CSE 157
Lab3: Interconnecting the Raspberry Pis

Isaac Garibay

May 14, 2025

Part 1 Getting Started

When approaching this project, I had to first consult the provided Socket Programming in Python guide to further understand the API that python provides for establishing communication links between devices via sockets.

The guide provided the following example code shown below of a bare bones echo server to understand the key functions responsible for establishing a communication link between devices.

```
#!/usr/bin/python

import socket

HOST = "127.0.0.1"  # Standard loopback interface address (localhost)
PORT = 65432  # Port to listen on (non-privileged ports are > 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Then this was followed by a bare bones echo client program as shown below.

```
#!/usr/bin/python
import socket

HOST = "127.0.0.1"  # The server's hostname or IP address
PORT = 65432  # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

Running these examples should yield the following output just as shown in the guide.

Client Side Output

```
(venv) isaac@raspberrypi:~ $ ./echoclient.py
Received b'Hello, world'
```

Server Side Output

```
(venv) isaac@raspberrypi:~ $ ./echoserver.py
Connected by ('127.0.0.1', 48224)
```

The guide then builds upon the echo client-server concept by introducing the ***multiconn-server.py*** and ***multiconn-client.py*** programs. These programs prove to be much more sophisticated than the previous echoserver and client programs. As the name suggests, it handles multiple client requests with a nonblocking call on the listening socket. This allows for the server to utilize the sel . select () to wait for events on multiple sockets when their is data available. More importantly, there are a few key functions that allow the server to behave this way.

The accept_wrapper() function featured below in the ***multiconn-server.py*** program takes a socket and accepts a client request on the listening socket. Then, the connection is set to a non-blocking state so that the server can move on to service potential clients on the listener socket. Without this, then the server has to return before servicing the next client which is not ideal for handling requests from multiple clients. The function then stores the data from the client into a ***namespace*** object, determines that a client connection is ready for reading/writing with an OR operation, and finally passes the mask, socket, and data objects to the selector register for the serve to select and service with the service_connection() function.

The service_connection() function takes the key returned from the select register tuple containing the socket and data objects specified as fileobj and mask respectively. When ready, the selector.EVENT_READ is set to true and the sock.recv() method is called which then appends the data to the data.outb buffer. In the case that there is no more data, then the server closes the connection and unregisters the client socket.

Finally, the server calls on the start_connections() so that it can evaluate that the client socket is ready to be written to using sock.send() which removing bytes from the data.outb send buffer and returns the amount of bytes sent.

multiconn-server.py

```
#!/usr/bin/python
import sys
import socket
import selectors
import types

sel = selectors.DefaultSelector()

host, port = sys.argv[1], int(sys.argv[2])
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print(f"Listening on {(host, port)}")
lsock.setblocking(False)
sel.register(lsock, selectors.EVENT_READ, data=None)

def accept_wrapper(sock):
    conn, addr = sock.accept()  # Should be ready to read
    print(f"Accepted connection from {addr}")
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b"", outb=b"")
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    sel.register(conn, events, data=data)

def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(1024)  # Should be ready to read
        if recv_data:
            data.outb += recv_data
        else:
            print(f"Closing connection to {data.addr}")
            sel.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print(f"Echoing {data.outb!r} to {data.addr}")
            sent = sock.send(data.outb)  # Should be ready to write
            data.outb = data.outb[sent:]
try:
    while True:
```

```
        events = sel.select(timeout=None)
        for key, mask in events:
            if key.data is None:
                accept_wrapper(key.fileobj)
            else:
                service_connection(key, mask)
except KeyboardInterrupt:
    print("Caught keyboard interrupt, exiting")
finally:
    sel.close()
```

The documentation makes note of the differences of the program shown above and ***multiconn-client.py***. The snippet below shows that the client side program initiates connections with the start_connections() function rather than listening. Moreover, the client side program takes an additional command line argument num_conns specifying the number of connections to establish to the server.

