CSE157

Lab 3: Interconnecting the Raspberry Pis

May 16, 2025

Author	Student ID
Eshaan Parekh	1939159

1. Introduction

The goal of this lab is to work with a team to connect the Pis to form a network and send messages using socket programming in Python. You will experiment with a polling-based and token-ring-based approach using three Raspberry Pis.

2. Procedure

2.1 Getting Started

In order to get started with this lab, much reading and background knowledge is necessary to understand and work with sockets. I began this section by reading the provided libraries in the lab document on socket programming and the multiconn_ed_server.py script to develop a basic understanding of communication between the Raspberry Pis.

2.1.1 Multi-Connection Discussion

First it is important to understand what the functions accept_wrapper(), service_connection(), and start_connections() functions do in the client and server script. This knowledge is necessary for any type of communication between the Pis.

1. accept_wrapper(sock) – Used in the server script
Purpose: This function is triggered when the listening socket (the server socket) is ready to accept a new connection.

Key actions:

- Calls sock.accept() to accept a new client connection.
- Sets the new socket to non-blocking mode using conn.setblocking(False).
- Creates a SimpleNamespace object to track:
    - the client address (addr)
    - incoming buffer (inb)

    - outgoing buffer (outb)
- Registers the new connection with the selector for both EVENT_READ and EVENT_WRITE events so it can be handled in the event loop

2. service_connection(key, mask) – Used in both server and client scripts
Common purpose: Handles read and write events for a socket already registered with the selector.
Differences in usage:

Server version:
- If mask & EVENT_READ:
    - Tries to read incoming data from the client with recv().
    - If data is received, it appends it to data.outb to be echoed back.
    - If no data is received (client closed connection), it unregisters and closes the socket.
- If mask & EVENT_WRITE:
    - Sends any data stored in data.outb back to the client using send().

Client version:
- If mask & EVENT_READ:
    - Reads server's response and updates data.recv_total.
    - If all expected data is received or the server closes the connection, the client closes its socket.
- If mask & EVENT_WRITE:
    - Sends queued messages from data.messages via send(), updates the buffer data.outb.

3. start_connections(host, port, num_conns) – Used in the client script
Purpose: Initializes multiple client connections to the server.
Key actions:

- Loops through num_conns times to create multiple sockets.
- Sets each socket to non-blocking mode.
- Uses connect_ex() instead of connect() to initiate connection without blocking.
- Initializes a SimpleNamespace for each socket to track:
    - connection ID
    - total message size
    - bytes received
    - messages to send
    - outgoing buffer

- Registers each socket with the selector for both read and write events.

2.2 Polling

2.2.1 Implementation

To make the three Raspberry Pis communicate effectively, we used TCP socket connections because they are reliable and ensure messages are delivered in the correct order. One Raspberry Pi is set up as the primary device. It connects to the two secondary Pis one at a time to ask for their sensor data by sending the message "Requesting data". The secondary Pis act as servers—they listen for this message and then reply with their sensor readings.

To prevent the primary Pi from getting stuck waiting for a response that might never come, we added a timeout to each connection. This way, if one of the secondary Pis doesn't reply in time, the primary Pi moves on without crashing. After polling both secondary Pis, the primary Pi collects its own sensor data as well and calculates the average values for temperature, humidity, soil moisture, and wind speed.

Once all the data is gathered, the primary Pi creates a plot that shows the sensor readings from each Pi, plus the averages. This plot is saved as an image file named like polling-plot-1.png, polling-plot-2.png, and so on, for each round. This setup makes it easy to track sensor data over time and helps keep the system running smoothly, even if one Pi temporarily fails or disconnects.

Below, you can see the final implementation of the raspberry Pi's, connected to each sensor. Individually, this looks no different than Lab 2. The only changes made were to the software, rather than the actual sensor capabilities and circuit.

