UCLouvain

epl

École polytechnique de Louvain

A Sandbox for Industrial Control Systems (security)

Author: **Yi ZHU**
Supervisor: **Ramin SADRE**
Readers: **Gorby KABASELE NDONDA, Sébastien GIOS**
Academic year 2023–2024
Master [120] in Cybersecurity

Title: A Sandbox for Industrial Control Systems (security)

Yi Zhu: Yi Zhu
Master in Cybersecurity – Corporate Strategies
Academic year: 2023 – 2024

Abstract

Industrial Control Systems (ICS) assume the central role of managing, monitoring, and orchestrating industrial operations. Given their indispensable role, the vulnerability of ICS to a plethora of malicious attacks is pronounced, necessitating robust security measures, which, regrettably, are not consistently upheld due to the long time services and cost to replace such systems. Research endeavours within this domain encounter notable challenges, primarily from the high cost of requisite equipment and the reluctance to share findings, owing to the potential disclosure of sensitive system information. Within the scope of this Master's Thesis, a novel approach is presented: the development of a lightweight ICS simulator designed to furnish an avenue for open research, facilitated by fully virtualized test environments. The solution uses Docker to emulate the network and contain the different parts of the ICS, including a physical simulation, multiple PLCs, an HMI and a Historian. The physical simulation is easily configurable and transferable across diverse scenarios through a straightforward YAML configuration file. Additionally, the simulation framework has the capability to generate basic ladder logic compatible with OpenPLC programmable logic controllers (PLCs). PlCs are implemented with OpenPLC, facilitating communication with the physical simulation via the Modbus/TCP protocol. Moreover, ScadaBR (and ScadaLTS) is deployed as the supervisory control and data acquisition (SCADA) system. Netem, integrated within Docker containers, enables the emulation of potential packet delays and/or losses, enhancing the fidelity of the simulated environment.

Keywords: SANDBOX ICS MODBUS PLC SECURITY SOFTWARE DOCKER

Acknowledgements

I would like to express my gratitude to all the people who participated, whether closely or remotely, in the completion of this master's thesis.

First of all, I would like to thank Pr. Jerome Dossogne provided this awesome template to write this thesis work.

A big thank you to Mr. Kabasele, my supervising researcher, for his supervision throughout the implementation of the testbed which contributed to the success of this work.

We would also like to thank Mr. Kabasele and Mr. Gios for taking the time to read this Master Thesis

Next, I wish to express my most sincere thanks to Pr. Ramin Sadre, my supervisor, for his availability, trust, and support throughout this year.

I would like to express my gratitude to Guillaume Hooreman, my internship mentor at HeadMind Partners, for allowing me to work on this thesis in parallel with the intern subjects.

Finally, a big thank you to my family and friends for their unwavering support throughout my studies and during the writing of this work.

Thank you all for your contribution to the success of this project.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ADU	Application Data Unit
CIP	Common Industrial Protocol
DNP3	Distributed Network Protocol 3
EIP	Ethernet Industrial Protocol
GUI	Graphical User Interface
HIL	Hardware In the Loop
HMI	Human Machine Interface
ICS	Industrial Control System
IDS	Intrusion Detection Systems
IED	Intelligent Electronic Device
IP	Internet Protocol
IT	Information Technology
KVM	Kernel-based Virtual Machine
LD	Ladder Diagram
LTS	Long Time Support
MITM	Man-In-The-Middle
MTU	Master Terminal Unit
OS	Operating System
OSI	Open Systems Interconnection
OT	Operational Technology
PDU	Protocol Data Unit
PLC	Programmable Logic Controller
PSM	Python SubModule
RTU	Remote Terminal Units
SCADA	Supervisory Control and Data Acquisition
SDN	Software Defined Network
SIS	Safety Instrumented Systems
ST	Structured Text
SWAT	Secure Water treatment
TCP	Transmission Control Protocol
UDN	User Defined Network
UDP	User Datagram Protocol
VM	Virtual Machine

Chapter 1
Introduction

1.1 Introduction to ICS

Industrial Control Systems (ICSs) assume the pivotal role of overseeing, monitoring, and orchestrating industrial operations, encompassing diverse physical processes across sectors such as oil, gas, and electric power. Given the pervasive automation and computerization in contemporary economies, the ubiquity of ICSs in industrial settings is notable. Comprising a multifaceted array of systems, ICSs commonly feature Programmable Logic Controllers (PLCs) as their critical components. These PLCs, serving as compact input/output (I/O) devices, facilitate seamless integration between production processes and the overarching ICS infrastructure, thereby enabling automated control of industrial equipment. This function underscores their critical importance. A Supervisory Control and Data Acquisition (SCADA) system offers operators a robust platform for high-level supervision of industrial processes. Human Machine Interfaces (HMIs) serve as indispensable conduits and offer operators user-friendly interfaces to facilitate interactions with machinery. As the expansive scope of devices within ICSs, we dedicated another chapter for more details.

1.2 Motivations

1.2.1 Security issues in ICS

ICS security aspects have to be well established, studied and assessed due to their life-threatening and business-critical nature. However, this is not the case due to constant evolution in the industrial production processes, and the skills of attackers.

Here we present an overview of the current situation. From a 2024 *Dragos*'s report [27], the number of reported attacks is still increasing, especially in ransomware, by about 50%. Nonetheless, Cyber adversaries with a wide range of capabilities have used Ukraine-Russia and Israel-Hamas wars to conduct targeted operations against critical infrastructure. It is also known that only 52% of IT/ICS facilities have an ICS-specific Incident response plan, and 17% are unsure whether they have such plans, said a SANS report [55]. The report also says that 38% of compromises to ICS come from IT networks. For severe attacks, in May 2017, Renault-Nissan experienced a manufacturing cyber attack involving *WannyCry* ransomware that stopped production at five plants and caused a speculated loss of about 4 billion dollars around the world [11]. In 2021, The US energy pipeline operator Colonial Pipeline faced a ransomware attack [67] that forced it to take down its system and caused massive disruption to the country's energy supplies for weeks.