multiconn-client and mutliconn-server diff

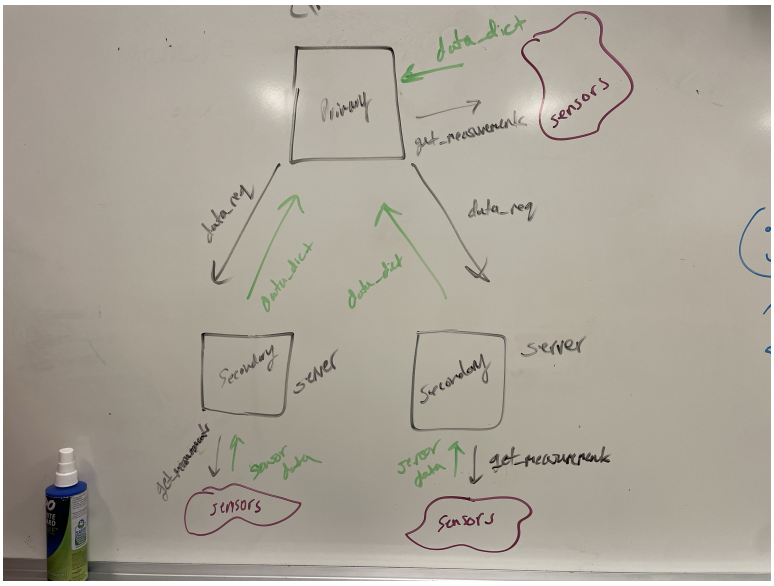
```
def service_connection(key, mask):
     sock = key.fileobj
     data = key.data
     if mask & selectors.EVENT_READ:
         recv_data = sock.recv(1024)  # Should be ready to read
         if recv_data:
-            data.outb += recv_data
+            print(f"Received {recv_data!r} from connection {data.connid}")
+            data.recv_total += len(recv_data)
-        else:
-            print(f"Closing connection {data.connid}")
+        if not recv_data or data.recv_total == data.msg_total:
+            print(f"Closing connection {data.connid}")
             sel.unregister(sock)
             sock.close()
     if mask & selectors.EVENT_WRITE:
+        if not data.outb and data.messages:
+            data.outb = data.messages.pop(0)
         if data.outb:
-            print(f"Echoing {data.outb!r} to {data.addr}")
+            print(f"Sending {data.outb!r} to connection {data.connid}")
             sent = sock.send(data.outb)  # Should be ready to write
             data.outb = data.outb[sent:]
```

Part 2: Connecting the Pi's

Initially, I independently developed the ***primary.py*** and ***secondary.py*** scripts to test basic functionality with polling data across an infrastructured network due to scheduling constraints of the team. Nonetheless, the team was involved in the testing and debugging process for the final version of the ***primary.py*** program interacting with the finished ***secondary.py*** program.

Below is the whiteboard used to visualize the implementation of primary/secondary topology. It describes the primary pi as the controller client and the two other secondary pi's as the servers fulfilling the request by retrieving the sensor readings, sending it upstream to primary, and finally primary polling it's own sensor and plotting the data.

Note: Excuse the code's french



First iteration of the primary.py script

```python
#!/usr/bin/env python3
import sensor_polling
import json
import sys
import socket
import time
import matplotlib.pyplot as plt

if len(sys.argv) != 5:
    print(f"Usage: {sys.argv[0]} <secondary_host1> <secondary_port1> <seondary_host2> <secondary_port2>")
    sys.exit(1)

host1, port1 = (sys.argv[1], int(sys.argv[2]))
host2, port2 = (sys.argv[3], int(sys.argv[4]))
doingShit = True

client1, client2 = (host1, port1), (host2, port2)
clients = [client1, client2]

def request_readings(host, port):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
            sock.settimeout(5) #5 second time interval
            sock.connect((host, port))
            sock.sendall(b"I demand readings!")
            data = sock.recv(2048)
        return json.loads(data.decode())

    except socket.timeout:
        print(f"Timeout upon request to {host} : {port} cause homie slow\n")
    except Exception as e:
        print(f"Network error polling {host}:{port}: {e!r}")


    return None

while(doingShit):
    local_readings = sensor_polling.get_local_measurements()
    print(f"Local Readings: {local_readings}")
    for host, port in clients:
```

```
        clientData = request_readings(host, port)
        print(f"From {host} : {port} => {clientData}")

    time.sleep(3)
```

The version of the ***primary.py*** script above was able to successfully make requests and had fault tolerance by handling a timeout exception when one of the secondary pi's dropped out of the network. However, it's functionality did not have the ability to plot yet. Moreover, adhoc functionality was yet to be tested.