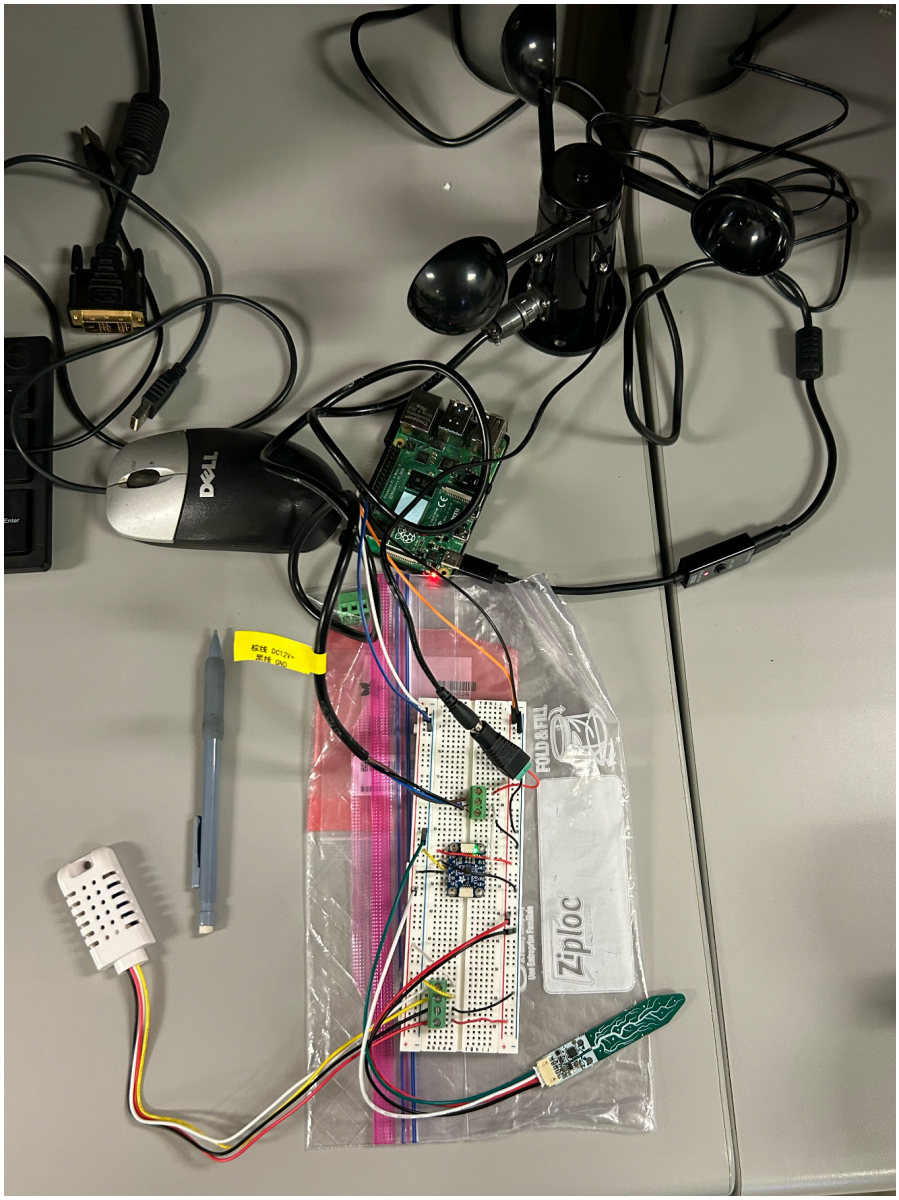

Image 1: Individual implementation of a single Raspberry Pi and its associated sensors.

