Comparative Study of Performance of the Modbus Protocol on Real PLCs Using the Hardware in the Loop Methodology

Isabella Scanlan
CEIT Research Center, San Sebastian, Spain

1. Introduction

Programmable Logic Controllers (PLCs) are crucial components to the functioning of critical infrastructures [1]. As critical infrastructures were originally designed to operate independently from traditional communication networks, PLCs serve as the backbone of necessary processes to maintain a cyber secure system. Their role in controlling and automating essential processes ensures the efficiency, safety, and reliability of operations across various infrastructures, including transportation systems, power grids, and wastewater treatment plants (WWTPs). To sustain a cyber secure system, one of the first steps is to minimize the amount of time between requests in order to prevent an attack. This research aims to create a testable PLC-WWTP interface that allows the evaluation of time between requests as improvements are implemented. Overall, the interface acts as a platform for applying cybersecurity studies to reliably mitigate cyberattack risks and enhance automation system security.

2. Background

The technologies and tools used in this study are defined to better understand the set objectives.

2.1 Programmable Logic Controllers (PLCs)

Programmable Logic Controllers (PLCs) are specialized electronic devices designed for operation in industrial settings. They are tasked with managing and monitoring process operations in real time. PLCs gather data from sensors, process this information, and issue control signals to actuators to perform specific tasks. This capability allows for the control logic to be easily designed and modified without the need for physical alterations to the system, facilitating quick adjustments to new requirements. While PLCs can be programmed using various languages, the most commonly used is ladder logic, which adheres to the IEC 61131-3 standard.

2.2 CJ2M-CPU31 PLC

The Omron CJ2M-CPU31 PLC [2] is a programmable controller included in Omron's CJ2-Series that is used for this project. The CJ2M-CPU3X has a slightly longer system overhead time and allows for ethernet connection unlike its predecessor, the CJ2M-CPU1X. Its features include allowance for two Pulse I/O Modules, eight interrupt inputs or quick-response inputs, four high-speed counter inputs, pulse outputs for four axes, and more. It is ideal for basic machine automation and applications that require high speed, multiple axes, and synchronization.

2.3 MATLAB Simulink

MATLAB Simulink, developed by MathWorks, is a comprehensive platform for modeling and simulating dynamic systems. It offers a range of tools for designing, analyzing, and simulating complex systems. The platform features an intuitive graphical interface that enables users to create models by connecting various blocks visually. Simulink supports real-time simulation, allowing users to assess how systems perform under different scenarios. It is widely used in fields such as control engineering, communication and signal processing analysis, as well as the modeling of mechanical and hydraulic systems. In this study, MATLAB Simulink will be used to simulate a WWTP.

2.4 Hardware in the Loop (HIL)

Hardware-in-the-loop (HIL) simulation is a method used to validate control algorithms by creating a real-time virtual environment that mimics the physical system being controlled [3]. This technique allows engineers to test and refine their designs before actual implementation. HIL simulation facilitates the exploration of numerous scenarios and assessment of control algorithm performance without requiring physical prototypes, thus saving on the costs and time typically associated with physical testing.

2.5 Modbus Protocol

Modbus is an open standard communication protocol that provides a standardized method for devices and equipment to exchange information over serial lines [4]. The Modbus protocol facilitates Client/Server communication over Ethernet TCP/IP networks, enabling real-time data exchange between devices, such as HMI/SCADA systems. In this communication model, one device, known as the client, initiates requests, while the other devices, referred to as servers, supply the requested data or perform the requested actions.

There are four Modbus registers, shown in **Figure 2.5.1**. Coils, which control discrete outputs, are 1-bit registers that can be both read and written. Discrete inputs are 1-bit registers that only read inputs. Input registers are 16-bit registers that only read inputs. Holding registers are 16-bit registers that manage outputs, inputs, requirements for holding data, and configuration data that can both read and write. The data from the Modbus request is sent to the registers to be stored in the memory stack of the Modbus server.