Secondary.py

```
#!/usr/bin/env python3
import sensor_polling
import sys
import socket
import json

doingShit = True
if(len(sys.argv) != 3):
    print(f"Usage: {sys.argv[0]} <host> <port>")
    sys.exit(1)

host, port = sys.argv[1], int(sys.argv[2])
lsock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
lsock.bind((host, port))
lsock.listen()
print(f"Listening on {(host, port)}")
lsock.setblocking(True)

while doingShit:
    conn, addr = lsock.accept()

    with conn:
        print(f"Connection on {addr}")
        try:
            req = conn.recv(1024)
            if req.strip() == b"I demand readings!":
                data = sensor_polling.get_local_measurements()
                payload = json.dumps(data).encode()
                conn.sendall(payload)
                print(f"Payload sent to {addr}\n")
            else:
                print(f"Weird ass request or something for {addr}\n")
        except Exception as e:
            print(f"Network error polling {host}:{port}: {e!r}")
```

The ***secondary.py*** script above was the simpler of the two implementations. The additional logic applicable to the lab context was all that was needed to be added to the ***echo_server.py*** example that the socket programming document provided. Here, it listens for a request from the primary pi and then uses a wrapper function that returns sensor data in a dictionary from the ***sensor_polling.py*** file (*an addition to Lab 2's program implementation*). The reasoning behind doing this was to reuse working code that retrieves data from the sensors and streamline the retrieval of the data for both the primary and secondary scripts. Note that ***primary.py*** uses this imported function as well.

get_local_measurements() wrapper function

```
def get_local_measurements():
    t, h      = read_temperature_humidity()
    soil_t, m = read_soil()
    _, wind   = read_wind_speed()
    return {
        'temperature':   t,
```

```
        'humidity':      h,
        'soil_moisture': m,
        'soil_temperature': soil_t,
        'wind_speed':    wind,
    }
```

After convening, my team and I were informed that all 3 pi's could communicate in an adhoc network if they shared the same subnet and had common SSIDs. In some sense, this was establishing a mesh network given that the SSID's and subnet was shared amongst the devices except that mesh networks are really for extending the coverage of an internet service.

Once we had learned of this requirement, we made changes to our respective interface files that define our adhoc-interface (*discussed in Lab 1*) and assigned unique host ID's.

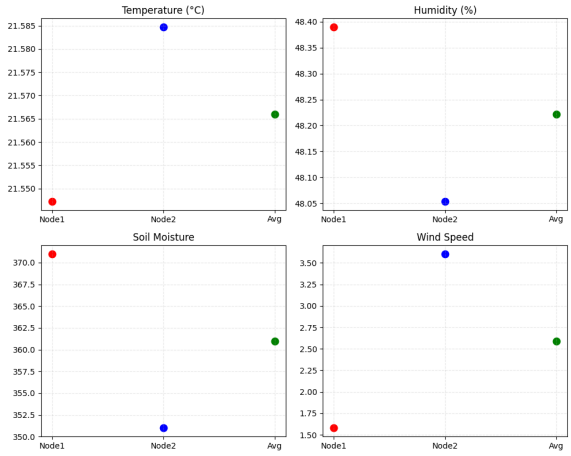
The next task was implementing the plotting functionality of the primary file. It needed to be patched such that it may plot the data aggregated among the secondaries which did not prove to be difficult to do. The timeout used in our ***primary.py*** script handled the case where one host was down. It allowed the primary to poll the other secondary pi gracefully and avoid crashing the program. One edge case that we did make a "catchall" exception for in both the primary and secondary scripts was invalid data incoming from the secondaries on the primary's side or an invalid request incoming from the primary on the secondary's side.