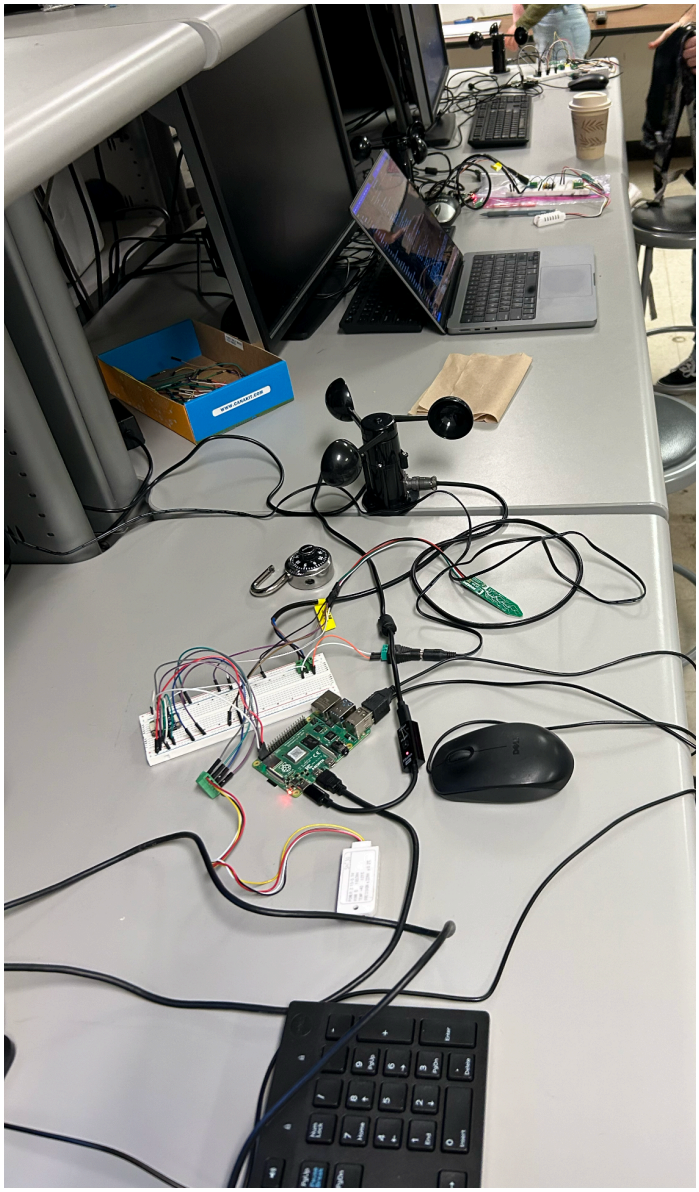

Image 2: All three Raspberry Pi’s connected to their respective sensors. There is no physical connection between the three sensor circuits.

2.2.2 Flowchart

Image 3: Flowchart of the communication between the Raspberry Pis.

Below is also the pseudocode for implementing the primary and secondary files.

primary.py:

```
Initialize TCP socket client
Define secondary Pi IPs and ports
Set polling round = 1

while True:
    own_sensor_data = read_sensors()  # temperature, humidity, soil moisture, wind

    received_data = []

    for each secondary_pi in [secondary1, secondary2]:
        try:
            open TCP connection to secondary_pi
            set socket timeout (e.g., 5 seconds)
            send "Requesting data"
            receive data from secondary Pi
            append to received_data
            close connection
        except Timeout or ConnectionError:
            log failure and append default/empty values to received_data

    compute averages of all sensor data (own + received)
    plot and label values from own + secondary Pis + averages
    save figure as polling-plot-{round}.png
```

```
    increment round
    sleep for some interval (e.g., 5 seconds)

secondary.py

Initialize TCP socket server
Bind to host and port
Listen for connections

while True:
    accept connection
    receive message
    if message == "Requesting data":
        read local sensors
        format data
        send data back
    close connection
```

2.2.3 Difficulties

In order to get all the Raspberry Pis to be connected, we first need to be able to send them into ad hoc mode. However, simply doing this as per the instruction from Lab 1 would result in none of the Pis being able to communicate with each other. To fix this, all of the Raspberry Pi's had to have a modified interfaces file for adhoc mode. Specifically, the SSID of each Pi would have to be named the exact same. In addition, the subnet of choice had to be the exact same. However, the last address bit had to be different. These assigned IPs and names were necessary to ensure that the Pis were in fact connected to the same network and discoverable to each other.

2.3 Token-Ring

2.3.1 Description

To begin the token-ring approach of communication between the Raspberry Pis, we began with the flowchart below to get started. Although messy, it helped guide our design process and overall code structure. The circuitry remains the same as above for the polling approach.

