PIC 16, Winter 2018 – Preparation 10M

Assigned 3/7/2018. To be completed by class 3/12/2018.

Intended Learning Outcomes

By the end of this preparatory assignment, students should be able to:

- define or describe `socket`, IP address, port number, protocol, TCP, and UDP;
- distinguish between the roles of client and server;
- `bind` a server socket to an IP address and port number and instruct it to `listen` for and `accept` incoming connection requests;
- `connect` a client socket to a server socket's IP address and port number; and
- `send` data, `receive` data, and `close` the connection between two connected sockets.

Tasks

- ☐ Watch the first half (16.5 minutes) of Python Advanced Tutorial 6 – Networking. While you're watching, please follow the notes below, presented in an order that corresponds with the video.
- ☐ "Local area network" – The PIC Lab is an example of computers on a local area network. In order for the computers to communicate with one another, the signal probably doesn't have to leave the room. It's *local*, which is in contrast to the Internet, which is *global*.
- ☐ "Client" and "Server" – I suggest definitions for these terms that are different from those presented in the video:
  - Client – the device that initiates the connection
  - Server – the device that accepts the connection

  When you call a friend on the phone, you are the client because you are initiating the connection. When your friend picks up, she is the server because she accepted the connection.
- ☐ Don't worry about the distinction between Client/Server and Peer-to-Peer for now. Remind me to talk about that in class.
- ☐ "IP address"– An Internet Protocol address is like a telephone number that a computer uses to "call" another computer. An example is `8.8.8.8`, which is the IP address of a particular Google computer. You can "call" and see the response from this Google server by typing at an OS command prompt (OS X terminal/Windows command console):
  ```
  ping 8.8.8.8
  ```
  On my Mac, I see something like:
  ```
  PING 8.8.8.8 (8.8.8.8): 56 data bytes
  64 bytes from 8.8.8.8: icmp_seq=0 ttl=56 time=16.988 ms
  ```
  This means my computer sent out 56 bytes of data (probably random) and received 64 bytes back (also probably meaningless) and the round-trip time from sending to receiving was about 17ms. This tool (`ping`) is not used to send meaningful data, it's just to test connections.
  `127.0.0.1` is a special IP address that allows a computer to call itself, as if there were a telephone number you could use to call your own phone. You can try `ping`ing this address, too. Note the connection is much faster. For more information on `ping`, see this.
- ☐ IP address are assigned to computers like your phone carrier assigns your phone a telephone number. You can determine your computer's IP address by searching Google for "what is my IP".
- ☐ *Programmers get to assign port numbers to their own programs*, they are like telephone extensions assigned by… well, somebody local. You can typically choose whatever port number you like to use to identify your program, as long as it's outside the reserved range 1-1024.
- ☐ "TCP and UDP" – "Transmission Control Protocol" and "User Datagram Protocol". Wait until later to understand the difference, just know now that they are different *network protocols* –

formal rules that computers obey when talking with one another. Given that computers only really speak `1`s and `0`s (no words), rules are important to understanding what the `1`s and `0`s mean based on context. The speaker does a great job of explaining the difference shortly after the 16.5 minute mark.

- "Socket family" and "Socket type" – don't worry about these; they're optional. We'll use the defaults.
- The speaker refers to `socket` as a "method" and a "constructor". I'm not sure what it is (it seems like it might just be a non-member *function* called `socket` in the `socket` module), but in any case it returns some sort of object that we'll call a `socket`. We'll use it like:
  ```
  s = socket.socket()
  ```
  To give us a `socket` object `s`. Currently this object doesn't know what its IP address (telephone number) or port number (extension) are. It just exists and can't do anything yet.
- The speaker hasn't mentioned it yet, but when you write networking applications, like a chat program, you basically have to write two programs – one for the client that initiates connections, and one for the server that waits for and accepts incoming connection requests.
  Let's say we want to create a connection between programs on computers A and B so that they can send messages to one another.
  1. *First*, computer B starts up the *server* program, and it starts waiting for connection requests.
  2. *Then*, computer A runs the *client* program, which requests a connection with the server program on computer B.
  3. *Finally*, computer B's server program accepts the request.

  Now the two programs are connected and can start sending messages between one another. If this seems abstract, relate this to the example of you (A) calling a friend (B) on the phone.
- `bind`, `listen`, and `accept` are all member functions (methods) of `socke`t objects that are used on *server* programs by *server sockets*, that is, sockets that have not yet been connected to a client socket.
  - `bind` tells the server socket what its IP address and port number are. The IP address is the IP address of the computer the server program is running on. Again, you can determine this by asking Google "what is my ip". The port number is any integer (you choose) within the acceptable range (1025 – 64,000 something).
  - `listen` tells the server socket to start waiting for incoming connection requests.
  - `accept` tells the server socket to accept a connection request when it receives one (from a client). The method "blocks" – doesn't finish, doesn't return - until a connection request is received. When the server socket receives a connection request, `accept` returns a *new* socket object, which I'll call a *connected socket*, that represents the connection established between the computers. A *connected socket* is different from a server socket because it is connected to a socket on another computer. The server's *connected socket* can send data to the other computer's socket and receive data from the other computer's socket, whereas a *server socket* could only listen for and accept incoming connection requests.
- `connect` is a method of `socket` objects that is used on *client* programs by *client sockets*, that is, sockets that have not yet been connected to a server socket. The hostname argument is the IP address of the computer that the server program is running on. The port number is the port number chosen by the server program. If `connect` is successful, that is, if a server program at the given IP address and port number is `listen`ing and `accept`s the connection request, then `connect` will convert the client socket to a connected socket, one that is connected to the server's connected socket. Now the client's connected socket can send data to the server's connected socket and receive data from the server's connected socket.

- `recv`, `send`, and `close` are methods that can be used by connected sockets on both client and server programs. Once the connection has been established, either the client or server program can `send` data, `receive` data, or `close` the connection.
    - If program A `send`s while program B is `receiving`, program B's `recv` method will return the data sent by program A.
    - The "bytes" parameter of `send` is the data to be sent, like a string.
    - The "buffer" parameter of `recv` is the maximum number of bytes to return.
    - Make sure the argument of `recv` (buffer) is enough to hold the data (bytes) sent by `send`.
    - `close` is like saying goodbye and hanging up. It severs the connection between the two sockets.
- The speaker writes his programs in a text editor and runs them from the OS console. You can write follow along by writing these programs in Spyder *and run them in separate Python/IPython consoles* if you prefer. That is, after you run your server program, start a new Python/IPython console, then with the new console selected, run the client program.
- Note that the example is for having two programs on the same computer communicate, hence the use of IP address `127.0.0.1`. Communication between two computers is no more difficult (as long as the two computers are allowed by their security settings to talk freely to one another, which is not a problem in the computer lab). Instead of using `127.0.0.1`, you would use the IP address of the computer running the server program. Instead of two separate Python/IPython consoles on the same machine, you use a Python/IPython console on two separate machines.
- After watching the video and reading these notes, you may find the following helpful for reference:
    - Python Network Programming
    - `socket` module documentation
- You might want to watch the next few minutes of the video (or all of it) to see how UDP works, which will make the difference between it and TCP clear. TCP is like a telephone conversation – the client calls, the server answers, and then the two can talk to one another until somebody hangs up. UDP is somewhat like a text message conversation. Users send messages to one another and their phones usually receive messages from others, but there is no long-term "connection" between the two phones, and sometimes text messages get lost. But I think we'll stick with TCP in this class, so you won't be quizzed on UDP next lecture.