Below is the final version of the ***primary.py*** script allowing for the expected behavior between primary and the secondary pi's.

```
#!/usr/bin/env python3
import sensor_polling
import json
import sys
import socket
import time
import matplotlib.pyplot as plt

REQUEST = b"Requesting Data"

if len(sys.argv) != 5:
    print(f"Usage: {sys.argv[0]} <secondary_host1> <secondary_port1> <seondary_host2> <secondary_port2>")
    sys.exit(1)

host1, port1 = (sys.argv[1], int(sys.argv[2]))
host2, port2 = (sys.argv[3], int(sys.argv[4]))
doingShit = True

client1, client2 = (host1, port1), (host2, port2)
clients = [client1, client2]

def request_readings(host, port):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
            sock.settimeout(5) #5 second time interval
            sock.connect((host, port))
            sock.sendall(REQUEST)
            data = sock.recv(2048)
        return json.loads(data.decode())

    except socket.timeout:
        print(f"Timeout upon request to {host} : {port} cause homie slow\n")
    except Exception as e:
        print(f"Network error polling {host}:{port}: {e!r}")


    return None
```

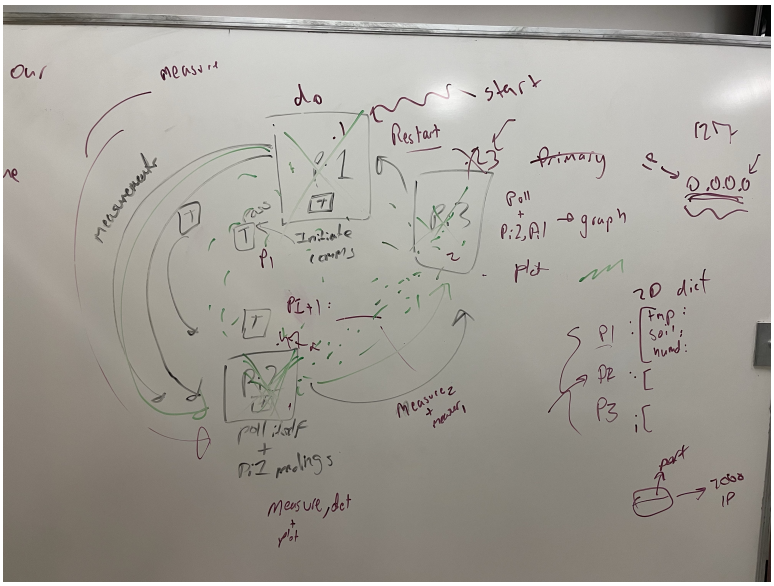
```
38
39 def plot_round(local, measurements, roundNum):
40     metrics = ['temperature', 'humidity', 'soil_moisture', 'wind_speed']
41     titles  = ['Temperature Sensor', 'Humidity Sensor',
42                'Soil Moisture Sensor', 'Wind Sensor']
43     ylabels = ['Temperature ( C )', 'Humidity (%)',
44                'Soil moisture', 'Wind speed (m/s)']
45
46     data_matrix = []
47     for metric in metrics:
48         vals = [ measurements[0].get(metric) if measurements[0] else None,
49                  measurements[1].get(metric) if measurements[1] else None,
50                  local.get(metric),
51                ]
52         #average of the readings
53         clean = [v for v in vals if v is not None]
54         avg = sum(clean) / len(clean) if clean else None
55         vals.append(avg)
56         data_matrix.append(vals)
57
58     x = [0, 1, 2, 3]
59     labels = ['Sec1', 'Sec2', 'Primary', 'Avg']
60     colors = ['red', 'blue', 'green', 'black']
61
62     #makes the subplots
63     fig, axes = plt.subplots(2, 2, figsize=(10, 8))
64     axes = axes.flatten()
65
66     for idx, ax in enumerate(axes):
67         vals = data_matrix[idx]
68         ax.set_xticks(x)
69         ax.set_xticklabels(labels)
70         ax.set_title(titles[idx])
71         ax.set_ylabel(ylabels[idx])
72         ax.grid(True, linestyle='--', alpha=0.5)
73         for xi, c in zip(x, colors):
74             y = vals[xi]
75             if y is None:
76                 ax.scatter(xi, y, marker='x', color='gray', s=100)
77             else:
78                 ax.scatter(xi, y, color=c, s=50)
79
80     fig.tight_layout()
81     fig.savefig(f"polling-plot-{roundNum}.png")
82     plt.close(fig)
83
84
85 def main():
86     roundNum = 1
87     while doingShit:
88         local_readings = sensor_polling.get_local_measurements()
89         print(f"Local Readings: {local_readings}")
90         measurements = []
91         for host, port in clients:
92             clientData = request_readings(host, port)
93             print(f"From {host} : {port} => {clientData}")
94             measurements.append(clientData)
95
96         plot_round(local_readings, measurements, roundNum)
97         print(f"Saved plot to polling-plot-{roundNum}.png")
98         roundNum += 1
99         time.sleep(3)
100
101
102 if __name__ == "__main__":
103     main()
```

Below is the resulting plot in our primary/secondary centralized topology. It shows each sensor reading for each node and a computed average for every sensor for each node.