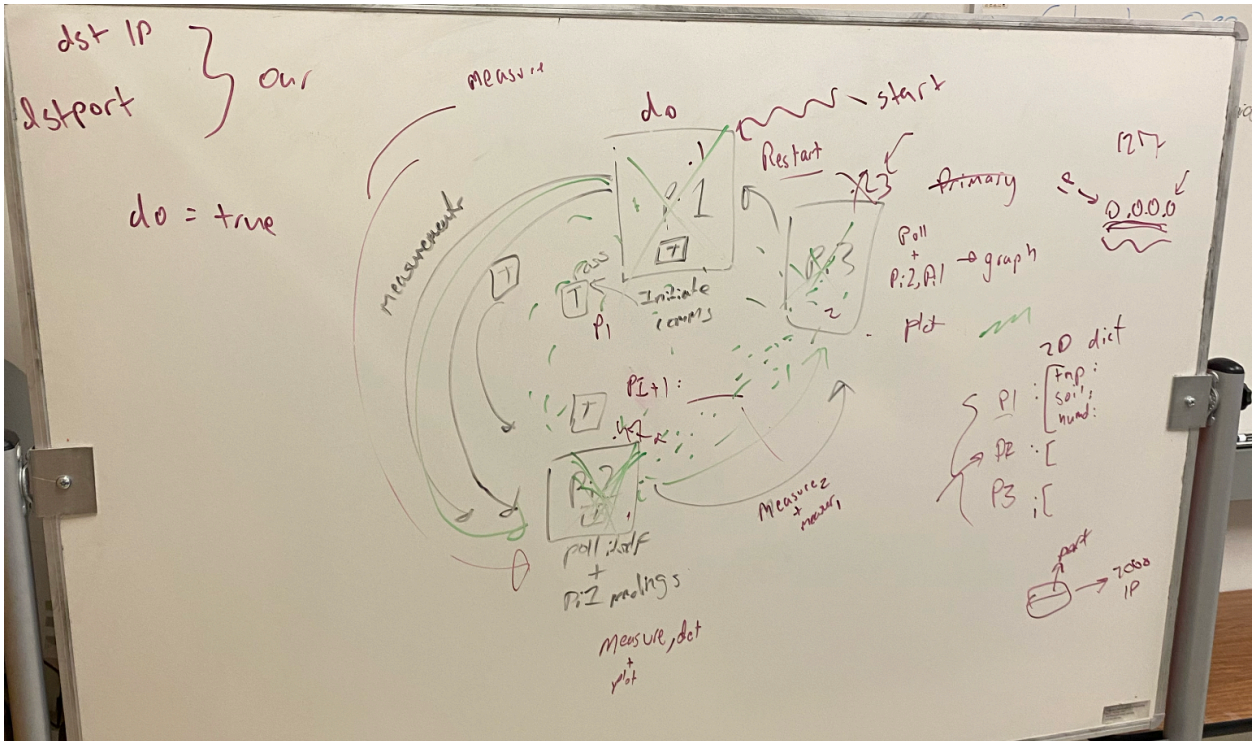

Image 4: Flowchart for the Token Ring approach.

Below, you can find the pseudocode as a high level description of the code itself.

token-ring.py

```
# Initialization
Set up UDP socket on each Raspberry Pi
Bind to its own IP and port
Define IPs and ports of next and previous Pis in the ring

# Start token passing (only one Pi starts with the token)
if is_token_holder:
    send token message to next Pi

# Main Loop
while True:
```

```
try:
    wait for incoming messages with timeout
    if received token:
        read local sensor data
        optionally perform task (e.g., print or log data)
        pass token to the next Pi
    elif received data or control message:
        handle message accordingly
except timeout:
    check if token might be lost or delayed
    optionally re-initiate token or retry communication
```

By implementing the above pseudocode, we were able to see output images such as the following:

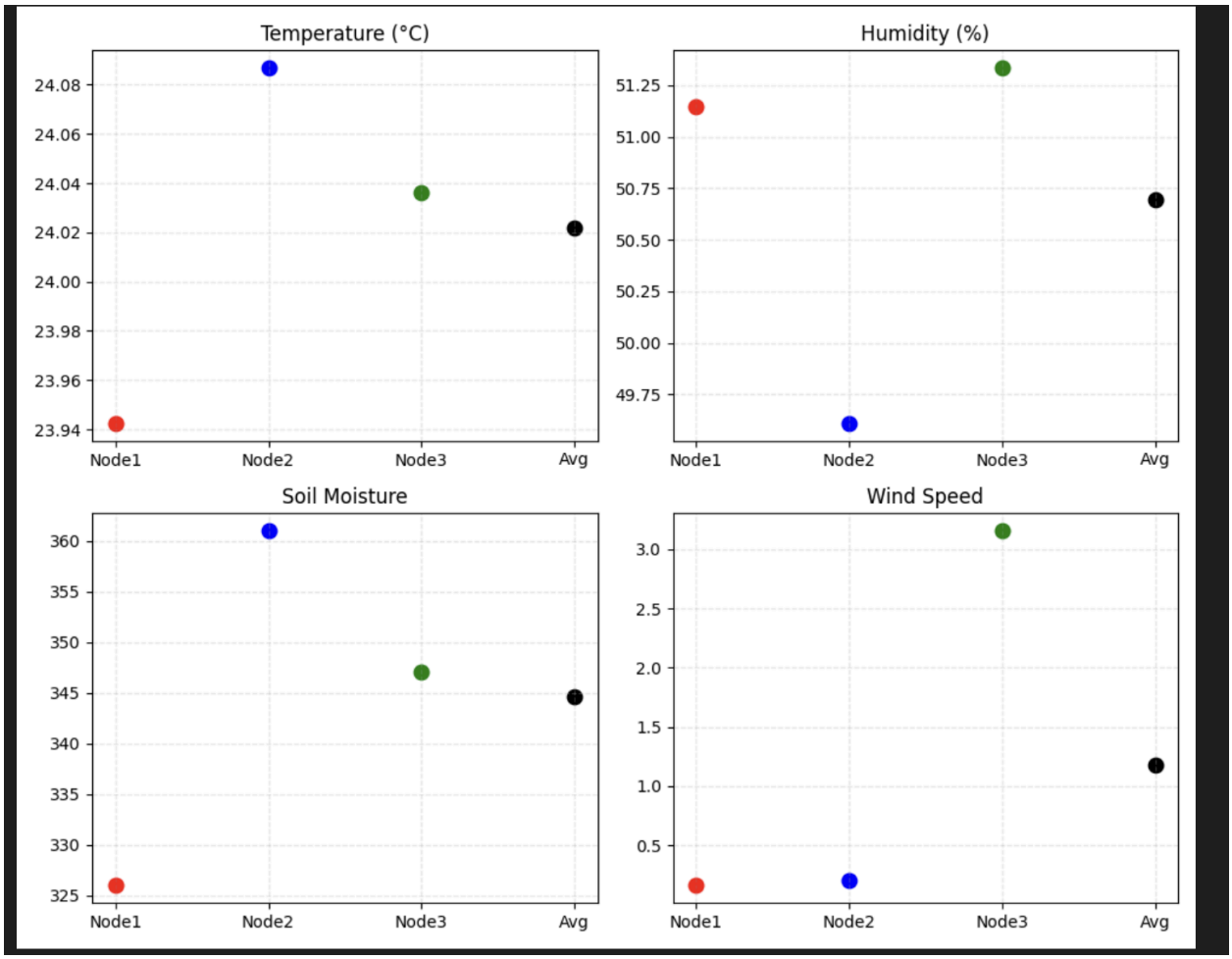

Image 5: Output flowchart of the Different nodes, with different sensors being tested.

