**Project 2 - CoAP Server**

For this project I created a script to turn my Raspberry Pi 3B (further referred to as Pi) into a simple server. I decided to use Python as my language and utilized the aiocoap library for the server and client handling. I also used this library to create client scripts that send GET requests to the server, as well as the tool Copper for Chrome (Cu4Cr) to access the server resources with a graphical user interface. This made it easier to see the packet information being passed between server and client and let me learn how to use more tools and get a better understanding of everything that was happening behind the scenes. The Pi had a temperature sensor attached to be used as a resource for my server that the client could GET and also had a simple GET handler to pass the "<<Hello world>>" response required in our project prompt.

**Node Setup**

The setup of my Pi node was pretty straightforward. I simply had a waterproof DS18B20 temperature sensor connected. My sensor came with a separate module that includes the 4.7k ohm resistor between the VCC and Data lines. As such, to connect the sensor to the Pi, it is as easy as wiring the DS18B20 to the module. Then, from the module, VCC is connected to Pi's 3V power pin, GND to Pi ground, and Data to a GPIO pin. As this server will read from the sensor and send the information to the client, there is no need for any other sensor, actuator, or LEDs to be attached. See the diagrams below (pg. 5).

**Code**

As previously mentioned, I used the aiocoap library in python to code the server. I chose this library as I wanted to get more experience with python, and it also seemed very intuitive. However, when I got into it, I realized that the documentation wasn't as fleshed out as some of the libraries I have previously used. Fortunately, I was able to get things working. I created a simple server that initializes a resource tree. I then add the resources “hello” and “temp” to the resource tree. These resources, when requested by the client, call the server functions HelloResource() and TempResource() respectively. After this initialization, the host is created and awaits any requests until the server script is terminated. In order to ensure that my server had a different IP address than whatever device is used as a client, I used an .ini file that the server uses to assign itself an IP address. This way, if I wanted to change the IP address of my server, rather than having to edit the code and hard code it every time, I can simply change the desired IP address in the .ini file and not have to worry about messing up my server. To implement this, I created a read\_config() function within my server script that reads from a file called “config.ini” and checks for a section labeled “Server.” If this section exists, it then looks for the host key and assigns its value to the server IP address. If this file doesn’t exist, it will bind the IP address of the server as the default value, which is just a localhost and always ended up being the same IP address as my client device.

The HelloResource() function initializes itself by setting a variable called content equal to the byte-type string “<<Hello World>>.” And when a GET request asks for it, it simply returns that content as payload.

For the TempResource() function, I first had to include the W1ThermSensor from the w1thermsensor package included with the Raspbian OS in my script. When the server receives a GET request on this resource the get\_temp() function is called. This function begins by assigning the W1ThermSensor object to a variable called sensor. The script then uses the W1ThermSensor member function get\_temperature() to get the current temperature from the sensor, in Celsius, and saves it in a variable. I then append this temperature in a message so it is legible rather than just sending random numbers. Then, the function converts the temperature to a string with the precision of one. And finally, all of this is then cast to bytes data type and saved as self.currentTemp and is returned to the client as payload.