Part 3: Token-Ring

Following soon after our fix with the adhoc functionality, we consulted the whiteboard to discuss the behavior of the token-ring topology. Our goal was to use as much of what we had before, rework it, and understand how we would handle 3 devices running the exact same script to perform different behaviors(*visually haphazard but aided our development process nonetheless*).



Our whiteboard session was followed by a pair-programming session to develop the ***token-ring.py*** script which we decided to have built upon a mash-up script consisting of the functionalities of both ***secondary.py*** and ***primary.py*** from part 2.

There was some initial trial and error with getting the first iteration up and running but eventually it panned out. The program below worked as expected of a token ring topology would. We assigned roles to specify who would initiate communicate with a data token and specified host ip's and ports for the participating pi's to listen on and forward data to. However, it did not handle fault tolerance.

```python
#!/usr/bin/env python3
import sys
import socket
import json
import time
import matplotlib.pyplot as plt
import sensor_polling

TIMEOUT       = 5
PLOT_INTERVAL = 1

USAGE = """\
Usage: token-ring.py <local_port> <next_host> <next_port> <role>
  role: start | mid | plot
  e.g. on Pi1: token-ring.py 6001 pi2.local 6002 start
       on Pi2: token-ring.py 6002 pi3.local 6003 mid
       on Pi3: token-ring.py 6003 pi1.local 6001 plot
"""

if len(sys.argv) != 5:
    print(USAGE)
    sys.exit(1)

local_port   = int(sys.argv[1])
next_host    = sys.argv[2]
next_port    = int(sys.argv[3])
role         = sys.argv[4].lower()
if role not in ("start", "mid", "plot"):
    print("Error: role must be start, mid, or plot")
    sys.exit(1)

def forward_token(token):
    try:
        with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
            s.settimeout(TIMEOUT)
            s.connect((next_host, next_port))
            payload = json.dumps(token).encode()
            s.sendall(payload)
    except Exception as e:
        print(f"[!] Failed to forward to {next_host}:{next_port}     {e!r}")

def receive_token():
    conn, addr = server.accept()
    with conn:
        raw = conn.recv(4096)
    try:
        return json.loads(raw.decode())
    except json.JSONDecodeError as e:
        print(f"[!] Bad token from {addr}: {e}")
        return []

def plot_token(token, round_num):
    metrics = ["temperature","humidity","soil_moisture","wind_speed"]
    titles  = ["Temperature ( C )","Humidity (%)","Soil Moisture","Wind Speed (m/s)"]
    labelsX = ["Pi1","Pi2","Pi3"]
    fig, axes = plt.subplots(2,2,figsize=(10,8))
    axes = axes.flatten()
    for i, ax in enumerate(axes):
        vals = [entry.get(metrics[i]) for entry in token]
        # compute avg
        clean = [v for v in vals if v is not None]
        avg = sum(clean)/len(clean) if clean else None
        vals.append(avg)
        # plot
        xs = [0,1,2,3]
        colors = ["red","blue","green","black"]
        for x,c,v in zip(xs, colors, vals):
            if v is None:
```

```
                ax.scatter(x,0,marker="x",color="gray",s=100)
            else:
                ax.scatter(x,v,color=c,s=80)
        ax.set_xticks(xs)
        ax.set_xticklabels(labelsX+["Avg"])
        ax.set_title(titles[i])
        ax.grid(True,linestyle="--",alpha=0.3)
    fig.tight_layout()
    fname = f"token-plot-{round_num}.png"
    fig.savefig(fname)
    plt.close(fig)
    print(f"[+] Saved {fname}")

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(("0.0.0.0", local_port))
server.listen(1)
print(f"[{role.upper()}] Listening on port {local_port}, forwarding to {next_host}:{
    next_port}")

round_num = 1

if role == "start":
    token = [sensor_polling.get_local_measurements()]
    print(f"[start] Sending initial token: {token}")
    forward_token(token)

try:
    while True:
        token = receive_token()
        print(f"[{role}] Received token: {token}")
        my_reading = sensor_polling.get_local_measurements()
        token.append(my_reading)

        if role == "plot":
            plot_token(token, round_num)
            round_num += 1
            time.sleep(PLOT_INTERVAL)

        forward_token(token)
        print(f"[{role}] Forwarded token ({len(token)} entries) to successor")

except KeyboardInterrupt:
    print(f"\n[{role}] Shutting down token-ring node")
```

