Tutorial: A Sample Communications and Speech System

ECE 180D: Systems Design Laboratory

Contents

1 Introduction

This tutorial will give you some of the necessary tools to setup a basic communications scheme between possible devices. Normally we would have set up your Raspberry Pis to connect to the Internet already when doing this tutorial, but we have pushed that back (if you see any references to RPis, please ignore until next week). However, you get an early view as to what are the communication possibilities present for your project in preparation for how you want to program your projects.

In this tutorial we introduce to you both a method to use TCP/IP and MQTT. Given that you are all still remote, please skip to the MQTT section. You could potentially use TCP/IP when you are all on the same WiFi network, but given we want some sort of remote interactions, MQTT provides a much easier interface. We leave the TCP/IP tutorial for your benefit and potentially for future use.

2 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.

2.1 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.

2.2 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.

2.3 An introductory pair of clients

Luckily for us, we can use a publicly 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.

2.3.1 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("test.mosquitto.org")
client.connect_async('mqtt.eclipseprojects.io')
# client.connect("test.mosquitto.org", 1883, 60)
# 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: # perhaps add a stopping condition using some break or something.
  pass  # do your non-blocked other stuff here, like receive IMU data or something.
# 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()
```

2.3.2 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("test.mosquitto.org")
client.connect_async('mqtt.eclipseprojects.io')

# 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.
print('Publishing...')
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.

3 Speech

There are many places for you to try speech processing. We'll be coming around **next week** to see which audio recognition system works well and which doesn't. Keep in mind that there are definitely some major points to keep in mind when considering these speech processing software.

- Purpose. What is the basic purpose of the library? Is it text-to-speech? Is it keyword-detection?
- Latency. That is, can this software operate on a real-time basis?
- Accuracy. In the limited vocabulary you want to use, what is the accuracy of the words that are picked up? In addition, how does the software operate under the presence of noise?

Since next week is focused on your design deliberations and audio is one of them, we hope that you will have a data-backed reasoning explaining why you choose one over the other.

1. CMU Sphynx, notably their Pocketsphynx, is a toolkit that is designed for mobile applications. Pocketsphynx is pretty tricky to set up, partially due to it being written in C. Not recommended for use unless you want to try implementing on an Arduino or something low-level. They also have a Tensorflow-based Python version of text-to-speech, but it doesn't satisfy the speech-to-text/action that we want.
2. Python library of Speech Processing. Here is a large-scale library of all sorts of speech processing tools. This includes CMU Sphynx, so if you were planning on using it after reading the description, this may be a great way to go. It is included in `pip`, so it is definitely a great way to easily get well-supported speech processing tools. Many students use the Google Speech API in this library.
3. There are a couple of higher-complexity speech processing tools used by the teams in past years. Some of the options used by previous years included Google Cloud Speech-to-Text and The Porcupine library.
4. If you plan on using Unity, there is also a speech processing toolkit. It is reportedly much better than CMU Sphynx, but may require knowledge of using C# as a programming language (not that it differs significantly with the standard programming languages). In the past, it was only available for Windows, so that may also be a bottleneck. Potentially check out some of these audio tutorials.

4 Task

1. Program a working MQTT subscriber and publisher to have two-way communications. If you have something relevant to your project, do it. If not, an easy idea is to ping pong between you and a partner a counter that increments each time you receive a message (with a delay). Screenshot the terminal and don't forget to mention your partner. In your group of 4, try doing a simple 4-way communications task. Notice the increase in complexity with each additional member. Also note any possible communications lag that you may be having.

   Consider how to use MQTT for the project. Based on your experiences, what is made possible using MQTT? What seems fairly difficult using MQTT? If you were to use MQTT, what would be a reasonable communications lag time be? Would you prefer to use a different method of transmitting data?

2. Choose some speech program, set it up, and work on an easy task. For instance, choose two fairly different words (e.g. "cat" and "dog"; ideally words you can think you might use in your project!) (for tips, see the "Guess the Word" game in this tutorial. See how accurate your program is in distinguishing these words. If you can get that working, try limit testing your speech program, considering latency and error rate:

   - Try more similar sounding words (particularly bad examples may be words like "sound" and "found", letters (A, ***B***, ***C***, ***D***, ***E***, F, ***G***). Does the performance start taking a hit?

- Phrases. How long of a phrase can work? Is the length of a phrase actually a good thing for "error correction"?
- Play music in the background. Go to a coffee shop (or work in the lab). How well does it work in noise? What are ways to improve its performance in noise?

Write some short bullet points with some individual thoughts on how this applies to your project:

(a) What can you do with your given speech program in the project?

(b) How complex do you want your speech recognition to be? How complex can you reasonably expect your speech recognition to be?

(c) What level of speech accuracy do you need? In other words, how quickly do you need an accurate recognition? Does a missed recognition hurt the progress of the game?

(d) Do you need specific hardware, specific conditions, etc. to have a reasonable confidence that it works well enough?

3. As usual, push any code that you have written onto your own personal tutorial repo (180DA-Warmup from last week).

5 Appendix: 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.

5.1 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."

5.2 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**.

5.3 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.