

Tutorial: A Sample Communications System

ECE 180D: Systems Design Laboratory

Contents

1	Introduction	1
2	An Introduction to TCP/IP and Socket Programming	2
2.1	What is Socket Programming?	2
2.2	An Introduction to TCP and IP	3
2.3	Implementing a Sample TCP/IP Protocol on RPi	3
3	MQTT Protocol for IoT	4
3.1	Publish/subscribe pattern	4
3.2	Installation	4
3.3	An introductory pair of clients	5
3.3.1	Subscriber	5
3.3.2	Publisher	6
4	Task	7

1 Introduction

This tutorial will guide you through the setup of a **communication protocol** for your RPi (that should have internet connection by now). If you don't yet have internet connection directly to the RPi, kindly follow the setup from last week to reconfigure your RPi. **Note: You will probably reconfigure your RPi many times during this course. Fear not - it gets faster on iteration :)**

The spec for last week was long, therefore this week's **expectations are fewer**, so you can finish up work from last week as well. Some repeated errors in last week's sections include:

1. **Don't forget** the 'p' in Ras'p'berry Pi
2. ONLY 1 of the 2 micro-USB ports accepts data; the camera can **only be operated** by 1 of these ports. In order to connect to the Rpi without the data micro-USB port, you **MUST** use a wireless connection.
3. You **CANNOT access** the SD card directly from your laptop if the SD card is plugged into your RPi, and you cannot **access** your RPi from your laptop if your SD card is inserted directly into your laptop.
4. What is the **difference between** `ssh pi@raspberrypi.local` and `ssh pi@IPADDRESS`?
5. Accessing your virtual python environment is **independent** of the local path you are in
6. Powering down the RPi can **change** the Dynamically Added IP for the device and prevent you from accessing it.
7. The Berryconda version to use on the RPi is a function of the **architecture of the processor** in the system.

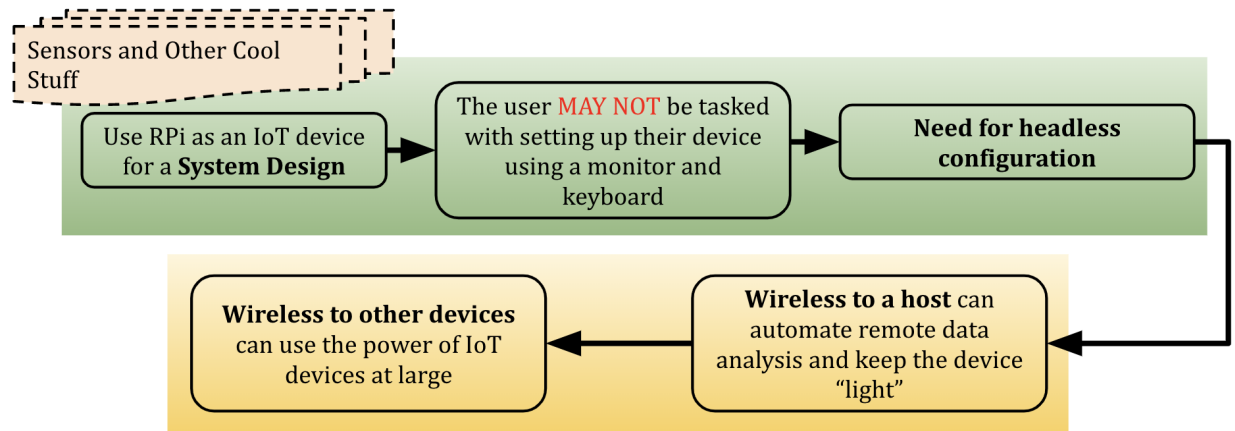


Figure 1: The yellow sections would be covered in this spec; while the green sections are covered in previous specs

2 An Introduction to TCP/IP and Socket Programming

Some important things to keep in mind before we start:

1. It is important to realize that ANY network of RPis you use to communicate information will often have restrictions imposed by the provider such that clients cannot directly talk to other clients. Please go through a server (like your laptop).
2. Python by default does not do parallel execution of code and is also synchronous by default. However, socket programming via TCP/IP need not be designed so. Libraries such as [asyncio](#) can be used to get the threading across client devices. Also you can use server code [here](#), written in C for multi-threading client service.
3. TCP/IP defines a set of rules on how computers or computer-like devices connected to the internet are expected to communicate to one another. **TCP/IP stands for Transmission Control Protocol / Internet Protocol.**
4. TCP is for communication **between applications**.
5. If one application wants to communicate with another via TCP, it sends a **communication request**. This request must be sent to an exact address. **After** a “handshake” between the two applications, TCP will set up a **“full-duplex” communication** between the **two** applications.
6. The “full-duplex” communication will **occupy** the communication line between the two computers until it is closed by one of the two applications.
7. UDP is very similar to TCP, but **simpler and less reliable**. Some details were taken from this [source](#). Also, IP is responsible for “routing” **each** packet to the correct destination.