For instance, if the pi assigned with the "start" role dropped off the network, then the other two pi's would block and continue to listen on their respective ports. That sort of behavior defeats the purpose of attempting to implement a token ring topology since this would be reflective of part 2's previous primary/secondary behavior. Moreover, if the second pi, expected to be present by the initiator, dropped off the network then the initiating pi would have indefinite timeouts because the next host in the ring is unreachable. The third pi in the ring would stand idle since it is unable to communicate with the initiating pi and is not receiving any data from the second pi.

Upon discovering this, it took an additional 8 hours to hash out the fault tolerance as we explored a method to have the script move on to the next viable host in the event a node dropped off the network leading us to one of our key design decisions. The final program below implements a flavor of the CHORD algorithm. CHORD uses a distributed hash-table that stores key-value pairs and assigns keys to nodes respectively representing it's responsibility to a set of values associated with those keys.

```
#!/usr/bin/env python3
import sys, socket, json, time
import matplotlib.pyplot as plt
import sensor_polling

TIMEOUT       = 10
```

```
PLOT_PAUSE    = 3
RETRY_PAUSE = 2

USAGE = """
Usage: token-ring.py <role> <my_host:port> <node1> <node2> <node3> [<node4>...]
  role: start | mid | plot
  each nodeX is host:port in ring order.
Examples:
  # Pi1 (starter):
  token-ring.py start 192.168.1.10:6001 \\
       192.168.1.10:6001 192.168.1.11:6002 192.168.1.12:6003

  # Pi2 (middle):
  token-ring.py mid 192.168.1.11:6002 \\
       192.168.1.10:6001 192.168.1.11:6002 192.168.1.12:6003

  # Pi3 (plotter):
  token-ring.py plot 192.168.1.12:6003 \\
       192.168.1.10:6001 192.168.1.11:6002 192.168.1.12:6003
"""

if len(sys.argv) < 5:
    print(USAGE); sys.exit(1)

role      = sys.argv[1]
my_addr   = sys.argv[2]
ring      = sys.argv[3:]
N         = len(ring)

if role not in ("start","mid","plot") or my_addr not in ring:
    print("Bad role or my_addr not in ring\n", USAGE)
    sys.exit(1)

my_index  = ring.index(my_addr)

pred_index = (my_index - 1) % N
pred_host, pred_port = ring[pred_index].split(":")

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
host, port = my_addr.split(":")
server.bind((host, int(port)))
server.listen(1)
print(f"[{role}] bound to {my_addr}, predecessor={ring[pred_index]}, ring={ring}")

def recv_token():
    server.settimeout(TIMEOUT*3)
    try:
        conn, addr = server.accept()
    except socket.timeout:
        return None
    with conn:
        raw = conn.recv(4096)
    try:
        token = json.loads(raw.decode())
    except Exception as e:
        print(f"[!] invalid token from {addr}: {e!r}")
        return []
    return token

def forward_token(token):
    for attempt in range(1, N):
        next_index = (my_index + attempt) % N
        next_host, next_port = ring[next_index].split(":")
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.settimeout(TIMEOUT)
                s.connect((next_host, int(next_port)))
                s.sendall(json.dumps(token).encode())
```

```
            print(f"[{role}] forwarded to {ring[next_index]}")
            return
        except socket.timeout:
            print(f"[!] timeout forwarding to {ring[next_index]}, trying next")
        except Exception as e:
            print(f"[!] can't forward to {ring[next_index]}: {e!r}")
    print(f"[ERROR] all successors unreachable from {my_addr}")


def plot_token(token, round_num):
    metrics = ["temperature","humidity","soil_moisture","wind_speed"]
    titles  = ["Temperature ( C )","Humidity (%)","Soil Moisture","Wind Speed"]
    labelsX = [f"Node{i+1}" for i in range(len(token))] + ["Avg"]

    fig, axes = plt.subplots(2,2,figsize=(10,8))
    axes = axes.flatten()
    for i, ax in enumerate(axes):
        vals = [entry.get(metrics[i]) for entry in token]
        clean = [v for v in vals if v is not None]
        avg = sum(clean)/len(clean) if clean else None
        vals.append(avg)

        xs = list(range(len(vals)))
        colors = ["red","blue","green","black"]
        for x, c, v in zip(xs, colors, vals):
            if v is None:
                ymin,ymax = ax.get_ylim()
                ymark = ymin + 0.05*(ymax-ymin)
                ax.scatter(x, ymark, marker="x", color="gray", s=100)
            else:
                ax.scatter(x, v, color=c, s=80)

        ax.set_xticks(xs)
        ax.set_xticklabels(labelsX)
        ax.set_title(titles[i])
        ax.grid(True, linestyle="--", alpha=0.3)

    fig.tight_layout()
    fname = f"token-plot-{round_num}.png"
    fig.savefig(fname)
    plt.close(fig)
    print(f"[+] saved {fname}")

round_num = 1
try:
    while True:
        if role == "start" and round_num == 1:
            token = [ sensor_polling.get_local_measurements() ]
            print(f"[start] initial token = {token}")
            forward_token(token)
            token = recv_token()
            continue

        tok = recv_token()

        if tok is None and role != "plot":
            print(f"[{role}] no token     re-initiating token ring")
            token = [ sensor_polling.get_local_measurements() ]
            forward_token(token)
            time.sleep(RETRY_PAUSE)
            continue

        token = tok or []

        print(f"[{role}] got token: {token}")
        token.append(sensor_polling.get_local_measurements())

        plot_token(token, round_num)
```

```
        round_num += 1
        time.sleep(PLOT_PAUSE)

        forward_token(token)

except KeyboardInterrupt:
    print(f"\n[{role}] shutting down")
```