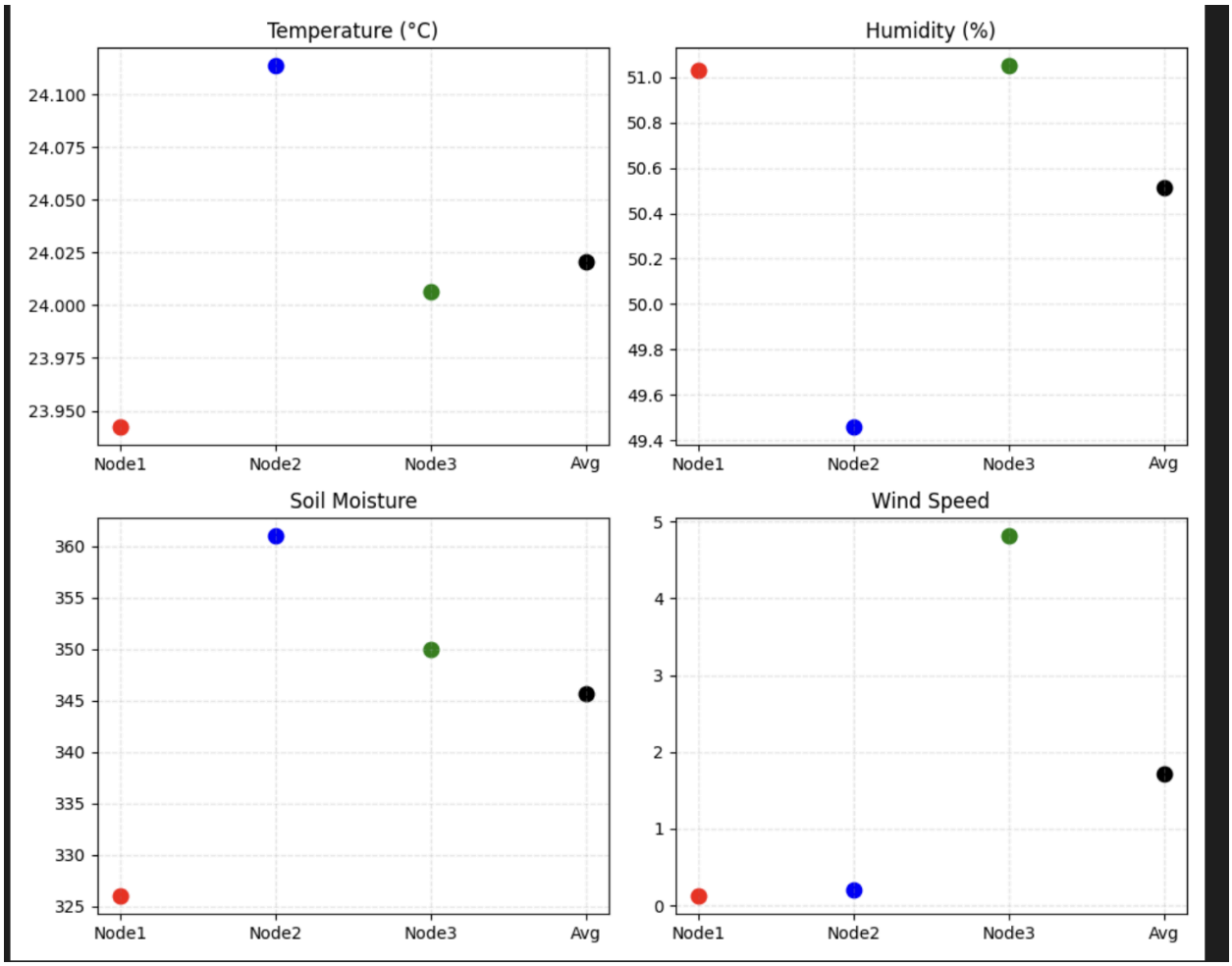

Image 6: Output flowchart of the Different nodes, at a later time, and different sensors being tested.