ICSs are currently experiencing numerous attacks due to the overall lack of security in communication protocols, software, devices, and configurations utilized. These protocols were initially designed for local fieldbus communications(equivalent to LAN types). But with the rise of the Internet, they were updated to support Ethernet and IP. However, they were never intended for such open mediums, resulting in a lack or even absence of security mechanisms. Many lack essential features such as data integrity, authenticity, and confidentiality, relying on air gaps between networks for protection. Moreover, they

now encounter common issues from Ethernet and IP networks. In the past, proprietary protocols, software, and devices relied on obscurity for security, which was effective as long as the technology remained secret. Hardware limitations, implementation bugs, and weak security practices like hardcoded passwords accentuate the problem. Although manufacturers now prioritize security and offer support for updates, equipment is often slow to be patched due to production downtime costs and challenges with legacy devices. The convergence of ICS with IT systems poses a significant challenge for companies, risking financial loss and damage to reputation.

1.2.2 Research Issues in ICS security

With all the security issues in ICSs, there are plenty of research activities to perform. The ICS community has gathered a lot of people from university researchers to manufacturers, specialized cybersecurity companies and amateurs. The research is mainly divided into 2 categories.

1) The initial approach involves direct experimentation with genuine equipment, ensuring high fidelity and accuracy to real-world systems. This method enables the replication of authentic behaviours and facilitates the identification of inherent vulnerabilities and bugs within the operational environment. However, this method is not without its challenges. Conducting security assessments within an actual industrial setting is difficult due to safety considerations for both production processes and equipment integrity. Furthermore, such assessments are limited to specific configurations, diminishing applicability for broader contexts. Additionally, the results obtained from these assessments cannot be readily shared, as they may disclose sensitive information about the utilized systems.

As an alternative, many assessments are conducted within replicated environments, known as testbeds, although this approach also presents its own set of challenges. The cost of hardware, their configuration, maintenance, and modification demands considerable space and time investment, resulting in non-portable, complex-to-reproduce, and challenging-to-share test environments. However, despite these drawbacks, details and findings from these kinds of testbeds can directly be shared with the community.

2) The second approach centres on virtual solutions, which offer significant advantages in terms of flexibility, utility through sharing, and cost reduction. These benefits encompass streamlined setup processes, ease of modifications, enhanced portability, and, with open-source initiatives, the potential for sharing results and source code to prevent redundancy, solicit feedback, and encourage contributions, ultimately fostering higher-quality outcomes.

The development of hybrid testbeds, combining virtual solutions with real equipment to evaluate interoperability, promises to enhance collaboration with the first approach. Emulation is hardly possible due to proprietary devices, it would require insane amounts of work for each device to support. However, the major drawback of software devices is their accuracy. Lower accuracy levels may be easier to manage, to understand and less resource-consuming, while high-level accuracy solutions will produce results comparable to real-world ones. From varying degrees of accuracy, each input contributes uniquely to community advancement, underscoring the importance of diverse perspectives in collective growth.

1.3 Project statement & contributions

According to the current state of security issues and research in ICS environments, we propose in this Master Thesis the following objectives:

- Creation of a general ICS simulator that can support different network topologies, execute software to simulate typical ICS components like PLC, MTU and SCADA system communicating with each other through the communication protocol Modbus/TCP.
- Generation of OpenPLC compatible ladder logic from the configuration file.
- Creation of scripts to automatize the deployment of the simulated system into Docker containers.
- Creation of easily configurable configuration file to create a simulation.

To summarize, we propose an ICS simulator that can be used to discover the most known ICS components, learn their role and how they can be configured, tested and create ICS examples.

1.4 Organization of this document

To explain how we have achieved our goals we organized this Master Thesis as follows.

- The second chapter provides more details about the ICS devices and the communications protocols that will be used in the following chapters
- The third chapter presents the state of the art in the field of ICS simulation.
- In the fourth chapter, we explain the decisions we made, the tools used and the modifications performed to simulate an ICS.
- The fifth chapter focuses on explaining how the physical simulation of the ICS works, its configuration, features and the limitations of the system.
- The sixth chapter provides a step-back discussion about our testbed and existing ones.
- Finally, we conclude this Master Thesis with a reminder of our contributions and some ideas to further improve this work.

Chapter 2
Industrial Control System (ICS)

This chapter will provide more details about ICSs in general, including some devices, and protocols. There is a focus on Modbus/TCP as it is the implemented one for physical simulation. Subsequent sections of the thesis will predominantly concentrate on these devices, recognized as fundamental components within ICS frameworks and frequently cited in existing literature. While acknowledging the existence of numerous other devices, it is noted that the most encountered in the literature have features that overlap each other, making them hard to classify into a specific category. As such, descriptions will maintain a general scope, highlighting essential functions as delineated within this thesis.

2.1 Programmable Logic Controller (PLC)

A programmable logic controller (PLC), is an industrial computing device specifically engineered and ruggedized to regulate manufacturing processes.PLCs can be used to start machines, regulate motor speed, open valves, etc. To do so, they communicate with sensors and actuators, also called field devices (Figure 2.1). These processes can be in various sectors and machinery, including assembly lines, machinery operation, robotic systems, and tasks requiring stringent reliability, simplified programming, and effective fault diagnosis. They can be tailored to use digital and analogue I/O, operate across extended temperature ranges, withstand electrical noise, and endure vibration and impact. PLCs often run on custom embedded operating systems(OS) called firmware. These firmware are designed with real-time considerations and usually provide fewer features than commercial OS. Their mode of communication is usually in master-slave fashion(or client-server). The master(client) will actively poll for data from one or multiple slaves(server) using one or multiple communication protocols.