The way that we implemented this algorithm is shown in our foward_token() function.

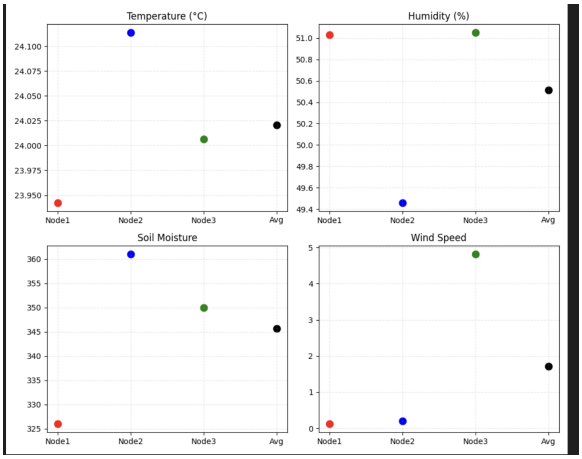
```
def forward_token(token):
    for attempt in range(1, N):
        next_index = (my_index + attempt) % N
        next_host, next_port = ring[next_index].split(":")
        try:
            with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
                s.settimeout(TIMEOUT)
                s.connect((next_host, int(next_port)))
                s.sendall(json.dumps(token).encode())
            print(f"[{role}] forwarded to {ring[next_index]}")
            return
        except socket.timeout:
            print(f"[!] timeout forwarding to {ring[next_index]}, trying next")
        except Exception as e:
            print(f"[!] can't forward to {ring[next_index]}: {e!r}")
```

In the snippet above, we have a loop that makes a number of attempts along the defined ring list holding the peers in the ring topology (*constructed from the command line arguments; exemplified in the usage message*) and returns when the forwarding is successful. To aid in detecting these cases, we used a timeout so that when a node drops off, then the loop can move on to try the next viable host. In the event that forwarding was not successful then the next viable host in the ring receives the forwarded data. The use of the modulo operator ensures that the indexing is treated in a bounded fashion to combat the risk of an index out of bounds error.