2.1 What is Socket Programming?

Since we have TCP and UDP above, the meaning of sockets and socket programming is found [here](#): “TCP/IP is a **protocol stack** for communication, a socket is an **endpoint** in a (bidirectional) communication. A socket need not be TCP-based, but it is quite often the case. The term socket is also often used to refer to the API provided by the OS that allows you to make a connection over the TCP/IP stack. A socket is mapped uniquely to an application as the ports are managed for you by the operating system.”

2.2 An Introduction to TCP and IP

TL:DR; TCP is responsible for **breaking data** down into IP packets before they are sent, and for **assembling** the packets when they arrive. IP is responsible for **sending** the packets to the **correct destination**.

2.3 Implementing a Sample TCP/IP Protocol on RPi

Please execute the following steps on your **laptop** to set up your first TCP/IP internet connection.

```
cd YOUR_WORKING_DIRECTORY_FOR_THIS_CLASS (GITHUB LINKED)
```

Next, please open a python file: call it serverTest.py. One easy way to do this is:

```
vim serverTest.py
```

Please copy the student-programmed code or type online instructions defined [here](#) or developed [here](#), into the file and read the comments in the script to understand what each line does. **Please type each word into Python to understand and own the code you are writing; this will form a basis for your project over the 2 quarters**

```
# Reminder: This is a comment. The first line imports a default library "socket" into Python.
# You don't install this. The second line is initialization to add TCP/IP protocol to the endpoint.
import socket
serv = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Assigns a port for the server that listens to clients connecting to this port.
serv.bind(('0.0.0.0', 8080))
serv.listen(5)
while True:
    conn, addr = serv.accept()
    from_client = ''
    while True:
        data = conn.recv(4096)
        if not data: break
        from_client += data
        print(from_client)
        conn.send("I am SERVER\n")
    conn.close()
    print('client disconnected')
```

Similarly, connect your RPi to the same network (Eduroam, if you managed it, or any local hotspot that your server is connected to, and is listening at the port.)

Inside your RPi, accessed through the following command (remember, you must be able to extract the IP address of the RPi first by a cable connection, retain power and then connect via Wi-Fi)

```
ssh pi@IPADDRESS
```

Please visit your git-maintained repo similar to the server instructions, but now inside the RPi, we create the following file:

```
vim clientTest.py
```

Inside this file, please write the provided code or the code from the previous link, to fully comprehend the architecture of socket programming via TCP/IP.

```
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('0.0.0.0', 8080))
client.send("I am CLIENT\n")
from_server = client.recv(4096)
client.close()
print(from_server)
```

Note: You will likely now need 2 concurrent terminal windows open: 1 talking to the client and 1 to the server. Please follow the steps listed below to execute your program:

1. execute the server file on your laptop from the command-line or any terminal.
2. wait for a minute - now run the client side python script and observe the output on both sides.

Please show this simulation to the TA so they can verify functionality. This step is very important, so we hope that we can help all of you finish this task in particular.

3 MQTT Protocol for IoT

Here, we will introduce a publish/subscribe message passing protocol for IoT device communication known as [MQTT](#). Despite being based off of TCP/IP, it is not based off of the traditional client-server model, which will likely make it easier to use in your projects.

3.1 Publish/subscribe pattern

The publish/subscribe pattern (sometimes referred to as pub/sub) differs from the traditional client-server model in that clients do not communicate directly with an endpoint. Instead, with the introduction of a broker (closest analogy to a server), clients (the publisher and/or subscriber) communicate with other clients without ever contacting each other directly. The broker will filter incoming messages and distribute them to the correct subscribers.

As such, in the publish/subscribe pattern, a client can decide to “publish” data to a “topic.” Corresponding clients that “subscribe” to the topic will get sent the data from the broker. One of the major aspects of this model is the decoupling of the subscriber and the publisher. The benefit of this are:

- **Space Decoupling:** Publisher and Subscriber do not need to know each other. So, the publisher and subscriber do not need to exchange IP address / port (which would need code reflashes in your project). This also makes it easier to add additional devices onto the same code.
- **Time Decoupling:** Publisher and subscriber do not need to be running at the same time. The broker will register the device, but not produce any error. This is more useful in IoT applications or sensor systems, but may be useful in your projects.

Ultimately, this model is more scalable than the traditional client-server approach. Another major benefit of the decoupling is that, if you use a [public server](#), you can communicate over different WiFi networks, without needing to set up more than one firewall exception or sending each other IP addresses and such.

3.2 Installation

For the purposes of using MQTT in Python, we will be using [Eclipse’s Paho for MQTT](#). MQTT clients exist for many, many languages, so for further information, please find [other Eclipse Paho services](#). This package allows us to code our own clients, so that we may adapt to data that we receive from another client.

