Labs 04 & 05: Client Server Programming on Raspberry

We're going to learn about the network, about the Internet and how to interact with it through a code. So eventually we want our Raspberry Pi to talk on the network. We want our IoT device to be able to actually interact on the network. It should be able to talk to other servers, other machines on the network, and get data from those machines, request services from those machines. Thus, it can offload work to a remote machine, and get results back. The Raspberry Pi, getting data from the world, interpreting that, sending it to the network, getting some service, getting results back to the Raspberry Pi and then sending actuations out to the world.

So, first we learn a little bit about the Internet.

The structure of the internet is relatively ad hoc. At any time, one can open up a laptop and have it connected onto the network and become a node on the internet. You need to send data from one local area network to another. So how it is done? Generally, it done through a protocol, internet protocol. A protocol is basically a set of rules that define the communication.

Protocol defines basically how you're supposed to communicate. It gives you some rules. Now it doesn't define everything. It doesn't define what you can say. You can put any data inside the payload. When one machine from one local area network wants to talk to another machine from another local area network it has to adhere to the protocol. It has to encapsulate data the way that the Internet protocol dictates. And as long as it does that, the machine at the receiving end will understand that data and be able to pull out the information, and they can communicate. And these protocols don't interfere with the LAN protocol. So these local area networks, they can have whatever protocols they want.

What is the payload of the message? Packets can have whatever information in them as long as you put the special header for the Internet protocol on the outside. So the local area networks can have whatever random protocols they want to have as long as they adhere to these Internet protocols and put their headers on the front and footers on the back. Then two machines can communicate.

IP Address

IP address is a unique number for all machines on the Internet. IP version 4 uses 32-bit addresses, this is standard right now. So that's four numbers that are 8 bits each, separated by periods. That's usually how it's notated, so 192.0.0.0. Since each number's an 8-bit number, it goes up to 255, so 0 to 255. So you get four numbers in sequence like that are in the range of 0 to 255.

Now, often this IP address is chopped up into two parts conceptually. There is a network address, which is the address that is common to all of the machines that are

inside that particular local network. These are the high bytes of the address. Then the host address are the low bytes and those are unique to every host in the network.

IPv6

IPv6 is the new IP. So the problem that with IPv4 is that it is only a 32-bit address. IPv6 uses 128-bit addresses, much longer addresses. So the 32 bits is limiting, IPv6 is supposed to be a solution to that, in that IPv6 gives you the 2 to the 128 addresses which is much bigger number. IPv6 integrates security also as it uses IPSec which is a protocol which performs encryption

Port Number

Every machine has its IP address if it is on the network. But TCP and UDP protocols use port numbers to specify particular network applications on a machine. TCP and UDP use process to process communication. So each process needs to be assigned to an individual port number. Thus port numbers are assigned to each network application layer protocol. Let's take for instance, web traffic. Web traffic uses a protocol called HTTP. HTTP is assigned to port 80. So port 80 is known as a port where web traffic is going to be. Therefore, every web browser, when it is talking to a web server, it sends data to port 80. And when a server is sending responses back to the web browser, it sends it on port 80. So the web server has to listen on port 80 and it sends back data on port 80. So port 80 is known for web traffic. This happens with all network applications. They are associated with a particular port, like Secure Shell. Secure Shell is on port 22.

Now port number is a 16-bit value and it's contained in the TCP and the UDP headers. So it's only 16 bits, but that so far has been enough bits. 16 bits gives you 64,000 or 64K which is 65,000 addresses, more than enough right now.

Domain Name

Each host on the internet has a unique IP address. Now, IP addresses are needed to send messages. If you want to send a message to a host, you need its IP address, and the IP address is included in the IP header of the package that's being sent. The IP addresses, though, are not easy for humans to memorize, so we use domain names instead e.g. bbc.com. Humans can memorize that. So in order to send a message each domain name that the human knows must be translated into an IP address. So the human types in the domain name, and then that has to automatically get converted into an IP address that can be included in the packet header and the message can be sent. So, for that we use what's called domain naming system, DNS. So DNS is basically a big hierarchical naming system to determine IP addresses.