Below is the output in the initiating pi's logs in the "happy" state.

```
start] initial token = [{'temperature': 21.54726482032501, 'humidity':
    48.38941023880369, 'soil_moisture': 371, 'soil_temperature': 25.17063934752, '
    wind_speed': 1.5795000000000001}]
[start] forwarded to 169.233.25.23:6003
[start] initial token = [{'temperature': 21.54726482032501, 'humidity':
    50.11062790875105, 'soil_moisture': 369, 'soil_temperature': 25.17063934752, '
    wind_speed': 2.1464999999999996}]
[start] forwarded to 169.233.25.23:6003
[start] initial token = [{'temperature': 21.54726482032501, 'humidity':
    50.59891660944533, 'soil_moisture': 382, 'soil_temperature': 25.47419751684, '
    wind_speed': 2.1464999999999996}]
[start] forwarded to 169.233.25.23:6003
[start] initial token = [{'temperature': 21.5339131761654, 'humidity':
    51.45952544441901, 'soil_moisture': 387, 'soil_temperature': 25.069443118560002, '
    wind_speed': 1.9439999999999993}]
[start] forwarded to 169.233.25.23:6003
```

The resulting plot is as shown below. Just as was done in part2, for each nodes' sensors readings, the measurements are plotted and an average is computed per sensor per node.



One may notice that the plotting ability is shared across all pi's regardless of the role. The reasoning behind this is for both simplicity and a contributing feature to fault tolerance. In the case where one pi does drop out, there are at least two other pi's that can plot. Although, in our performance testing, we found two edge cases that were not detrimental to a viable demonstration but certainly considerations we should have taken into account. One of them being that if there was no initiator specified to begin with, the other participants would sit idle which was not our intention. We wished to have at least one pi take on the responsibility after a certain timeout. However, this does occur if there is a starting pi initially and later drops out. The second edge case presented as the initiating pi, although expressed otherwise in our script, not plotting data like the rest of the participants in the ring. Ideally, these edge cases should be covered for a truly fault tolerant ring topology. In retrospect, these features would not be difficult to incorporate in the existing script.

Discussion on the token-ring and centralized polling topologies

Regarding the two topologies we have implemented, there are some trade-offs that are observed from these two implementations.

In the centralized polling topology, the primary node actively requests readings from each secondary in sequence before sampling its own data. This on-demand model yields minimal latency for individual sensor reads and ensures that every value is as fresh as possible at the moment of collection, but at the cost of increased network traffic (two messages per sensor per cycle) and a single point of control. The primary must manage all connections and handle any slow or unresponsive node, which can delay the aggregation of the full data set or require complex timeout logic to maintain robustness. Moreover, there is a single-point of failure. If the primary node goes down, then the secondaries will remain idle and deemed useless.

The token-ring topology passes a single "token" message around the ring, allowing each node to append its latest measurement before forwarding onward. This design dramatically reduces total messages (one per cycle) and balances processing evenly across devices, but the end-to-end latency is increased because the overall average of all three nodes can only be computed until the token completes a full circuit. Moreover, the very first reading in the ring ages by (N–1) hops by the time the ring closes, introducing a bounded skew in data freshness that must be tolerated in return for lower bandwidth usage.

With these trade offs in mind, it would make sense to apply polling when sensors have heterogeneous sampling requirements or when immediate, on-demand data is critical—for example, event-driven measurements or when rapid response to a single sensor's change is required. The token-ring approach shines in bandwidth- or energy-constrained environments (such as low-power or power-line communications)

and when fairness and distributed responsibility are crucial, since no single node becomes a bottleneck and each device contributes in turn with predictable overhead.

Conclusion

Through the experience of designing the two topologies discussed above, one main takeaway is that fault tolerance is one of the more complicated features to implement but is crucial for ensuring that an IoT system remains robust. The manner in which you design your fault tolerance can greatly affect how relevant or fresh data is at the time of collection. Using a CHORD inspired method to implement fault tolerance was interesting but revealed it's limitation when there is a high rate of drop outs in the network. Perhaps this was how we defined the timing intervals but it was observed that upon drop, a significant amount of time would pass before the network stabilized. Although optimization was not the objective, I see where optimization can play a major role in easing the impacts of the negative trade-offs of a token-ring topology. Concerning the centralized approach, I find that having a single point of failure could be mitigated by having a set of primaries that could be notified when one primary is down who then takes on the responsibility of communicating with secondaries. However, this would require further sophistication of both scripts.