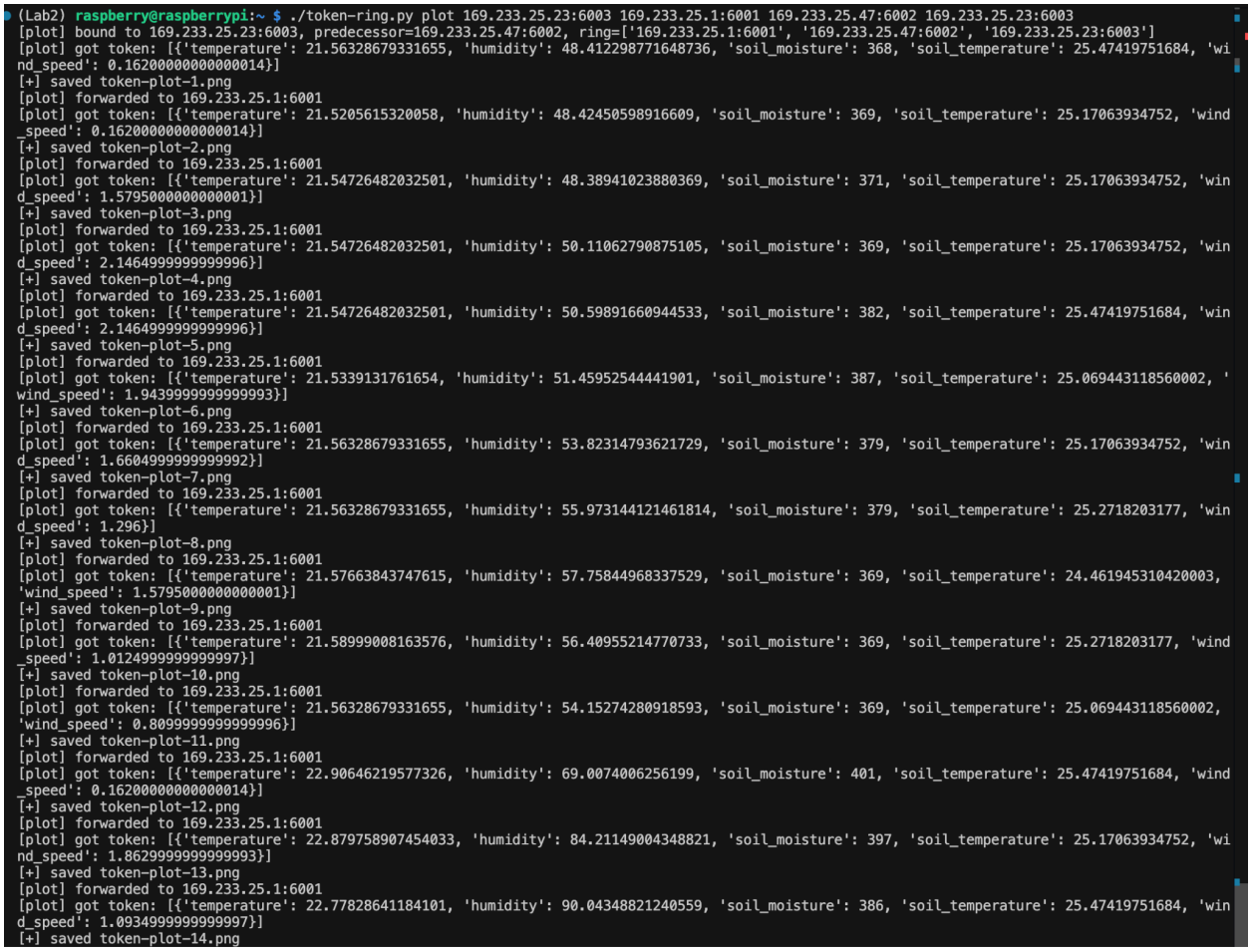

Image 7: Command line output of running the token ring file. We can see which node has the token, where we are sending it, and the different pieces of data.

2.3.3 Difficulties

Originally, the system assumed that all three Raspberry Pis in the token ring would remain online and responsive. However, this design created a critical vulnerability: if the Pi assigned the “start” role dropped from the network, the remaining two would be stuck indefinitely—one waiting to receive the token, and the other waiting to send it. This behavior mirrored the earlier primary/secondary structure and defeated the purpose of decentralizing coordination via the token ring. Even if the second Pi dropped off, the initiating Pi would stall on a timeout, and the third Pi would be idle, unable to proceed without receiving any data.