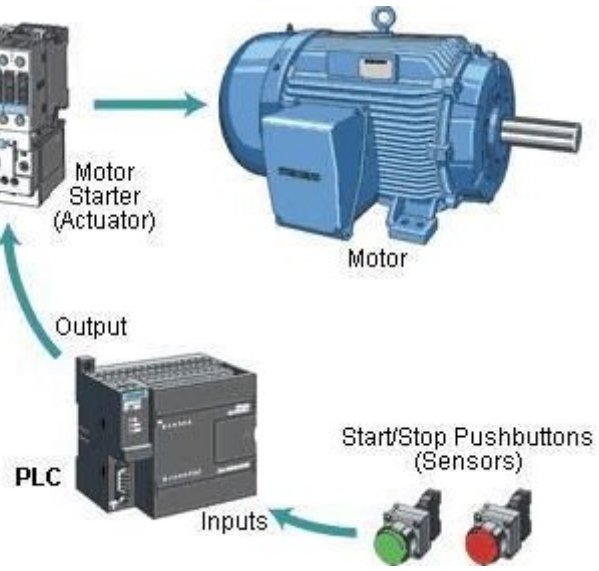

Figure 2.1: PLC connected to sensors and actuators [41]

PLCs pose significant security challenges due to their critical role in industrial operations; any disruption to their functionality halts production processes. However, PLCs themselves have a lot of vulnerabilities. They are highly susceptible to overloads due to limited processing power and memory size. Furthermore, modern PLCs are equipped with increasingly complicated features, such as support for multiple communication protocols and integrated web servers, which increase the attack surface. Additionally, they offer limited physical tamper resistance, as their installations are typically assumed to occur within private environments or secured enclosures.

In literature, we can also find Remote Terminal Units (RTUs) and Intelligent Electronic Devices (IEDs), While RTUs are usually simpler, and are used primarily on data acquisition and basic control functions, IEDs are predominantly used in electric power systems and can provide more complex controls. In this thesis, we regrouped them all under PLC category as most of their functionality overlaps each other, and the fact that PLCs are highly customizable.

PLC runs in an infinite loop called scan cycle (Figure 2.2), this refers to the sequence of operations that a PLC performs to execute its control program and update the outputs based on the inputs.

PLC inputs are classified into two categories, discrete and analogue. Discrete input from a discrete signal will translate values to 0 or 1. Typical discrete input is the start/stop button of a machine. Analogue input represents continuous input from analogue signal, translated into value from 0 to the maximum scale with precision depending on the PLC. Such signals can be the motor speed, water flow, temperature oil pressure etc.

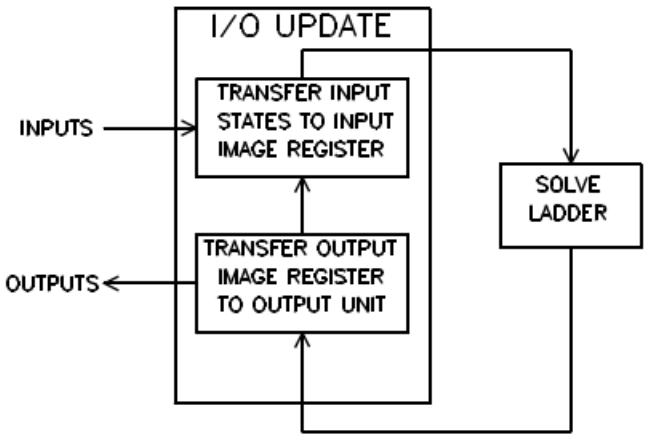

Figure 2.2: PLC scan cycle [39]

Similarly, outputs carry the same types of signals as the inputs and are categorized as discrete or continuous outputs. These signals are transmitted to actuators to effect changes in the status or behaviour of the equipment.

IEC 61131-3 deals with basic software architecture and programming languages of the control program within PLC. It defines three graphical and two textual programming language standards:

1. Ladder diagram (LD), graphical
2. Function block diagram (FBD), graphical
3. Structured text (ST), textual

4. Instruction list (IL), textual (deprecated in 3rd edition of the standard)

5. Sequential function chart (SFC), graphical, has elements to organize programs for sequential and parallel control processing.

We only introduce the Ladder Diagram, also called ladder logic, as it is the most commonly used because of its simplicity and ease of programming with a graphical editor, it comes from relay logic in electric circuits.

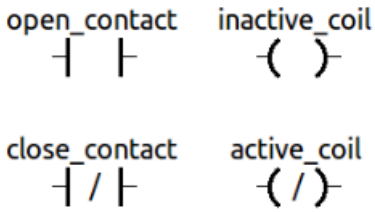

Figure 2.3: Contacts and coils [45]

The contacts serve as inputs to the logical operations (Figure 2.3). The received value can be external, such as from a physical switch or memory. An open contact stops the flow of electric current by default, resulting in a false (0) state for its corresponding input, termed de-energized. In reverse, when the input is true (1) or "energized," the current can traverse the contact. In reverse, a closed contact operates inversely: it allows current passage by default and halts it when its corresponding input is energized.

On the other hand, the coils serve as outputs. They can be transmitted to an external module or retained in internal memory to preserve their current state for subsequent logical operations. An inactive coil defaults to a de-energized state and becomes energized when current flows through it. In reverse, an active coil defaults to an energized state and becomes de-energized when current traverses it.

While contacts and coils handle discrete values, variables handle analogue values. These variables can serve as input as well as output, depending on the location.

A lot of other elements called function blocks provide built-in operations like arithmetic (addition, multiplication,...), selection (min, max,...), comparison (greater-than, equals,...), bitshift, bitwise and time. We won't describe them as it is not required for the use of our sandbox. We use the simplest possible ladder logic to reduce the learning cost.

In ladder logic, each horizontal line is called a rung and can be interpreted from left to right. A logic structure comprises a series of rules that are sequentially assessed from top to bottom, constituting a scan. If the scan time is small enough(typically less than 100 milliseconds), the simultaneity and immediacy of relay logic can be replicated. However, since execution remains sequential, the order is crucial. A recommended practice is to position coils at the end of a rung to visualize the generated outputs clearly. If the evaluation of a rung is halted prematurely, perhaps due to an open contact, the coils will maintain their default state or the value assigned during a preceding rung. A coil will retain its status for the next scan; If the value is not refreshed, it will return to its default state. The same process applies to variables, with the exception that there is no open/close state, but directly the analogue value.