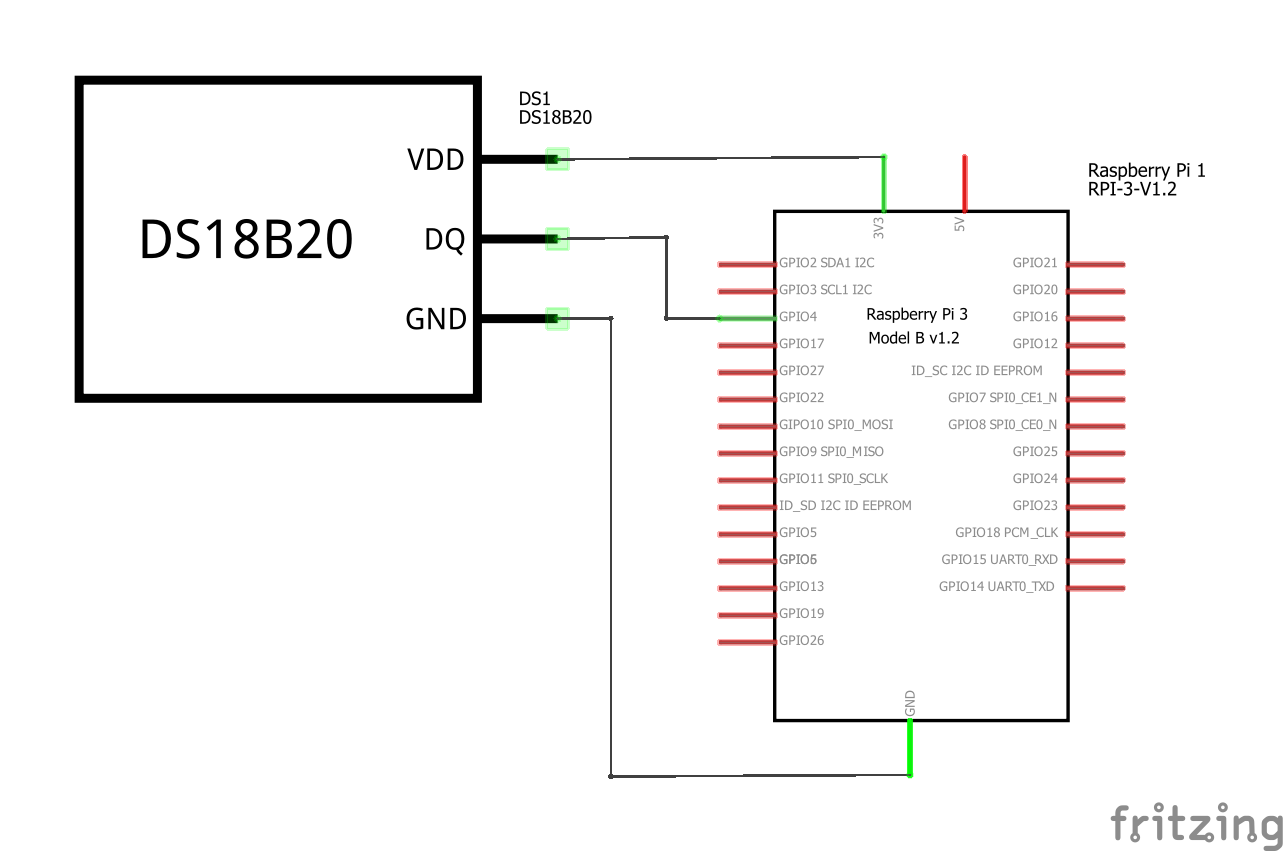
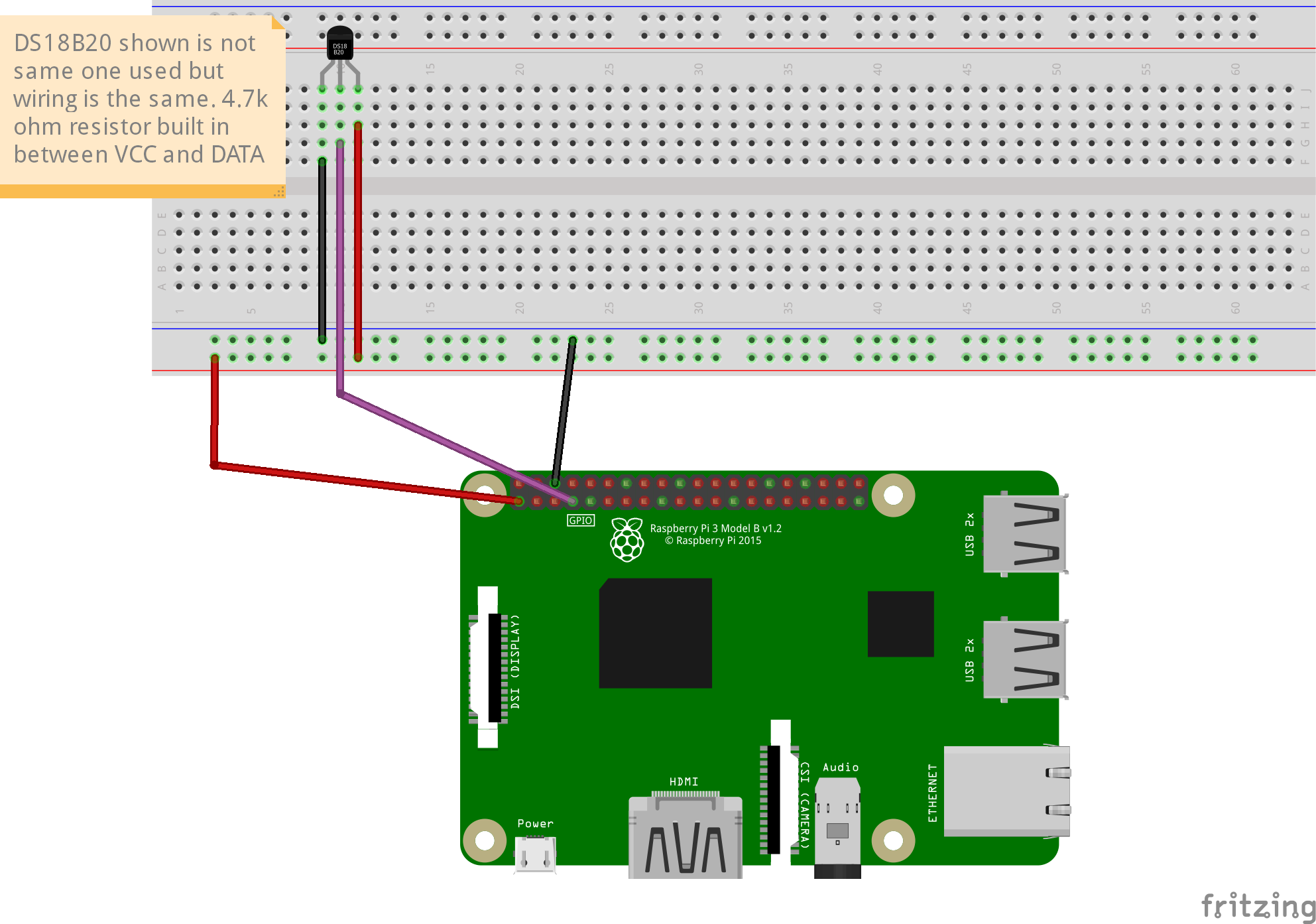
I also coded (and included in my project submission) two python GET scripts that request the aforementioned resources to be used on a client device, provided the client device also has the aiocoap library installed. However, I wanted to learn to use more tools and liked the simplicity that Cu4Cr offers, so in my video demonstration I used this in place of my own written scripts to access the server and its resources.

