Introduction to Sockets

David Croft

October 26, 2016

Abstract

This week we are looking at network communications. Specifically we are looking at how to use sockets in Python, the syntax may differ but the same concepts apply to most general purpose languages.

1 What are sockets?

Sockets are the entrances and exits for two-way communication links between two programs. You can imagine it like a tube running from one computer to another, everything you put in one end (socket) of the tube comes out of the other end (socket). At the moment we don't care about exactly how the link works or what route it takes, all we care about is that this 'tube' lets us communicate between programs on different machines or the same machine.

Look at the lab_getting_started code and try running it. The program is a very, very simple server.

Lab work:

1. Run lab_getting_started.
2. To connect to our server we are going to use a PuTTY.
   - Download it[a] and run it.
3. Open a connection to the lab_getting_started server.
   - Hostname is 127.0.0.1, port 12345 and connection type Raw.
4. What happens?

[a] `http://www.chiark.greenend.org.uk/~sgtatham/putty/`

You can also connect from the command line by typing:

```
$ putty -raw 127.0.0.1 -P 12345
```

Or you can use netcat on Linux (or Windows if it is installed) by typing:

```
$ nc 127.0.0.1 12345
```

1.0.1 Ports

Ports are used to allow a single computer to run multiple network processes simultaneously, each process communicates over it's own port.

Many port numbers have types of connections already assigned to them. For example port 80 is used for HTTP (web) traffic, port 22 is used for SSH. While it is possible to use other port numbers this adds an additional layer of complexity so it not recommended.

As a group you will need to agree on a port number to send your TicTacToe game data over:

- Ports 0-1023: System ports, hands off.
- Ports 1024-49151: User ports, can register port numbers with Internet Assigned Numbers Authority (IANA) to make it less likely another program will use the same port as you but there is nothing actually stopping you from using any of them.
- Ports 49152-65535: Private ports, do whatever you want in here.

For your game you are probably best off picking something in the user ports area, there are lists of ports that are already known to be in use available online[1].

2 Servers

Most interprocess communication (sending data between two different programs) uses a client-server model. This refers to the two processes that are trying to communicate with each other. One of the processes is the *client* which connects to the *server*. A server can have multiple clients connected to it, but a client will typically be connected to a single server.

So far our server is only able to send messages to the connected client but we are going to want to receive messages as well.

Lab work:

5. Look at the lab_getting_recv code and try running it.
6. What does this code do?

Notice the `encode()` and `decode()` functions. Sockets only transmit raw bytes, however as of Python3 text in Python is stored as unicode. This means that Python strings can contain every possible symbol you can imagine (including emoji ☺) but also means that each character can be made from several bytes. In order to send text via sockets, we have to convert between Python3 unicode strings and sequences of raw bytes whenever we send/recv data.

You can imagine it like a telegram system, messages are normal text (Python strings) at each end but have to be converted into morse code (byte sequences) in order to be sent over the wire.

[1] `https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers#Well-known_ports`

Lab work:

7. Modify the lab_getting_recv server so that it converts the messages it receives to uppercase before returning them.

So far we've created a server that can accept connections from clients and respond to them. How ever it only responds to a single message from each client before disconnecting them.

Lab work:

8. Modify your server so that it will keep replying to a client until the client disconnects.
    - What happens when the client disconnects?
9. Modify your server so that the server can handle clients disconnecting.

3 Clients

We've managed to create a simple server and have been using Putty and/or netcat for our clients, but how do we create our custom client program? It's pretty much the same a with a server expect that a client doesn't need to open a listening socket, it just connects straight to a server.

Lab work:

10. Look at the lab_getting_started_client code and try running it at the same time as your server.
11. What does the code do?

So far the client code can only send messages but we're going to want to be able to receive messages as well.

Lab work:

12. Modify the client code so that it can receive the server's response to the messages it sends.

3.1 Blocking

Congratulations you've created your first server and client programs.

1. Try having two clients connect to your server at the same time. What happens?

There is a problem with `send()` and `recv()` however. In the default configuration, if your code calls `recv()` it will wait until at least one byte of data has been received, regardless of how

long that takes. If the data is never received (it never got sent, network problems etc) then it will wait forever. The same applies to `send()`, only it will wait until the data is sent however long that takes.

This is obviously not very useful for programs such as chat servers where we never know when someone might decide to send a message. How do we solve this problem?

1. If we can't send or receive then don't block.

   This can be achieved if we change the socket settings to non-blocking. In Python this is done by calling the `setblocking(0)` function on the individual sockets. If the socket is not able to immediately `send()` or `recv()` then it will raise an Exception that you can catch.

2. Only try sending and receiving when we know that we can.

   The easy way to do this is using the `select` module and in particular the `select()` function. This takes 3 arguments[2]; a list of sockets we want to check are readable, a list of writable and a list of ones that might have errors. It returns 3 lists, a list of sockets that are readable, a list that are writable and a list of ones with errors.

Lab work: Blocking

13. Look at an run the lab_getting_select server. Try having multiple clients connect to it, what happens?
14. Modify the server so that messages from one client are relayed to all the other clients.
15. Put your chat server on your Rpi, try connecting to it with multiple clients on multiple machines.

4 Summary of Deliverables

You have two weeks for the networking, at the end of the two weeks you are expected to have completed the following tasks:

1. Have simple echo servers working (you send the server a message, it replies with the same message).
2. Have a simple chat server working
3. Start thinking about how you are going to get your game to work over the network. What sort of messages are you going to need to send? This would make a good blog post.

4.0.1 Extension task

1. Modify your chat server so that clients are assigned a random nickname when they connect that is appended to the start of any messages they send.
2. Allow clients to change their nicknames.

[2] 3 mandatory, 1 optional.

4.0.2 Extended Extension task

Learning about sockets and select is important as it lets you understand what's actually happening in your software. However, there are libraries and tools that will do most of the socket handling for you.

Extended work:

16. Investigate the asyncore library.
17. Can you recreate the chat server in it?