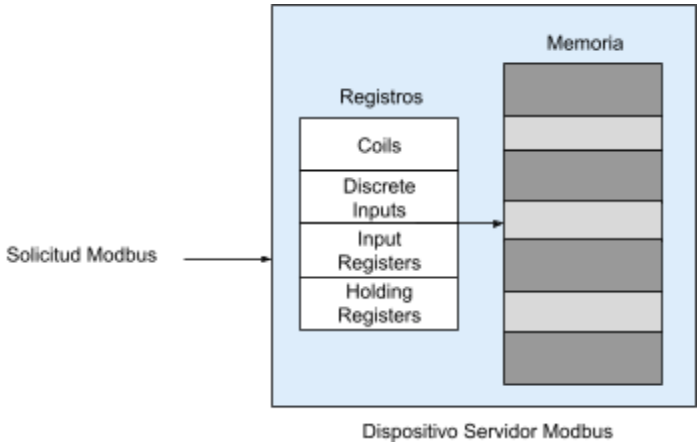

***Figure 2.5.1**: Four Modbus Registers*

The most used Modbus function codes are as follows. To read data, 01-04 are used, which cover coils, discrete inputs, multiple holding registers, and input registers, respectively. To write data, 05 is used for a single coil, 06 is used for a single holding register, 15 is used for multiple coils, and 16 is used for multiple holding registers. To write a command to a Modbus device, its address must precede the function code [5].

2.6 PyModbusTCP

The pymodbusTCP library is a Python package specifically designed to facilitate Modbus communication over TCP/IP networks [6]. As a subset of the broader pymodbus library, it provides a streamlined interface for implementing Modbus TCP clients and servers, enabling efficient communication between devices in a Modbus TCP network. Using the library gives access to a modbus/TCP server using a ModbusClient object.

2.7 Fabric

Fabric is a Python library that streamlines the process of managing and automating remote servers via SSH. It simplifies tasks like configuration and deployment by allowing one to write Python scripts that execute shell commands and remotely manage files. Fabric makes it easy to define tasks, handle remote connections, and interact with a remote environment, making it a valuable tool for system administrators and developers looking to efficiently automate their workflows.

2.8 Docker

Docker is an open-source platform designed for containerizing and managing applications. By focusing on container efficiency, Docker ensures consistent deployment and optimal resource utilization. It offers user-friendly tools that simplify the creation and administration of containers, enhancing both development and deployment processes. Docker's compatibility with microservices architectures and its seamless integration with various tools contribute to a more cohesive and effective environment, boosting the overall efficiency and portability of the system. A tool used in Docker is Docker Compose, which combines multiple

containers and allows them to run at once [7]. Written in a single YML file, this tool simplifies overall control of the application.

2.9 Latency

A performance metric analyzed in this study is latency, which refers to the amount of time it takes for a packet of data to travel from one point to another. Latency is typically measured in milliseconds, where the amount of time accounts for the complete cycle of a packet. Assessing round-trip latency is crucial for evaluating network performance, particularly in TCP/IP networks where a server sends a set amount of data.

Identifying constant delays or unexpected spikes in latency can indicate a performance problem, which can stem from various issues such as cybersecurity [8]. Furthermore, elevated latency can impair overall performance by extending the time required for data transmission.

2.10 Scalability

Scalability can be categorized into two types: programmatic and vertical. Programmatic scalability involves increasing the number of threads to analyze the impact on system parameters, while vertical scalability, the focus of this project, adds more clients to assess how system performance degrades under increasing load until the PLC reaches its maximum capacity [9].

In performance testing, the objective is to progressively increase the system load and evaluate how well it maintains its performance. This project focuses on vertical scalability by incrementally adding pymodbusTCP clients to gauge the PLC's capacity, effectively simulating the maximum number of SCADA systems it can handle.

3. Design

To evaluate system performance using vertical scalability, an interface utilizing the pymodbusTCP library has been designed. The interface is capable of connecting to the PLC as a client and saving its data in a virtual server created inside the interface. It then sends the data of the virtual server to the WWTP in MATLAB Simulink, where values of oxygen, nitrates, ammonium, and temperature are outputted according to the input values of aeration, internal recirculation flow rate, and water sludge flow rate. The output values are then stored in the virtual server, where they are written to the PLC. This process runs in a continuous loop (**Figure 3.1**), where the outputs are constantly being updated in order to monitor the WWTP system.

