This document and its associated python code can be found here:

https://github.com/sgarrow/sockets

STARTING THE SERVER AND THE CLIENT

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.

INTRODUCTION

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.

DETAILS

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.

UNEXPECTED EVENT HANDLING

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:

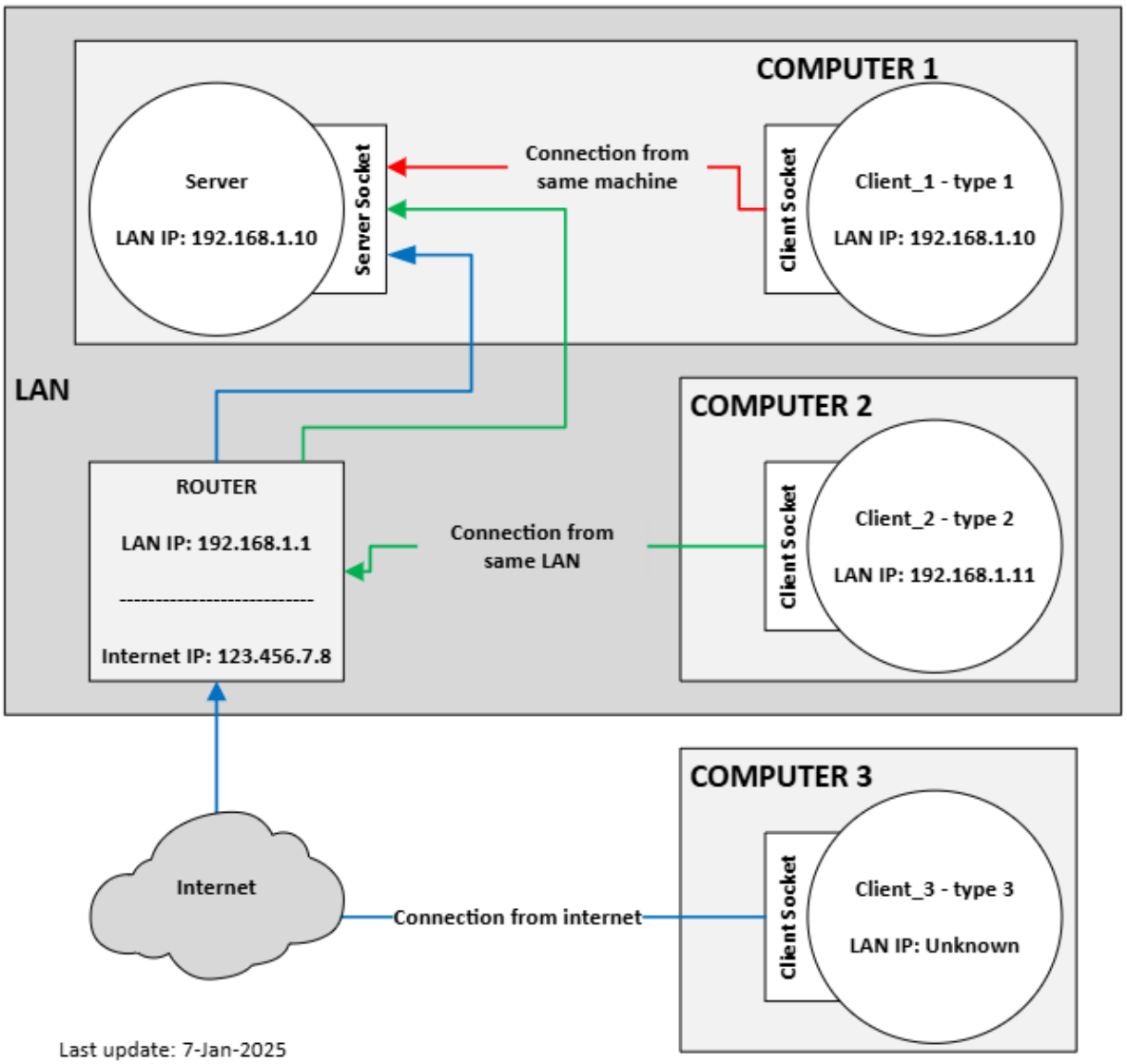
```
port =
```

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

For connection types 2, in addition to the port number, the IP of the server needs to be entered (value for the key `'l'`) needs to be entered. The IP address can be found using the ipconfig command in a command window open on the machine that will be running the server.