Despite its apparent simplicity, ladder logic can easily contain bugs that are difficult to identify and rectify. In the case of this simple example (Figure 2.4), the stop button

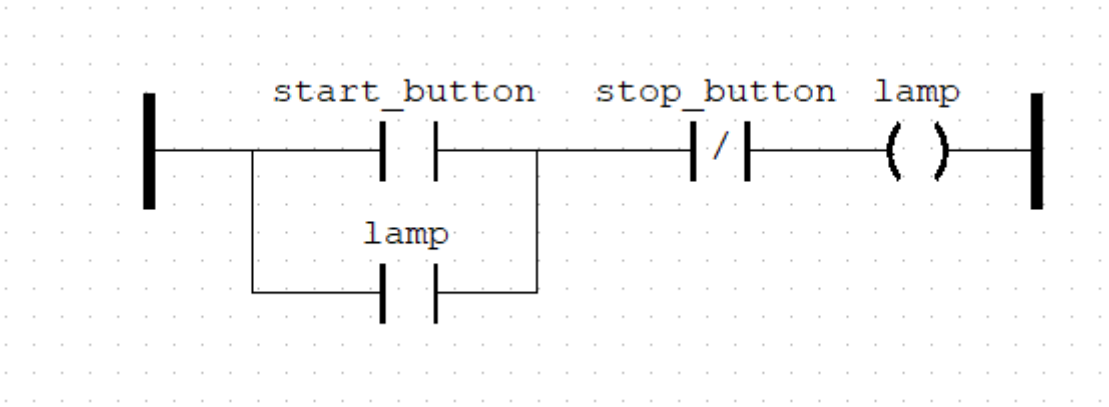

Figure 2.4: Ladder logic example

is dominant over the start button. When simultaneously pressed, the stop always wins. If the start button is stuck in a pressed state, the only way to halt the process is to persistently press the stop button until the start button is eventually repaired and released. This scenario can pose significant safety risks in certain situations, underscoring the need for Safety Instrumented Systems (SISs) to be implemented. While these are just two fundamental considerations, complex situations will need even more precautions. Many vendors furnish software alongside their devices for logic editing and testing purposes, aiming to mitigate the occurrence of bugs in the final production environment, which could result in financial losses and endanger lives.

2.2 Supervisory Control And Data Acquisition (SCADA)

In the industrial control system hierarchy, the top tier is known as Supervisory Control and Data Acquisition (SCADA). SCADA systems are employed for monitoring and managing centralized data gathered from various field sites. This setup integrates data transmission and acquisition systems with HMI (Human Machine Interface) to centralize the data.

SCADA systems are composed of different types of devices, usually, there are PLC(explained above), MTU(Master Terminal Unit) which gather data from RTU or PLC, and HMI to visualize and control field operation. Furthermore, all important data from fields are saved in a data Historian (a kind of database).

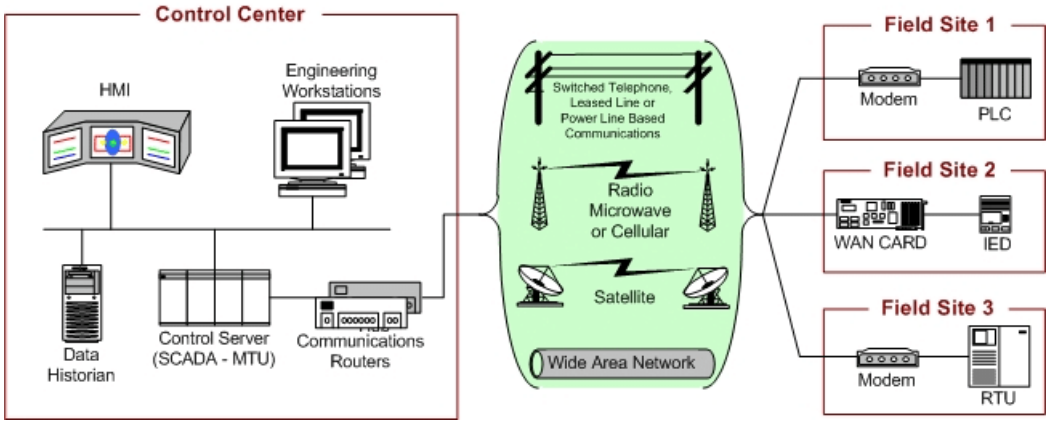

Figure 2.5: SCADA Basic Topology [64]

The workflow of a SCADA system can be resumed as shown in Figure 2.5: data from PLC and RTU are gathered and processed by the Control server(SCADA) after which, data will be displayed by the HMI to the operator, and sent to the Data Historian to be saved. Based on what the operator sees, the operator can take action, to regulate the whole process(this process may be automatized).

The main security of SCADA systems networks was that these networks could not be accessed by an attacker because they were electronically isolated from other networks. But now with the transformation to Industry 4.0, the wish to increase connectivity between factory and the cooperated zone is huge. IoT devices and Cloud computing, which are new factors in the industry, are also contributing to this kind of change. All of these new factors increase the attack surface, increasing considerably the risk. Furthermore, over the years, SCADA systems protocols moved from proprietary Standards to open international standards, resulting in a better comprehension of ICS protocols for attackers. This is why it is important to continue security research on ICS.

But now, with the wish to increase the connectivity between the factory floors and the cooperated zone, the network security of these SCADA systems has decreased as there are multiple access points that a determined attacker could exploit. Furthermore, over the years SCADA systems protocols moved from proprietary Standards towards open international standards, resulting in attackers knowing precisely the protocols. That is why there is a gain of interest in reinforcing industrial control systems security. Therefore, multiple challenges need to be addressed. [2] The most obvious one is to reinforce the access controls to the SCADA networks, due to the difficulty of defining the scope of a SCADA network. IDS and IPS for ICS still have not been completely studied. Development of testbeds and datasets is still needed for better testing and validation. Forensic analysis, incident response and decision-making also need to be tailored.