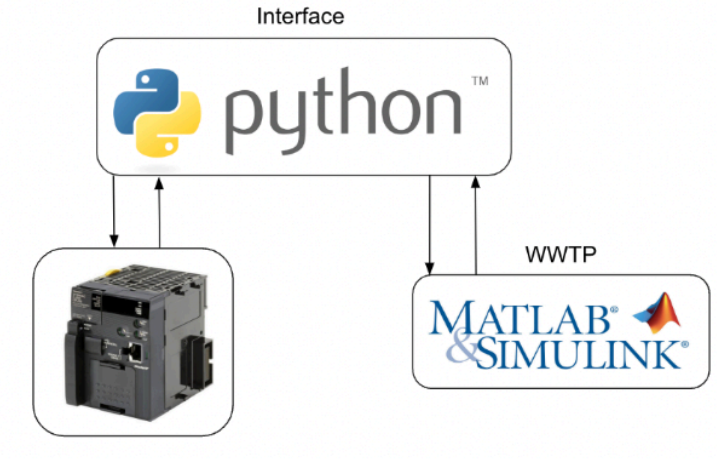

***Figure 3.1**: Design of the PLC-WWTP Interface*

4. Implementation

4.1 Code Structure

4.1.1 Server.py

Server.py is a file that creates a Modbus server and initializes its holding registers to zeros, displayed in **Figure 4.1.1.1**.

```
server = ModbusServer("0.0.0.0", port=5020, no_block=True)
server.start()
server.data_bank.set_holding_registers(0,[0,0,0,0,0,0,0])
try:
  while True:

    hrv = server.data_bank.get_holding_registers(0,8)
    print("Receiving Sensors and Actuator Data", hrv)
    print(f" Oxygen:  {hrv[0]}\n Nitrates:  {hrv[1]}\n Ammonium:  {hrv[2]}\n Temperature:  {hrv[3]}")
    print(f" Aeration:  {hrv[5]}\n Internal Recirculation:  {hrv[6]}\n Waste Sludge Flow:  {hrv[7]}\n")
    time.sleep(.1)
```

***Figure 4.1.1.1**: Server.py structure*

Following, it continuously receives data being sent from the WWTP and prints updated values to the terminal to be evaluated by the user, shown in **Figure 4.1.1.2**. This file runs concurrently with Plcsend.py, described in **Section 4.1.2**.

```
Receiving Sensors and Actuator Data [4, 12, 0, 15, 0, 25000, 106, 100]
 Oxygen:  4
 Nitrates:  12
 Ammonium:  0
 Temperature:  15
 Aeration:  25000
 Internal Recirculation:  106
 Waste Sludge Flow:  100
```

***Figure 4.1.1.2**: Server.py Printing Sensor & Actuator Data to the Terminal*

4.1.2 Plcsend.py

Plcsend.py is a file that creates two Modbus clients that connect to the PLC and virtual server, shown in **Figure 4.1.2.1**. After reading data from the virtual server and PLC, it writes the actuator data to the virtual server and the sensor data to the PLC. When run in a separate terminal from server.py, its output looks similar to the ones displayed in Figure 4.1.2.2.

```
# client connects to virtual server
vs = ModbusClient(host='10.63.28.53', port=5020, auto_open=True, debug=False)

# client connects to  real PLC
plc = ModbusClient(host='10.63.28.65', port=502, auto_open=True, debug=False)
```

***Figure 4.1.2.1**: Modbus Clients*

```
Write Actuator values to virtual server: [25000, 106, 100]
Write Sensor Data to PLC: [4, 12, 0, 15]
Write Actuator values to virtual server: [25000, 106, 100]
Write Sensor Data to PLC: [4, 12, 0, 15]
Write Actuator values to virtual server: [25000, 106, 100]
Write Sensor Data to PLC: [4, 12, 0, 15]
```