So basically imagine a gigantic table that maps domain name to IP address. Now there isn't just one table because this is worldwide network. So it's actually a hierarchy of tables in different machines across the world. The idea is that when you want to go to a web page, say cnn.com then that cnn.com has to get converted to the IP address through a DNS lookup. The web browser sends a message to a DNS server and says, look, what is the address of cnn.com? And if this Domain Name Server knows that address, it sends it back. If it doesn't, then it goes and asks another DNS server higher

up. And if it knows, it sends it back. If it doesn't, it asks another DNS server and so on. Eventually you get to the top level DNS server and that will be able to find any address. Now for performance reasons you don't want to do this DNS Lookup every time you go to a website if you can help it. So if I go to cnn.com once, rather than looking it up every time it stores it in a cache, a local memory. It stores that address in local memory, and so that the next time you go to cnn.com it doesn't have to look it up again.

So this DNS lookup usually perform by your tools, by your web browser let's say. Behind the scenes, you don't have to do it manually, but you can do it manually on a Linux machine just to see how it works. So you can use, command nslookup, so if you go to your raspberrypi or any Linux machine and you type nslookup and it gives back IP address.

nslookup www.google.com

Client Server Model

We need to know client-server model so we can start programming, connecting to the Internet through programs rather than just as users. So client-server model describes many interactions over the network on the Internet between the machines. Sometimes you're talking peer-to-peer but client-server is a very common model for communication on the Internet. In client-server model two processes are running, a client on one machine, a server on another machine. The server is guarding some resource, whatever the resource might be. So say it's a print server, then the client might want to print something, it sends a request to the server, says please print.

And the resource would be the printer itself and the server prints it or doesn't, it sends a response, e.g I printed it. Or a web server, the client sends a request to the web server, says server give me the contents of this Web page, the resource is going to be the contents of the Web pages and the server gets the resource, sends it back in response to the client, and so on. A server waits for request, sit there listening, waiting for request and then it sends the responses that have the results of whatever the request is. The client and the server communicate over a connection on the Internet and a socket is the endpoint of a connection. So if there's a connection between a client and a server, there's going to be two sockets, one for the client on its end, and one for the server on its end. So they'll be listening to sockets. And this term socket is used as we start writing network code, we're going to be creating sockets, opening sockets, closing sockets, and so on. So a socket is really a programming concept for how you're going to connect on the network. Now, we've already defined a port, again, it's a 16-bit integer that identifies a process. So if you take an IP address and a port together that is a socket. Multiple clients and servers can exist on the same machine.

There can be multiple clients, web client and secure shell client, you could have any number of clients as long as they are each associated with different points on the machine. Server processes listen to their assigned ports. So, the web server is listening to its port, the SSH server is listening to its port and they only hear traffic that comes in on their particular ports.

Creating a socket in Python

Import socket

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

AF_INET declares the Address Family to be Internet. Now there are other address families for other types of networks. But by far the most common's going to be Internet.

Also, the next argument SOCK_STREAM indicates that we're using TCP, connection-based communication as compared to UDP which is conectionless.

Sending Data : Client Side

We will be writing a generic client code.

Here, we want to connect our socket to the remote socket of the server. So, you need some host to connect. You need to know the host address.

host = socket.gethostbyname("www.google.com")

mysock.connect(host,80)

So, the first line of code is calling function socket.gethostbyname, and that performs a DNS lookup and it returns the actual IP address of that server. Now, once you got the IP address, you can connect to that server. For this purpose, we need to call the mysock.connect() function. mysock is the socket that we created in last part. So mysock.connect, that function creates a connection to the remote host. Further, the next argument is 80, the port number of HTTP meaning web traffic. Sockets are associated with the IP address and the port number.

message = " GET /HTTP/1.1\r\n\r\n"

mysock.sendall(message)

We don't have to go into the details of the http protocol, but the first type of request that is made a GET request, and all we are asking for the webpage. In www.google.com, we're asking for the contents of its webpage, so that we can display it locally on the client. So, Get, you'd get a slash as the next argument, slash is basically the path. So we're just looking for whatever is at its top level path. And then HTTP/1.1 is the version, and then you got \r\n, character return and line feeds. This message is actually an HTTP request message asking for the contents of the web page. So, we make that message string and then we send it. So we call this mysock.sendall and we pass it the message as the argument. So sendall sends all the data. It just takes whatever the message is and sends it to the server.