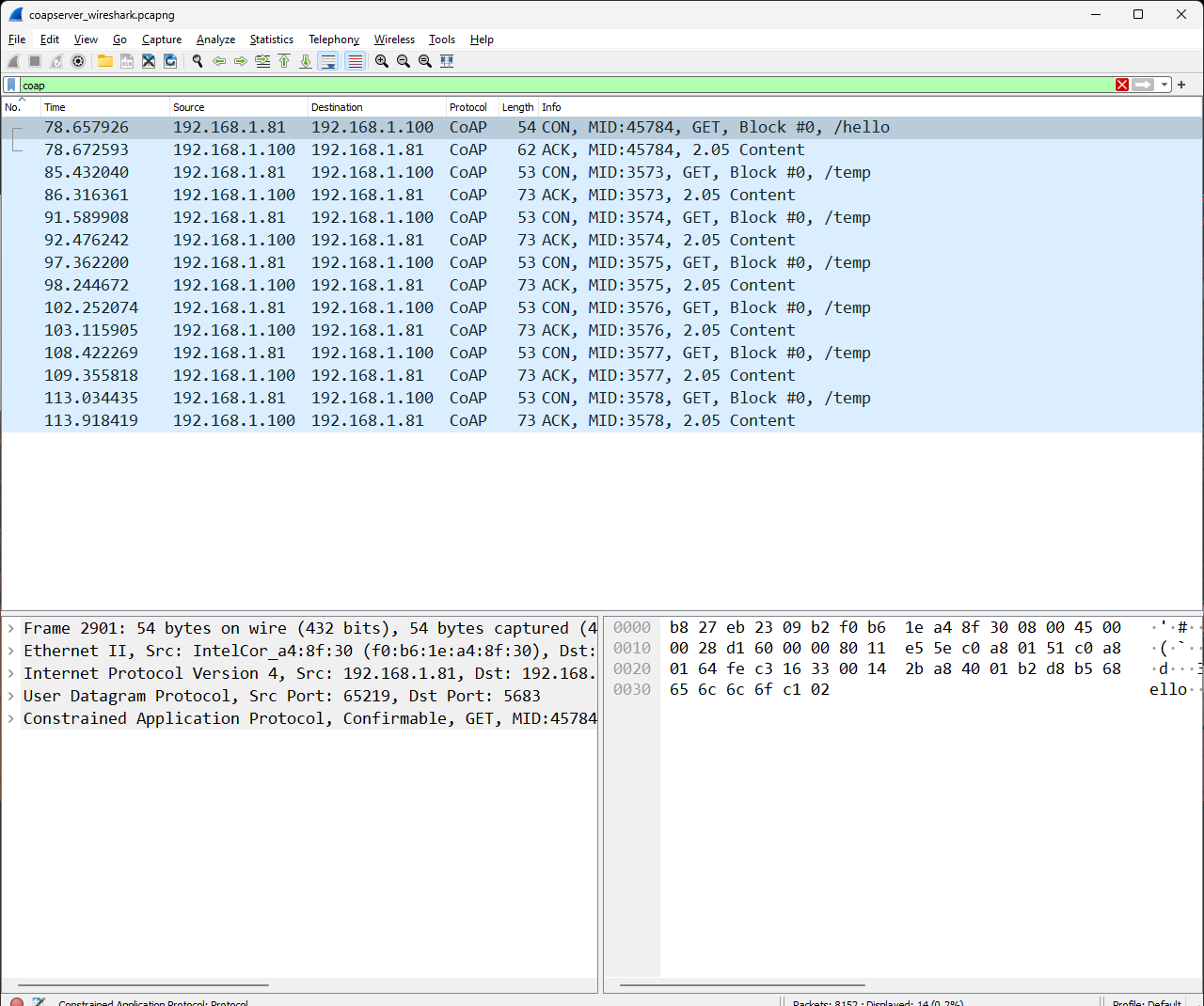
**Wireshark Packet Analysis**

I used the packet analyzing tool Wireshark to monitor the messages being sent to and from my server during the recording of my demonstration video. I have also saved the Wireshark session in a Wireshark file that I have included in my project submission. In order to better see the communication between my server and client devices you must first enter “coap” in the filter bar and you will see the packets much like the image I have attached below (see pg. 6). From here, you can see that the first packet sent was from IP address ending in 81 (client) to the IP address ending in 100 (server). This message has the CON (confirmable) type and passes the resource path “/hello.” The next packet was going in the opposite direction, 100 (server) to 81 (client) and had the ACK (acknowledgement) type with the payload <<Hello World>> and a payload length of 15, which makes sense. This demonstrates the communication between the client and server when sending a GET request of the hello resource to the server. After these two packets, there are six pairs of CON and ACK messages (CON from client to server, and ACK vice versa) that demonstrate this same communication but for the Temp resource. During my demonstration I requested this resource several times and held the temp sensor in my hand in order to get the temperature to change for each request. As a result, each of the ACK packets carries with it a different temperature reading in its payload. Each of the CON packets, however, sends the same message just requesting the resource at the path /temp. Since I chose to limit the precision of the temperature passed to the payload, each of these messages has the same payload length of 26.

**Issues Encountered**

Overall, I did not encounter many problems when completing this project. The only real issue that I ran into was that the aiocoap documentation was a lot less easy to follow than some of the other libraries I had used before. It didn’t offer much in the sense of tutorials and as such I just had to copy their version of a tutorial server and walk through each line of their code and cross reference every specific usage of their library to really understand what was happening on a line-by-line basis. For instance, I couldn’t find in their documentation a straightforward explanation of how to change the IP address of my server so that it wouldn’t keep defaulting to my localhost IP address. In order to figure out exactly how to set the IP I wanted, I needed to go through their tutorial server and find out where exactly the host object was being created. Once I found it was the line “aiocoap.Context.create\_server\_context()” I had to look up that specific function in the documentation to get an explanation of what exactly the function does and how to set my own IP. Other than that, another issue I ran into was just time. I had the project completed fairly early and as such wanted to add a bit more. After recording my demo video, I added a photoresistor to my board and added the resource to my server script. I got to learn how photoresistors work and a bit more of the electronic side to set it up correctly on my Pi. However, due to having several projects and exams between then and the due date for this project I was unable to record a new demonstration video to showcase the new resource.

****

****