2.3 Master Terminal Unit (MTU)

The Master Terminal Unit (MTU) serves as the central hub of the SCADA system, depicted in Figure 2.5. It initiates communication with the RTU or PLC, collects data from the PLCs, and processes it. When the MTU requests data from a PLC, the PLC responds, establishing a master-slave communication dynamic where the MTU acts as the Master and PLCs as the slaves. Messages from the MTU to the PLCs can be operator-triggered or automatic. These messages can be reading memory segments representing current values such as water flow, oil pressure, or tank temperature, or writing values to memory to modify configurations. Like PLCs, MTUs present security challenges; if an attacker gains access to an MTU, they can manipulate PLC configurations. Additionally, MTUs encounter similar security issues as PLCs.

2.4 Human Machine Interface (HMI)

The Human Machine Interface (HMI) serves as a graphical interface enabling operators to interact with PLCs, RTUs, IEDs, or MTUs. HMIs visually present information regarding production status and values, helping operators interact with the control process, adjusting set points, initiating or halting cycles, and more. Figure 2.6 illustrates typical interactions between HMIs, controllers, and the field plant floor. Acting as a link between PLC logic and human operators, HMIs simplify complex PLC operations, allowing operators to focus

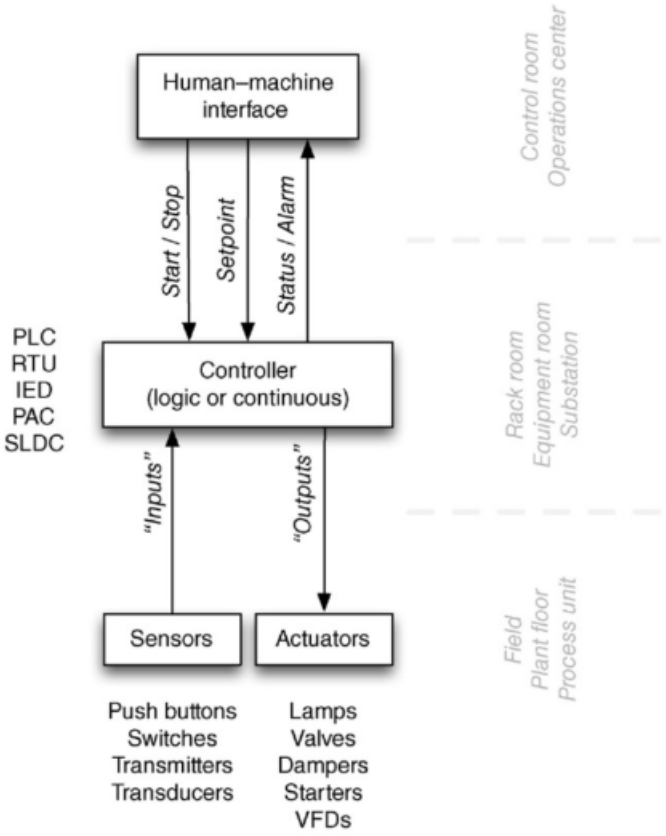

Figure 2.6: HMI interaction [42]

on the process.

Regarding security, access control is a key concern. While authentication may not always be required for HMI access to ensure uninterrupted operation during abnormal events, such as password lockouts, this practice is generally acceptable when utilized by trained, authorized personnel in physically secure environments, though not entirely without risk.

2.5 Communication Protocols

Many protocols are used in industrial environments, and many are proprietary protocols. Here is a non-exhaustive list of commonly used protocols, Modbus, DNP3 and Ethernet/IP will be more detailed as they are used in this project.

2.5.1 PROFINET

PROFINET(Process Field Network) is an industrial Ethernet-based communication standard that offers a wide range of applications. The fieldbus uses TCP/IP and information technology standards, enabling real-time processing and the integration of fieldbus systems. PROFINET was developed by Siemens and member companies of the PROFIBUS User Organisation(PNO) and is used in conjunction with Siemens control systems. Thanks to the IEC 61158 and IEC 61784 standards, PROFINET has been standardized and, as a universal communication technology, covers all automation technology requirements. It is widely used in sectors like Manufacturing, Robotics, Logistics, Water treatment etc. In

terms of security, PROFINET lacks of security (Authenticity, Integrity, and Confidentiality), but guidelines provided by PI North America should resolve these problems. These guidelines are divided into 3 classes. Class 1 provide robustness, by tightening up security for the DCP & SNMP protocols and protecting GSD files. Class 2 provides authenticity between a controller and a device to ensure only allowed nodes can join the network, preventing MITM attacks. Finally, class 3 provides encryption to the real-time I/O data. However, only class 1 security guideline [59] is officially published.

2.5.2 EtherCat

EtherCAT or Ethernet Control Automation Technology is an industrial network system that depends on the Ethernet system used to attain very fast and more proficient communications. So, EtherCAT is a very fast network used to process data with dedicated hardware& software. This network uses a master-slave, full duplex configuration with any network topology. EtherCAT is used to overcome the typical faults of industrial Ethernet through its high-performance operation mode, where usually a single frame is enough to transmit & receive control data from and to all nodes. EtherCAT protocol is built on the physical layer of Ethernet. However, EtherCAT utilizes a processing-on-the-fly approach for transport & message routing which is also called communicating-on-the-fly rather than using TCP/IP. In terms of security, as EtherCAT is based on Ethernet, it is susceptible to any of the vulnerabilities of standard Ethernet [43]. EtherCAT over UDP is transactionless, so there is no inherent network-layer mechanism for reliability, ordering or data integrity checks. Due to its excellent real-time performance, EtherCAT is also highly sensitive and susceptible to DoS attacks. EtherCAT is also subject to spoofing and MitM attacks due to the lack of bus authentication. Fortunately, EtherCAT can use the protocol Safety over EtherCAT: FailSafe over EtherCAT(FSoE) [30]. This allows