Receiving Data on a Socket

```
data = mysock.recv(1000)

mysock.close()
```

Client is going to wait until it receives a response. So, in this case, we requested the webpage, and we want to wait until we get the webpage. The receive function, recv does this functionality.

Notice that mysock.recv taking one argument, 1000. That argument is the maximum amount of bytes that are to receive. Now, one thousand might be too small. The reason why you have that limit is actually this is more important for a language like C. Normally, that limit is there because received data is getting copied into some buffer. Now, in Python, you don't have to specify the size of that buffer. Python automatically adjust the size of buffer according to how much data comes in.

This receive function is a blocking wait. By default, it's blocking. It means the client will sit there and not do anything until it receives the response. You can also change that making it non-blocking if required.

So that's how you receive data. Now once you've received the data and then you close the socket by calling mysock.close() and that will close the socket. Now, closing a socket is important to free up the port. You see, once you've opened up a socket on a particular port, port 80 in this case, no other processes running on your machine can use that port until that port is released.

Handling Error During Execution

So we'll talk a little bit about exception handling and how we use it with socket. Exception handling is useful, because sockets can go wrong and we want to handle such eventuality. Code is as below:

```
import socket
import sys

try:
    mysock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
   except socket.error:
    print('Failed to create socket')
    sys.exit()

try:
    host= socket.gethostbyname("www.google.com")
   except socket.gaierror:
    print('Failed to get host')
    sys.exit()

print(host)
mysock.connect((host, 80))
```

```
message = " GET /HTTP/1.1\r\n\r\n"
print (message)
message_in_bytes = bytes(message,'utf8')
mysock.sendall(message_in_bytes)

try:
    mysock.sendall(message_in_bytes)
    except socket.error:
    print ('Failed to send')
    sys.exit()

data=mysock.recv(1000)
print(data)
mysock.close()
```

Socket on the Server Side

A server, once you have a server process running on your machine, it needs to sit there and wait for requests to come in. It listens for requests, waits for requests, when it receives the request then it processes the request, gets the results, and then sends the results back. So a server's bit of a different thing than a client. The first thing a server is going to have to do is create a socket that it is listening. Then it binds the socket to an IP address and port. So when you create the socket at first, it's just floating there, not connected to anything. It needs to be associated with a particular port and IP address of the remote machine that is connecting to it. Server receives a request, then the connection is made, subsequently the client sends some data and the server receives that data. Server then processes the request and sends the response, so it sends data back along the same connection to the client.

```
mysock=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
mysock.bind(“”, 1234)        # bind () binds the socket to a port, here port number is 1234
mysock.listen(5)
conn, addr = mysock.accept()
```

The first line of code is the same as on the client to create the socket. But, the next line is different. You have to bind the socket to an IP address and a port. Now the port number being used here is 1234. It's not a well known port. It's just some generic port. But notice that if you were making a web server then use Port 80. The first argument to the bind is just empty quotes. The first argument is the client that want to connect to server. But of course, we don't know that client yet. The server's sitting there waiting for a host to connect to. It doesn't know what host it's going to connect to. So, you pass as an argument, just as nothing, as quotes. It allows it to receive request from any host machine.

As it is bound, you're going to want to listen to the socket and then accept any connections that come in. So listen starts listening for a connect. The listen means it is waiting until some client calls connect, trying to connect to the socket. Notice that listen takes an argument, a number. That number is called a backlog. That's the number of requests allowed to wait for service. We are doing single threaded code right now that means that we're only going to be able to handle one service request at a time. Therefore, five clients can be waiting in line to get served.

