CS 346 Cloud Computing

# Lab: Networking

## Overview

In this lab, you'll be creating a tiny networking tool in Python. If you have extra time, you are welcome to also create a more complex version - but if you do, please give it a different name. For the sake of grading, make sure that the **required** files have exactly the names I've specified, and that they are small, simple Python programs.

### Required Files

For this lab, you are required to turn in two files, each of which is a small Python program:

```
net_server.py
net_client.py
README.txt
```

The server program will open up a port and listen on it. When it accepts a new incoming connection, it will listen on that connection until the client closes it down; for each message it receives, it will reply back with a modified version of the mesage. When the connection closes, the server will then attempt to accept a new connection; it will keep doing this until it is killed.

The client program will connect to the same port, and send messages to the server. It will close the socket when it is done.

The README file will describe how to run the two programs; for instance, if you enhance your program to have any command-line arguments, document them here.

Details on both programs are below.

### Imports

Both of your programs must import the library `socket`, so that you can access the socket functions.

# 1 The Server

Your server program must first create a listening socket. To do this, it must perform three steps:

- Create the socket
- Decide what port it will be attached to
- Start listening on it

Once it has started listening, the operating system will allow incoming connections from other programs. Accepting is easy, you simply:

- Call `accept()`, which will block until a connection comes in

Once you have a new connection established, you will exchange data with the other end. This involves three steps, in a loop:

- Receive new data, and turn it into a string
- Modify the string in some fashion
- Send the data back to the client

## 1.1 Creating a Socket

To create a new socket, you must call:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

We don't have time to discuss all of the possible socket types in this lab - but you can do a lot of reading online. Short answer is, this call creates a socket which is (a) connected to the Internet, and (b) uses the TCP protocol.

## 1.2 Binding to a Port

Listening sockets must be "bound" to a port before they can listen. To do this, you must assign it a local address and a port number:

```
addr = ("0.0.0.0", choose_a_port_number)
sock.bind(addr)
```

You may choose any port number between 10000 and 65535, but try to make it unusual - don't use the first 5 digits of $\pi$, for instance. (If you choose the same port number as another group, you won't be able to both run at the same time, on the same machine.)

## 1.3 Listening and Accepting

To start your socket listening, you must call

```
sock.listen(5)
```

**Only do this once;** once a socket is listening, it will continue listening forever, until your program dies, or you close the socket.

However, for each new incoming connection, you must accept it locally. Do the following, over and over in a loop:

```
(connectedSock, clientAddress) = sock.accept()
```

You won't need to use the clientAddress - although if you want, you can print it out as debugging information. However, the `connectedSock` is critical - it is a new socket object (different than the listening one), which you can use to send and receive data.

## 1.4 Receiving Data

To receive data from the connected socket, you must receive it into a buffer. The buffer contains bytes, not text (assuming that you are running Python 3), which means that you will also need to call `decode()` to convert it to a string. (Of course, in the real world, sockets often contain binary data, not just text. But our program is pretty simple!)

To receive a buffer, do this:

```
msg = sock.recv(1024).decode()
```

The parameter to `recv()` is the maximum buffer size; you will never receive more than that at a time. For maximum efficiency, set it to a moderately-large power-of-2 size; 1024 is a fine default. (Most of our messages will be quite short, so we don't really have to worry about cropping any of these.)

**NOTE:** When the socket hits the end of the data stream, it will throw a `ConnectionAbortedError`. So you should wrap your `recv()` calls in a `try/except` block to catch that error. Remember to call `sock.close()` in the `except` block![1]

## 1.5 Changing the Data

Your server program must change the message that it was sent, somehow; you get to choose how. Maybe you want to add a prefix to it, like "Here's the data:". Or, you could simply print back the **length** of the buffer, or reverse the buffer, or whatever. Be creative: just make sure that the data you send back is not the same every single time, and that it is based on the data you received, somehow.

[1] I haven't investigated whether it is **necessary** to call close on a socket that's already hit the end of the data. I suspect that it is, but I'm not 100% sure. But why not be careful?

## 1.6 Sending Data

To send data back through the socket, we have to first convert it back to a binary form (Python 3 won't let you send strings). So we'll call `encode()` to do that.

When sending, there are several ways that Python can handle it; one of the simpler functions is `send()`. However, `send()` has the downside that sometimes it will only send part of the message (because the network is congested), so I recommend that you call `sendall()` instead; it automatically calls `send()` over and over, until all of the data is flushed.

```
sock.sendall(message.encode())
```

# 2 Client

Client sockets work exactly like the server sockets, once they're connected - and they are slightly easier to set up.

Many of the steps are exactly like the server. To create a new socket on the client side, do the following:

- Create the new socket - exactly like the server
- (Do **not** bind it to an address, though)
- Instead, call `connect()` to connect it to the server

Once it's connected to the server, you will do the following:

- Generate a series of messages, however you like
- Send each message with `sendall()` - exactly like the server
- Call `recv()` to receive each reply - exactly like the server

## 2.1 Connection to a server

To connect a client socket to a sever, instead of doing `bind()/listen()`, you perform a single call:

```
addr = (host,port)
sock.connect(addr)
```

The `host` that you provide must be a string, which gives either the hostname or IP address of the server. Since you will be running the client and server on the same machine, I recommend that you set `host` to `"localhost"` - which always means "the machine I'm on." But if you would like to use Lectura's hostname (`"lectura.cs.arizona.edu"`), that also works well.

The port number must be exactly the same integer that you used in the server program.

## 2.2 Generating the Data

You may generate the stream of input data any way that you like. You might have a loop, calling `input()` to get data from the user. You might open up a file, and iterate over it with a `for` loop. You might simply count from 1 to 100.

Do whatever you like! Just make sure that you send more than one message, and that for every message, you first `sendall()` to send it to the server, and then `recv()` to get the server's reply.

Oh, and print out some information as you are going (perhaps the replies?) so that you can confirm that the client/server connection is working.

# 3 Running the Lab

To run the lab, you will need to log into Lectura **multiple times.** If you are on a UNIX-like system, you can simply open up multiple terminals. If you are using Putty, you can also open multiple connections to the same machine.

Arrange the windows so that you can see both terminals at the same time. Then, on one terminal, start up the server program. On the second one, start up the client. Watch the output to confirm that they are actually sending data back and forth.

(You included some debug messages so that you could see the output, right?)

## 3.1 Running on Another Machine

If you have time, try running the client on another machine. To make this work, update the client code so that it connects to lectura (not to `localhost`), and then run it on some other machine, such as your own laptop. Can you connect to the server, running on Lectura?

# 4 Going Further

Congratulations, you've completed the lab! Make sure that you've pushed it up to the GitHub repo, so that we can grade it. And please remember - don't put lots of complex features into your required programs.

But now that you're done, you can start exploring. Copy the code into new files, and see what you can do - if you come up with cool programs, check them into the Git repository, we'd love to see them!

Some ideas:

- The current server can only connect to one client at a time. What if you used the `threading` library to create a new thread every time that a new client accepts; use that thread to handle that client, while the main thread goes back to `accept()` another connection.
- Change the server so that, when it reads a message, it interprets it as a filename. Open up the file specifed (if it exists), and send the entire contents back to the client.
- Using the `threading` library, write code which can both send and receive at the same time; in one thread, get input of some sort (maybe call `input()` ?) and send everything you get to the other side. In the other thread, simply `recv()` over and over, and print out what you get.

  If you do this on **both** ends of the connection, you've created a little chat server - you can type messages on one end, and they automatically get echoed out on the other!