- A single communication system for both control and safety data
- The ability to flexibly modify and expand the safety system architecture
- Pre-certified solutions to simplify safety applications
- Powerful diagnostic capabilities for safety functions
- Seamless integration of the safety design in the machine design
- The ability to use the same development tools for both standard and safety applications

2.5.3 POWERLINK

POWERLINK is an Industrial Ethernet solution devised to give users a single, consistent and integrated means for handling all communications tasks in modern automation. It is generally suitable for all conceivable applications in machine and plant engineering as well as for process industry applications. A POWERLINK network integrates all components in industrial automation, such as PLCs, sensors, I/O modules, motion, controllers safety controls, safety sensors and actuators and HMI system. POWERLINK is a real-time Industrial Ethernet protocol, and as such it is susceptible to any of the vulnerabilities of other forms of Ethernet communication. As with many real-time Ethernet protocols, POWERLINK is sensitive and susceptible to DoS attacks and can be easily disrupted via

the insertion of rogue Ethernet frames. Thus requiring the separation of POWERLINk from other Ethernet systems.

2.5.4 Modbus

Modbus was designed by Modicon in 1979, and integrated into the first PLC. It became a de facto standard communication protocol for ICS due to its simplicity, low cost, open standard, and raw messages that make it easy to use without authentication restrictions or excessive overhead. Since then, it is still widely supported by members of the Modbus Organization, which still operates today.

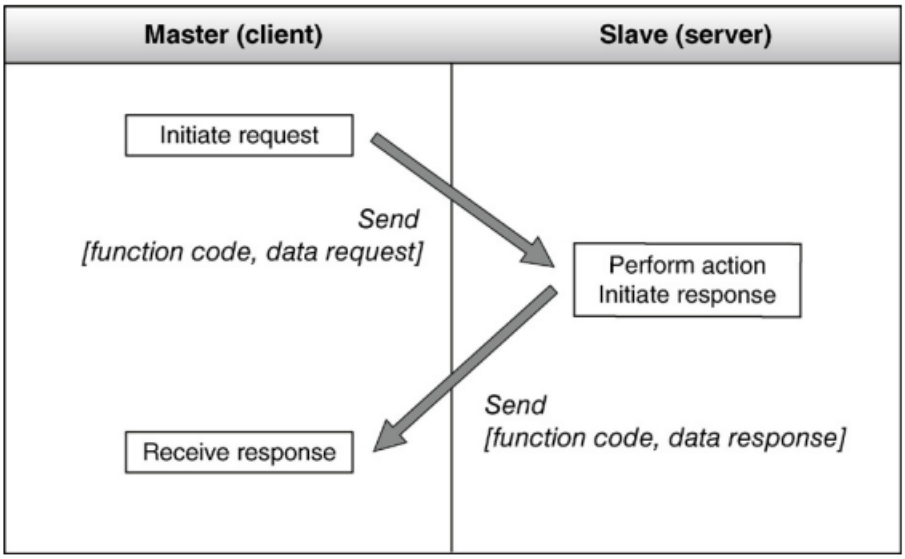

Figure 2.7: Modbus transaction (error free) [43]

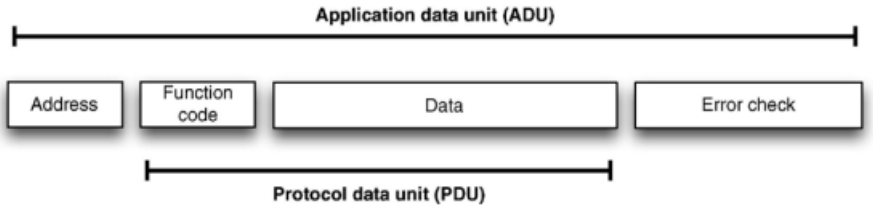

Figure 2.8: General Modbus frame [43]

Modbus allows for efficient communications between interconnected assets based on a "request/reply" methodology as Figure 2.7. A "transaction" begins with the transmission of a function code and a data Request within a Protocol Data Unit (PDU). If there are no errors, it will respond with a Function code and a data Response within a Response PDU. If there are errors, the devices will respond with an Exception Function Code and Exception Code within a Modbus Exception Response. As Modbus operates at layer 7 of the Open Systems Interconnection (OSI) model, the application layer can be adapted for different under layers by adjusting the Application Data Unit (ADU) (Figure 2.8).

Modbus defines data in 4 categories as shown in Table 2.1.

Discrete inputs and coils are binary values represented as 0 or 1 (false or true). Input and holding register values can range from 0 to 65535, and can represent any physical data that you want. For example. it can be used to represent temperatures, water flow, pressure,

Data Type	Size	Access	Address Range
Discrete Input Contact	1 bit	Read-only	00001-09999
Discrete Output Coil	1 bit	Read-write	10001-19999
Input Register	16 bits	Read-only	30001-39999
Holding Register	16 bits	Read-write	40001-49999

Table 2.1: Modbus Data Model

etc. Additional registers can be utilized for increased precision or to accommodate larger values. Each address range designates its first bit as 1 to signify a data type and can store a maximum of 10,000 values, although newer Modbus specifications extend this limit to 65,536 values. An important point is that Modbus lets the different PLC manufacturers define their mapping to real memory addresses. Not all devices support the entire address range or all data types. In the chapter dedicated to the experiment, we will describe the mapping used by pyModbusTCP and OpenPLC. Additionally, Modbus is unable to write to discrete inputs and input registers, as these are designated as read-only and reserved for external I/O modules.

