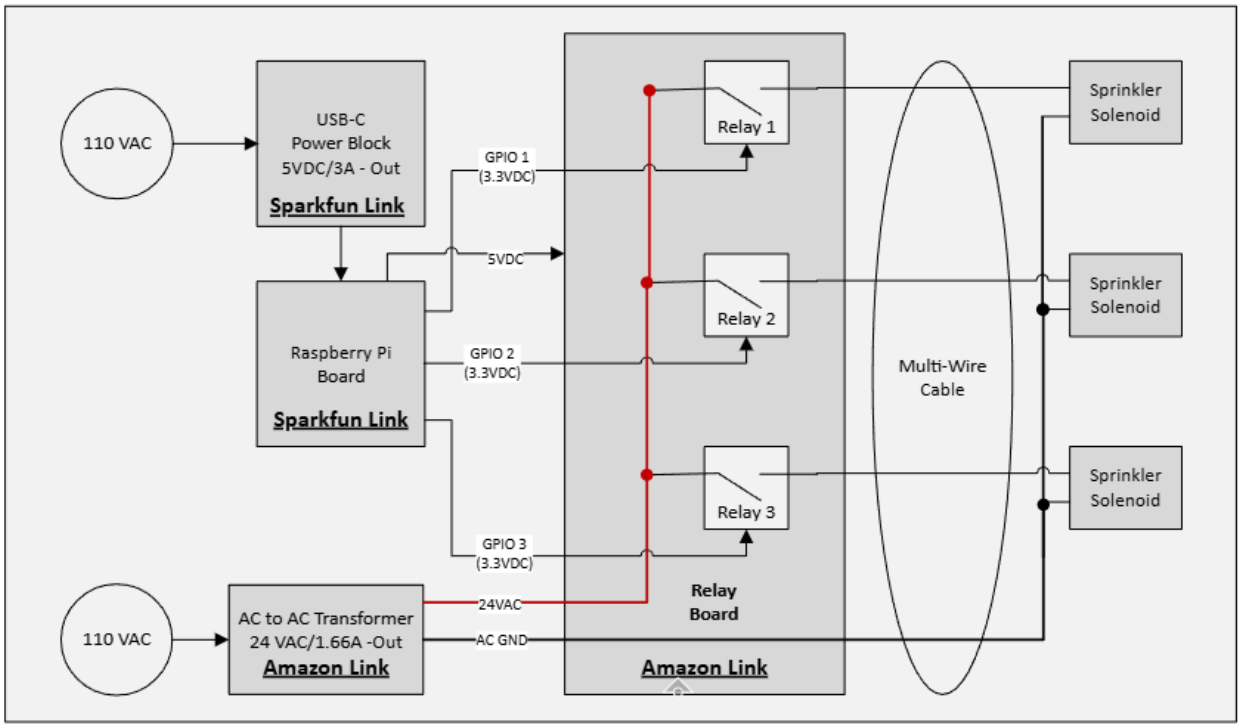


A client's connection request gets transmitted to the server over a socket where it is recieved by function startServer. When the server accepts the connection it spawns a thread that runs function handleClient that is then dedicated to servicing commands recieved from that client.

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

A Python Implementation of a Client-Server Architecture



Relay Board The GPIOs are 3.3V with max current of 16mA. The GPIOs don't actually control the relay but control a 5V (sourced from the RPi) signal that goes to and controls the relays, the diagram is a bit misleading. A relay closed relay draws 16mA on the 3.3V and 6mA on the 5V = 3.3*.016 + 5*.006 = 0.083W from the RPi board.

USB-C Pwr Block The power block can supply 3.0A at 5V = 15W. The RPi itself draws about 1.0A @5V = 5W and the 8 relay control signals will draw 8*0.083 = 0.66W. Total power draw from the 15W supply is 1.66W.

AC/AC Transformer The xformer can supply 1.66A at 24V = 39.8W. Solenoids draw 0.6A at turn-on and 0.2A at steady state. Solenoids draw 0.6A*24V = 14.4W at turn-on and 0.2A*24V = 4.8W at steady-state. Max solenoids that can be turned-on simultaneously 39.8/14.4 = 2.7.