***Figure 4.1.2.2**: Plcsend.py Terminal Outputs*

4.1.3 Fabfile.py

To simplify running the two files, a third file was created using Fabric. By running this file, it automatically runs server.py and plcsend.py concurrently instead of needing to start the two files separately. Fabfile.py works by defining a task that can run server.py and plcsend.py simultaneously, shown in Figure **4.1.3.1**. This simplifies the process when integrating with Docker, where only fabfile.py will need to be called in order to deploy the interface. **Figure 4.1.3.2** demonstrates the terminal output after running fabfile.py.

```
@task
def run_concurrent(c):
    server_thread = threading.Thread(target=run_server)
    client_thread = threading.Thread(target=run_client)

    server_thread.start()
    sleep(2)
    client_thread.start()

    server_thread.join()
    client_thread.join()
```

***Figure 4.1.3.1**: Task that Runs a Server (server.py) and Client (plcsend.py) Concurrently*

```
Write Sensor Data to PLC: [4, 12, 0, 15]
Receiving Sensors and Actuator Data [4, 12, 0, 15, 0, 25000, 106, 100]
 Oxygen:  4
 Nitrates:  12
 Ammonium:  0
 Temperature:  15
 Aeration:  25000
 Internal Recirculation:  106
 Waste Sludge Flow:  100

Write Actuator Data to virtual server: [25000, 106, 100]
Receiving Sensors and Actuator Data [4, 12, 0, 15, 0, 25000, 106, 100]
 Oxygen:  4
 Nitrates:  12
 Ammonium:  0
 Temperature:  15
 Aeration:  25000
 Internal Recirculation:  106
 Waste Sludge Flow:  100
```

***Figure 4.1.3.2**: Fabfile.py Running in Terminal*

4.2 Container Integration

To house the interface in Docker, a Docker Compose file was written in order to combine the interface, the MATLAB Simulink WWTP, and the four clients used to test the system performance. These components are separated in their own containers, for a total of six containers. A green play symbolizes the container is running, shown in **Figure 4.2.1**.

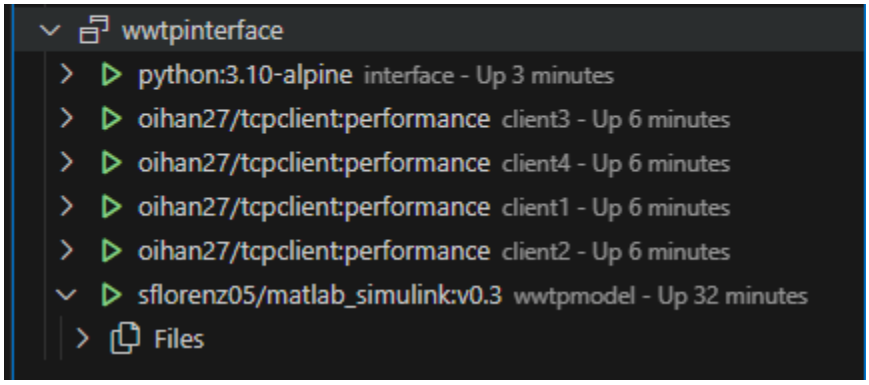

***Figure 4.2.1**: Docker Containers Running*

The Docker Compose file, written as a .yml file, includes defining each of these containers with a command to run it, ports each container is utilizing, the network they operate on, and more. Specifically, the python:3.10-alpine image contains a command that will run fabfile.py, where the interface will be deployed, further discussed in **Section 4.3**.

4.3 Interface Deployment

Once all containers are running, one can view the outputs of running fabfile.py in the terminal, shown in **Figure 4.1.3.2**. These values update as the values in the WWTP update, where they are stored in the virtual server and then written to the PLC. The interface runs constantly unless stopped by the user.

5. Results

The four client containers discussed in **Section 4.2** are used to analyze the system performance and scalability. Specifically, they measure the amount of time it takes for a single client to complete 1000 cycles with the interface –updated every 1000 cycles–, the minimum and maximum time taken every 1000 cycles, and the average of all 1000 cycles, calculated by using timers in a Python script. The results for each client are displayed in **Table 5.1**, measured in seconds. These values represent a single 1000 cycle.