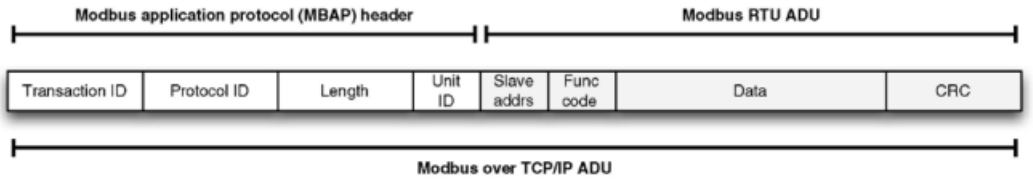

Figure 2.9: Modbus TCP [43]

When Modbus first started out, it had two versions: Modbus RTU and Modbus ASCII. These were made to work on serial lines like RS-232 and RS-485. But as people wanted devices to talk to each other across networks, a new version called Modbus/TCP was created. Modbus/TCP removes the legacy address and error checking, and places only the Modbus PDU together with an MBAP header into a new frame (Figure 2.9).

Modbus was not designed with security issues in mind as they were not of any concern during its creation. Those issues are critical with Modbus/TCP now being transmitted over untrusted networks :

- Confidentiality: Everything is transmitted in clear, captured packets can be read without further processing
- Integrity: There is no mechanism to detect possible modifications on the messages.
- Authenticity: No authentication is required to exchange messages. The slave always responds to any master's request.

An attacker can find out important details about the system by simply capturing packets and analysing them. They can also capture, modify, forge and replay messages, and nothing can detect it. By exploiting these vulnerabilities, attackers can easily disrupt devices and cause losses. It could even put people in danger. This is the reason why lots of researchers are working on ICS, to improve its security.

In 2018, the Modbus organization released a security guideline which embeds the Modbus MBAP PDU into TLS (Figure 2.10). This secure version of Modbus TCP, by the mean of TLS, allows to mitigate the following problems:

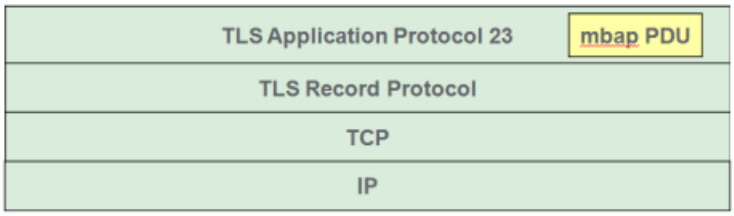

Figure 2.10: Modbus TCP [50]

- Authenticity: Mutual client/server TLS authentication
- Confidentiality & Integrity: x.509v3 certificate-based identity and authentication with TLS, allow the encryption of TLS payload, thus providing confidentiality, and the hash function providing integrity
- Anti-replay attack: TLS prevents replay attacks through sequence numbers, timestamps and nonce.
- Authorization: via information embedded in the certificate such as user and device roles.

2.5.5 DNP3

Distributed Network Protocol (DNP) originally started out like Modbus, as a serial protocol connecting master stations with slave devices. However, DNP3 stands out because it's very reliable, efficient, and perfect for transferring real-time data. It uses standardized data formats and supports time-stamped data, which makes real-time transmissions more efficient and reliable. DNP3 also ensures reliability through the frequent use of cyclical redundancy checks (CRC). Unlike Modbus, DNP3 is bidirectional, meaning it supports communication from both the master to the slave and vice versa. It also supports exception-based reporting. Additionally, DNP3 can use link-layer confirmation to enhance reliability. If a confirmation request isn't received, the link layer will resend the frame. This confirmation is optional because it adds some overhead, which may not be desired in real-time environments.

DNP3 provides a method to identify the remote device's parameters and then use message buffers to identify incoming messages based on event data classes 1 through 3. This allows the master station to only collect new information resulting from changes in data points or events on the outstation.

At first, the master station typically sends a class 0 request to the outstation, asking for all point values to be read into its database. When a change occurs on an outstation, a flag is set to indicate that there is new data. Then, the master station is then able to poll only those outstations where there is new information to be reported.

Communication starts with the master station sending requests to the outstation, as shown in Figure 2.11.

Because DNP3 operates bidirectionally and supports unsolicited responses, as shown in Figure 2.12, each frame requires both a source address and a destination address so that the recipient device knows which messages to process, and which device to return responses to. This extra address does add some overhead. In contrast, purely master/slave protocols don't need a source address because the master is always the source. However, this overhead brings the benefit of significantly increased scalability and functionality.