For connection types 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/



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 **Inter**connected set of Local Area **net**works).

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.

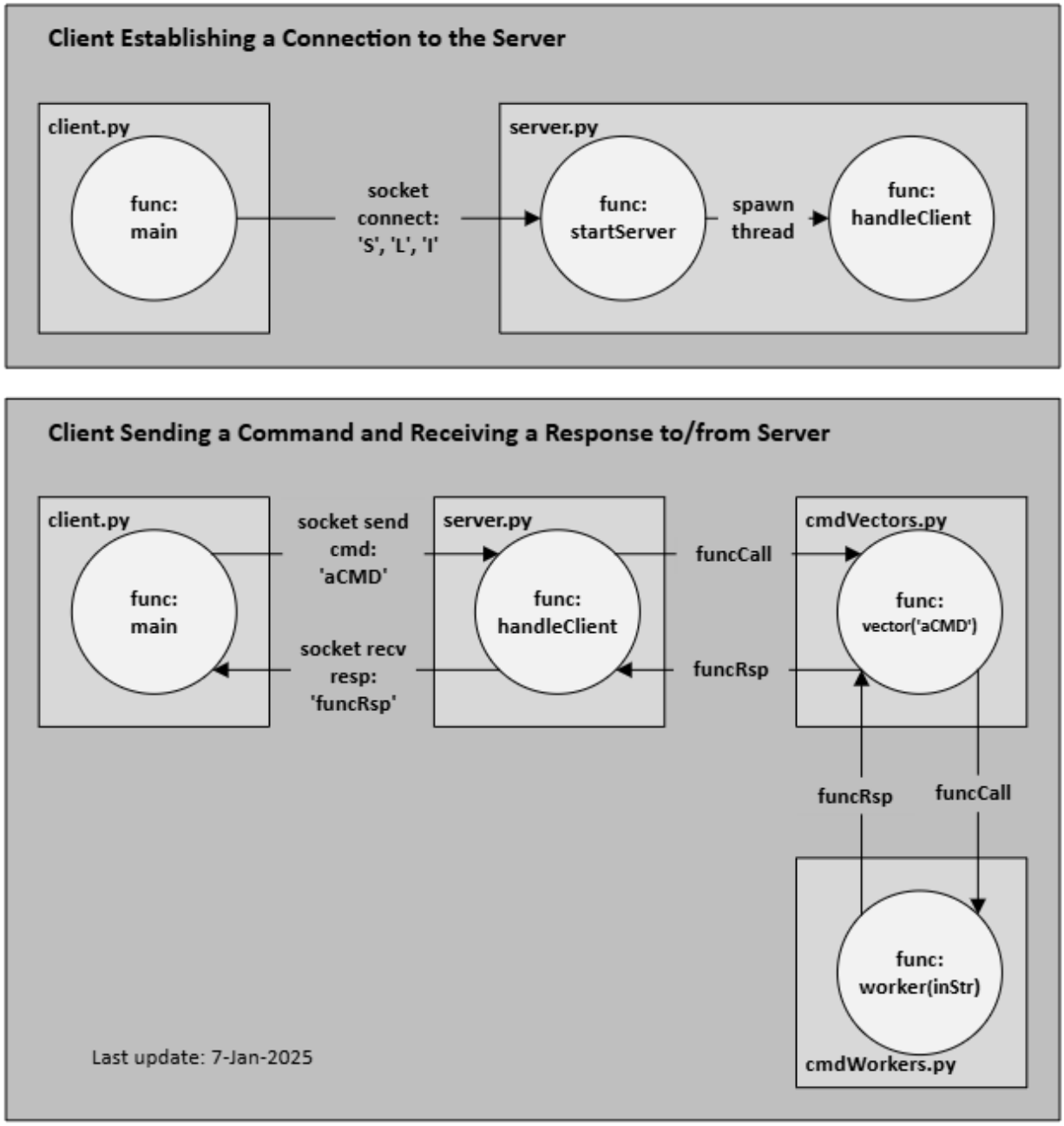
Computer 1 is running both the server and the client so the obvious connection type is type 1, as shown. However, computer 1 could also connect over the LAN or over the internet (neither shown).

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