Figure 3. Sprinkler Controller Functional Block Diagram

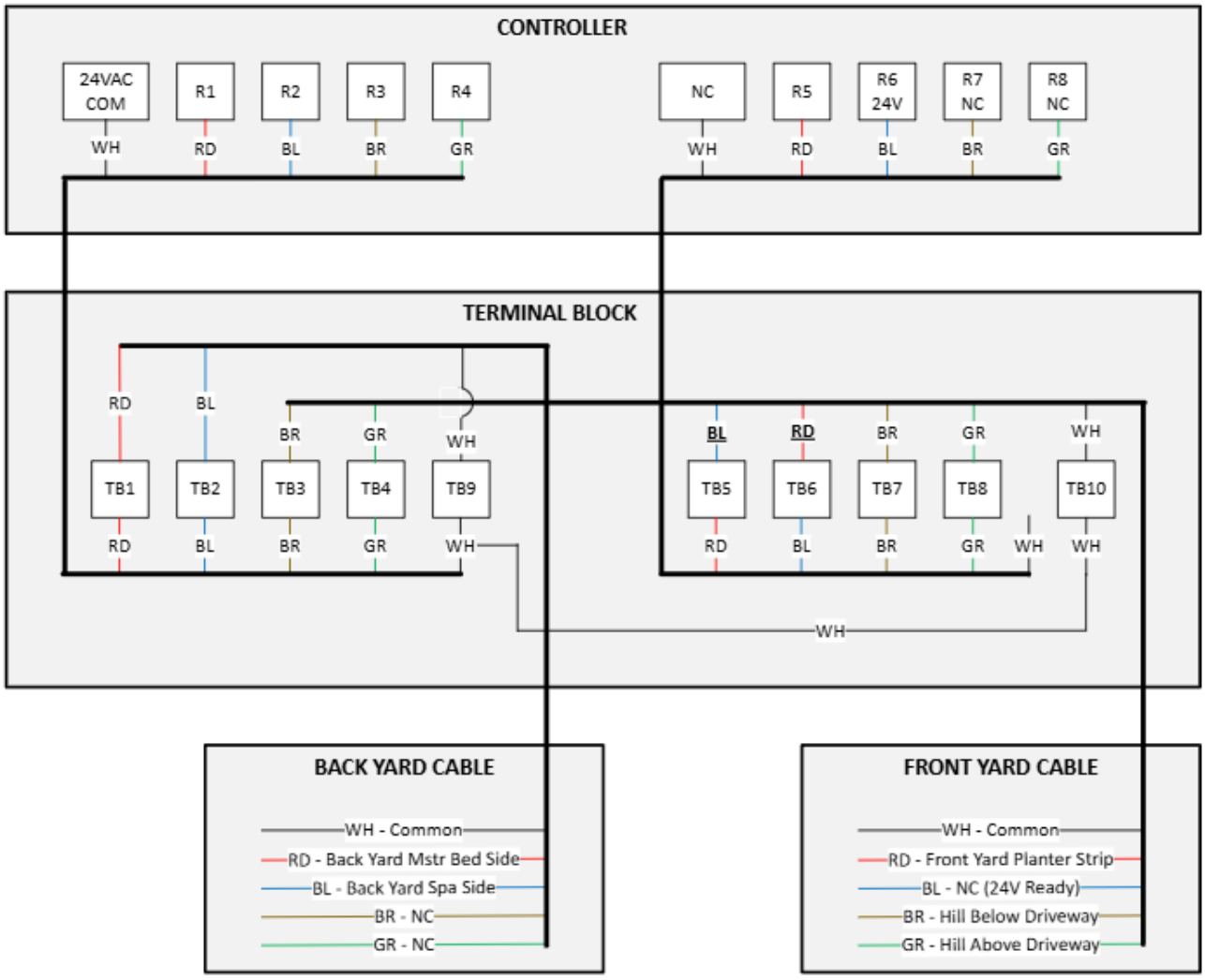


Figure 4. Sprinkler Controller Wiring Diagram.