Now you call mysock.accept, and it returns two things. The first thing is going to be the connection to receive and send data, the second thing will be the address. The addr is the address your server is connected to. So that's the IP address and the port number. Now the port number we know because we bound to that port, but the IP address we wouldn't know until you call accept, then the addr is going to contain the IP address of the client that's connected

Full Server Program

```
import socket
import sys

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
        mysock.bind(“”, 1234)
except socket.error:
       print ('Failed to bind’)
       sys.exit()
mysock.listen(5)
while True:
      conn, addr = mysock.accept()
       data = conn.recv(1000)
       if not data:
                break
       conn.sendall(data)

conn.close()
mysock.close()
```

Modified Server Program to Control LED on Raspberry Pi

```
import socket
import sys

mysock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
        mysock.bind(“”, 1234)
except socket.error:
```

```
        print ('Failed to bind')
        sys.exit()
mysock.listen(5)
while True:
      conn, addr = mysock.accept()
        data = conn.recv(1000)
        if not data:
                break
        if data == b 'on':
           GPIO.output(13, True)
        if data == b 'off'"
           GPIO.output(13, False)
conn.close()
mysock.close()
```

Client Demo Raspberry Pi

We are going to use our Raspberry Pi to create a socket and create a client, a socket client, and send messages to another machine. Now, in this case we're actually going to send messages from the Raspberry Pi to itself, so it will send from it's IP address back to itself. So connect to your Raspberry Pi through PUTTY. Open another window through PUTTY, because we are going to send from one window and then receive in the other window. Basically we write a client program and send a message, hello world to other window. We are going to use a program called Netcat,NC for short. It can be used as a client or a server. In the receiving window (the server), we write the command

```
nc -l 1234
```

So this window is just waiting because there's nothing coming in on that port, so it's just waiting. We now start writing Python code of our client and when we actually send messages they should appear in the other window.

MyClientSocket= socket.socket(socket.AF_INET, socket.SOCK_STREAM)

AF_INET means we're using internet addressing and socket.SOCK_STREAM tells that TCP/IP will be used. As in this exercise we are talking to same machine, so the address for connect is to be given is the loopback address and that is 127.0.0.1 and also we will be giving the port as 1234 because our server is listening at the port 1234. To get the address information, we use the command

ainfo = socket.getaddrinfo("127.0.0.1", 1234)

Print(ainfo) gives the details of ainfo function. It gives three 5 tuples. We need address and port tuple and give to connect function.

print(ainfo[0][4]) gives the address we want to give as argument to Connect function.

MyClientSocket.connect(ainfo[0][4]) # Connection made to server machine

MyClientSocket.sendall(b" Hello World ") # Now sending text hello world

As the argument to sendall is not a string. It's not actually a string. It's a byte array which looks just like a string except you put b in the front. This will result in appearance of the message at the other window that is waiting for the message. We can send many messages that are displayed at the other window.

MyClientSocket.close()   #closes the socket.

Note:

- For running another instance of Putty, click on left corner of application Putty and click new session
- On shell type python3 to run, only after it is run then you can call import socket
- For running listening server at second Putty shell type
  nc -l 1234
  It means a Netcat server will run and listen at port 1234
- A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Server Demo Raspberry Pi

Here we will make a simpler server on Raspberry Pi and have a client running on the same Raspberry Pi running through Netcat.

```
import socket
mysoc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

ainfo= socket.getaddrinfo(None,1234)

# we don't know IP address of client so given None as one argument while 1234 is the
#port number where our server listens

print(ainfo)

# This command gives us many tuples and see which one we want to use. It will be
#used to bind. 127.0.0.1 is the IP address for loop back

print(ainfo[3][4])

# To check that it gives us loop back address and port number. It may  be different for
#you so confirm it
```

mysoc.bind(ainfo[3][4])

mysoc.listen(5)

conn, addr = mysoc.accept()

IP address of the client will go in addr and data in conn. This is now blocking, and will be waiting for some client to send it a message.

Now we go to other PUTTY window on the same raspberry pi, and start a client. We initiate a netcat client by command:

nc 127.0.0.1 1234

As we press enter, the blocking function at the other window will be unblocked as it has accepted the client. Now it is time to send data, just type hello world or any other message at the client and press enter.

Now go to the other window where server is running. There type

data =conn.recv(1000)

print (data)

what so ever is sent by client will be displayed with a b before it, showing that it is byte array and at the end is \n depicting character return.

So now more data can be sent. To close the communication write at the server following commands

conn.close()

mysoc.close()

Acknowledgement:

The lab is prepared using the material developed by Prof Ian G Harris of University of California at Irvine.