To overcome this, the implementation evolved to include fault tolerance inspired by concepts from the CHORD distributed hash table algorithm. Specifically, the code introduced a mechanism allowing the script to dynamically forward the token to the next viable host if the expected node was unreachable. This change was made in the

forward_token() function, where the Pi tries to forward the token using a bounded loop that attempts to reach other hosts in the ring. If the first target (e.g., the immediate next Pi) fails due to a timeout or network error, the loop continues with the next candidate using modular arithmetic to wrap around the ring. This ensures the system doesn't crash due to an out-of-bounds index or halt when a single node becomes unavailable.

Each attempt is wrapped in a try-except block: if a socket.timeout occurs, the script logs the error and proceeds to the next host; if another exception arises, it prints a more general error message. The settimeout() function prevents indefinite blocking by specifying a maximum time to wait for the next host to respond. As a result, even if one or more Pis drop out, the remaining nodes can maintain communication, process tokens, and continue passing messages in a ring-like fashion—thus preserving the spirit of the token ring while making the system more resilient in real-world scenarios.

2.4 Discussion

2.4.1 Timeouts

In the polling approach, the primary Raspberry Pi sends a request to each secondary Pi sequentially and waits for their sensor data. A timeout mechanism is implemented on the socket connection using settimeout() to prevent the primary from waiting indefinitely in case a secondary device is offline or unresponsive. This ensures that the system remains robust by skipping unresponsive nodes and continuing the round with available data. The timeout is particularly useful when a secondary Pi disconnects unexpectedly or crashes, allowing the primary Pi to continue polling other nodes and generate plots without being blocked.

In the token-ring approach, timeout mechanisms are implemented within the forward_token() function to ensure that if a Pi cannot successfully forward the token to its intended next node, it will automatically attempt to pass the token to the next viable host in the ring. This is especially useful when the next Pi in the ring has dropped off or is unresponsive, which would otherwise halt the token-passing process and break the communication loop. The use of a bounded loop and modulo indexing ensures that the ring remains intact and the token continues circulating even in the presence of node failures. Other potential corner cases include malformed data, race conditions (in token regeneration), and network partitions. In polling, additional issues might arise from delayed responses or partially corrupted sensor messages, while token-ring systems risk token duplication or complete token loss if not managed carefully.

2.4.2 Polling vs. Token-Ring

The polling-based approach provides low-latency data collection and high freshness, as each secondary Pi provides real-time measurements when directly queried by the primary Pi. However, this comes at the cost of centralization: the primary Pi carries most of the computational and communication load, making it a single point of failure. Resource usage is asymmetric, with secondaries remaining idle until queried. Conversely, the token-ring-based approach distributes responsibility across all nodes, improving scalability and fault tolerance. While the token ring introduces more latency—since each Pi must wait its turn to process and forward the token—the load is more evenly balanced and no single node dominates the operation.

Polling is best suited for smaller systems with limited nodes where timely and fresh data is crucial, such as centralized monitoring applications. Its simplicity also makes it ideal for systems that prioritize ease of implementation. In contrast, token ring architectures are better for larger, distributed systems where resiliency and balanced participation are more important than minimal latency. The token ring is particularly useful when no central controller is desired and the system needs to gracefully handle node failures without halting overall operation.

3. Conclusion

This lab served as a very good introduction on socket programming and creating communication between the different devices. By developing and working with sockets in both polling and token-ring, I was able to gain a deeper understanding of how sockets work and what goes on between them as information gets sent back and forth. The biggest takeaway from this lab was the intricacies of error handling. Being able to successfully get a program working is relatively easy. Making the program continue working when unintended changes happen, such as a Pi falling off the topology in the token-ring example, is far more complex and requires significant design changes and modifications. Being able to make the programs robust is especially critical with IoT devices, especially considering how error prone these devices may be. By doing this lab, I was able to gain a deeper appreciation of robust programming and an understanding of fault tolerant design processes.