With that, the installation is quite simple. On either your computer or your Raspberry Pi, we can just make use of the Python package managers. As with many other packages, we simply have to install paho through pip. In your Anaconda environment, input

```
pip install paho-mqtt
```

to get paho. Try to continue with this tutorial to see if installation was successful or not.

3.3 An introductory pair of clients

Luckily for us, we can use a publically available server to play around with MQTT, and so we only have to code the clients. Clients can be publishers, subscribers, or both. They connect through the use of a topic. Topics are just strings like “ece180d/team9/imu/hand1” that will typically identify what the sent data will be used for. For more information about topic name practices, [view this article](#).

More details about all the options of each client can be viewed [at their Github page](#). This includes all the callback types, the different settings for quality of service, and other possible security concerns. However, to begin, most of the default settings should be good enough.

3.3.1 Subscriber

A typical subscriber can be written as follows:

```
import paho.mqtt.client as mqtt

# 0. define callbacks - functions that run when events happen.
# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("Connection returned result: "+str(rc))

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    client.subscribe("ece180d/test", qos=1)

# The callback of the client when it disconnects.
def on_disconnect(client, userdata, rc):
    if rc != 0:
        print('Unexpected Disconnect')
    else:
        print('Expected Disconnect')

# The default message callback.
# (you can create separate callbacks per subscribed topic)
def on_message(client, userdata, message):
    print('Received message: "' + str(message.payload) + '" on topic "' +
          message.topic + '" with QoS ' + str(message.qos))

# 1. create a client instance.
client = mqtt.Client()
# add additional client options (security, certifications, etc.)
# many default options should be good to start off.
# add callbacks to client.
client.on_connect = on_connect
client.on_disconnect = on_disconnect
client.on_message = on_message

# 2. connect to a broker using one of the connect*() functions.
client.connect_async('mqtt.eclipse.org')
# client.connect("mqtt.eclipse.org")

# 3. call one of the loop*() functions to maintain network traffic flow with the broker.
```

```

client.loop_start()
# client.loop_forever()

while True:
    pass
# use subscribe() to subscribe to a topic and receive messages.

# use publish() to publish messages to the broker.

# use disconnect() to disconnect from the broker.
client.loop_stop()
client.disconnect()

```

3.3.2 Publisher

A typical publisher for the previous subscriber can be written as follows. This specific publisher simply publishes 10 random numbers to the topic. If the subscriber is run, then they should receive 10 messages each containing a random number.

```

import paho.mqtt.client as mqtt
import numpy as np

# 0. define callbacks - functions that run when events happen.
# The callback for when the client receives a CONNACK response from the server.
def on_connect(client, userdata, flags, rc):
    print("Connection returned result: "+str(rc))

    # Subscribing in on_connect() means that if we lose the connection and
    # reconnect then subscriptions will be renewed.
    # client.subscribe("ece180d/test")

# The callback of the client when it disconnects.
def on_disconnect(client, userdata, rc):
    if rc != 0:
        print('Unexpected Disconnect')
    else:
        print('Expected Disconnect')

# The default message callback.
# (won't be used if only publishing, but can still exist)
def on_message(client, userdata, message):
    print('Received message: ' + str(message.payload) + ' on topic ' +
          message.topic + ' with QoS ' + str(message.qos))

# 1. create a client instance.
client = mqtt.Client()
# add additional client options (security, certifications, etc.)
# many default options should be good to start off.
# add callbacks to client.
client.on_connect = on_connect
client.on_disconnect = on_disconnect
client.on_message = on_message

# 2. connect to a broker using one of the connect*() functions.
client.connect_async('mqtt.eclipse.org')

```

```
# 3. call one of the loop*() functions to maintain network traffic flow with the broker.
client.loop_start()

# 4. use subscribe() to subscribe to a topic and receive messages.

# 5. use publish() to publish messages to the broker.
# payload must be a string, bytearray, int, float or None.
for i in range(10):
    client.publish('ece180d/test', float(np.random.random(1)), qos=1)

# 6. use disconnect() to disconnect from the broker.
client.loop_stop()
client.disconnect()
```

Notice that the two are very similar and differ only in the publishing and subscribing statements. One thing to always remember is that a publisher can also be a subscriber as these two are both clients in the MQTT model.

Because of the way that things are, one thing to note is that there are still limitations in what can be published. In the Python version of MQTT, the only thing that can be published are strings, bytearrays, ints, floats, or None. Of course, because you can send strings, you can always send a [JSON string](#) or a string version of a [Pandas array](#) (if your data is more like a spreadsheet), although the decoding may be less obvious there. We leave these exercises up to you, but pose them as options in case large amounts of data must be sent at once in a specific order.

4 Task

Program EITHER an MQTT subscriber or a publisher to run EITHER a. On your local machine or b. On your RPi (brownie points for this version). In a group of 4, let there be 2 subscribers and 2 publishers. In a group of 3, let there be 1 publisher instead. As usual, your progress should be added to your report. Showing us a working protocol will be awesome too! **As usual, commit and push this code to your own personal tutorial repo.**