Client	Time Taken to Complete 1000 Cycles by One Client	Minimum Time to Complete 1000 Cycles by One Client	Maximum Time to Complete 1000 Cycles by One Client	Average
1	0.00242	0.0	0.00978	0.00236
2	0.00190	0.0	0.00956	0.00237
3	0.00212	0.0	0.00980	0.00234
4	0.00194	0.0	0.01044	0.00220

***Table 5.1**: System Performance Data for Four Clients*

6. Conclusion & Future Discussion

This study has demonstrated the performance of the Modbus protocol on real PLCs through the Hardware-in-the-Loop (HIL) methodology. By creating a PLC-WWTP interface and utilizing the pymodbusTCP library, the interactions between the two components were effectively simulated, evaluating the impact of vertical scalability on system performance. The results indicate that the system can handle multiple concurrent Modbus clients, with the performance metrics showing consistent latency under varying loads. If there were a sudden spike in latency, this signifies the system is more prone to cyberattacks.

The performance data, as detailed in **Table 5.1**, revealed that the time taken to complete 1000 cycles by individual clients remained relatively stable, with minimal fluctuations in both minimum and maximum times. This suggests that the system's scalability is robust and the PLC-WWTP interface can manage the demands of multiple Modbus clients effectively. The results align with the expected behavior of the system, demonstrating that the interface is capable of maintaining performance even as the load increases.

While the current study provides valuable insights into the performance of Modbus communication and the scalability of PLC systems, there are several areas for future research and development, including extended scalability testing and more performance metrics to analyze. Extended scalability testing would consist of deploying an increased number of clients to test the current system's performance under higher loads. Furthermore, the loads can be increased until the maximum number of clients is discovered before the system crashes. This would help further understand the PLC's capability and limits of the system, along with potentially identifying any failure points.

Additional network performance metrics to be studied include jitter, throughput, bandwidth, network availability, packet loss, and more [10]. Measuring these values is an effective way to determine the efficiency of the system and successful cybersecurity.

By addressing these areas, future research can build upon the foundation laid by this study, enhancing the performance, scalability, and security of Modbus communication in critical infrastructure systems.

7. References

[1] CJ2 FAMILY. (2024). Omron. https://assets.omron.eu/downloads/latest/datasheet/en/p059_cj2-series_programmable_controller_datasheet_en.pdf?v=9

[2] Conceptos básicos de la simulación de hardware-in-the-loop - MATLAB & Simulink - MathWorks España. (2024). Mathworks. https://es.mathworks.com/help/simscape/ug/what-is-hardware-in-the-loop-simulation.html

[3] Docker. (2020, February 10). *Overview of Docker Compose*. Docker Documentation. https://docs.docker.com/compose/

[4] Dosal, E. (2019, November 12). *What is Packet Loss in Network Security?* Compuquip. https://www.compuquip.com/blog/what-is-packet-loss

[5] Gardiyawasam Pussewalage, H., Ranaweera, P., & Oleshchuk, V. (2013, December). *PLC security and critical infrastructure protection*. Research Gate. https://www.researchgate.net/publication/271463503_PLC_security_and_critical_infrastructure_protection

[6] Lefebvre, L. (2023, November 21). *pyModbusTCP: A simple Modbus/TCP library for Python*. PyPI. https://pypi.org/project/pyModbusTCP/

[7] Modbus Protocol. (2024). Modbus Tools; Witte Software. https://www.modbustools.com/modbus.html

[8] Modbus Protocol Reference. (2019). Control Solutions Minnesota. https://csimn.com/MHelp-VP3-TM/vp3-tm-appendix-C.html#:~:text=Modbus%20Register%20Types&text=Discrete%20Inputs%20are%201%2Dbit

[9] nOps. (2024, July 22). *Horizontal vs. Vertical Scaling: HPA, VPA & Beyond*. NOps. https://www.nops.io/blog/horizontal-vs-vertical-scaling/#:~:text=Vertical%20scaling%20involves%20adding%20additional

[10] What is Latency? | How to Fix Latency. (n.d.). SentinelOne. https://www.sentinelone.com/cybersecurity-101/what-is-latency/#:~:text=Latency%20can%20have%20a%20significant