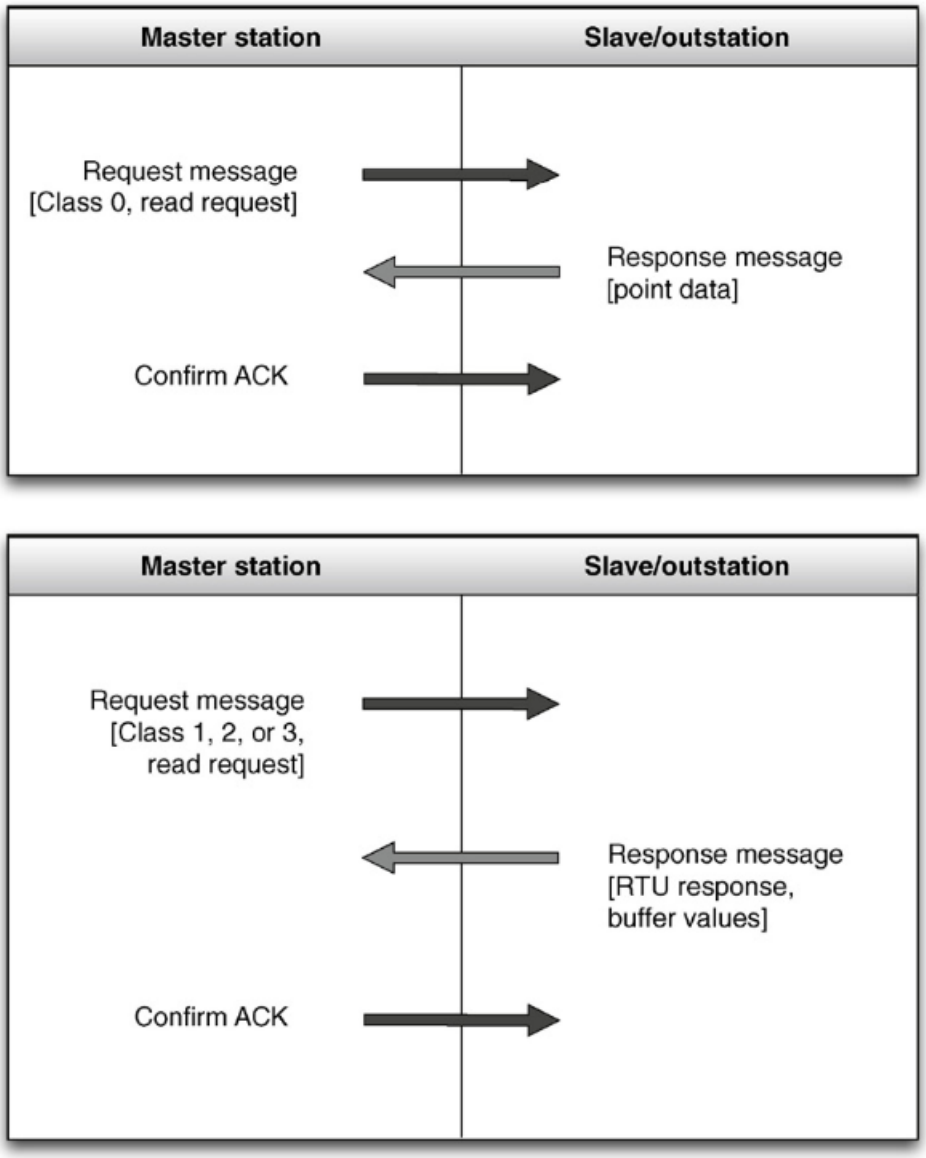

Figure 2.11: DNP3 protocol operation [43]

Secure DNP3 is a variant of DNP3 that includes authentication in the process of sending/receiving data, as shown in Figure 2.13. This authentication is done by issuing a challenge to the device receiving the data. The challenge can happen when a session begins (like when a master station starts communicating with an outstation), after a certain period (usually 20 minutes), or when certain critical actions occur, such as writes, selects, operates, direct operates, starts, stops, and restarts. Authentication uses a special session key that is combined with data from the sender and the challenger. This process verifies authority (checksum against the secret key), integrity (checksum against the sending payload), and pairing (checksum against the challenge message). This makes it really hard for someone to mess with the data, insert malicious code, or spoof/hijack the protocol.

The DNP3 Layer 2 frame contains information about where the data is coming from, where it's going, the type of data, and the actual data itself. It can work over different types of internet connections, including TCP and UDP over IP, which support TLS for added security.

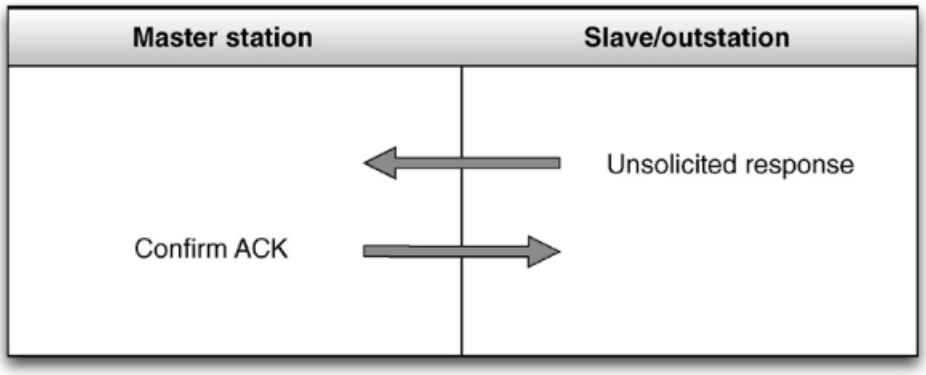

Figure 2.12: DNP3 Unsolicited responses [43]

2.5.6 Ethernet/IP

EtherNet/IP(Ethernet Industrial Protocol) (EIP) is an open fieldbus standard based on the Common Industrial Protocol (CIP). It was developed by Rockwell Automation and the ODVA(Open DeviceNet Vendor Association) and is primarily used in the field of automation technology. Devices that use this protocol not only adhere to its standards but also have their compatibility certified by the ODVA.

EIP or CIP on Ethernet uses regular Ethernet frames to talk to nodes, alongside the CIP suite. Like other CIP systems, EIP lets you connect I/O, control, data collection, and device setup on one network. For real-time I/O and control data, EIP uses a type of messaging called "implicit messaging" over multicast UDP (port 2222/udp). This setup lets devices quickly share data with each other, which is important in ICS setups. There's also a method called "explicit messaging" using unicast TCP (port 44818/TCP) for sending larger data amounts used in device setup, diagnostics, and event info.

The Common Industrial Protocol (CIP) uses object models to describe different aspects of a device. Each CIP object contains attributes (data), services (commands), connections, and behaviours (how attributes and services are related). There are three types of objects:

1. Required Objects: These define essential attributes like device identifiers (such as manufacturer, serial number, and date of manufacture), routing identifiers for messaging between objects (Message Router Object), and physical connection data (Network Object).

2. Application Objects: These define input and output profiles for devices.

3. Vendor-specific Objects: These allow vendors to add their own custom objects to a device.

Except for vendor-specific objects, objects are standardized based on device type and function, ensuring interoperability. For example, if one brand of pump is replaced with another, the Application Objects will remain compatible, avoiding the need for custom drivers. The widespread use and standardization of CIP have led to a large library of device models, which promote interoperability but can also be used for control network scanning and enumeration.

EtherNet/IP is a real-time Ethernet protocol, and as such it is susceptible to any of the vulnerabilities of Ethernet. The CIP does not define any explicit or implicit mechanisms

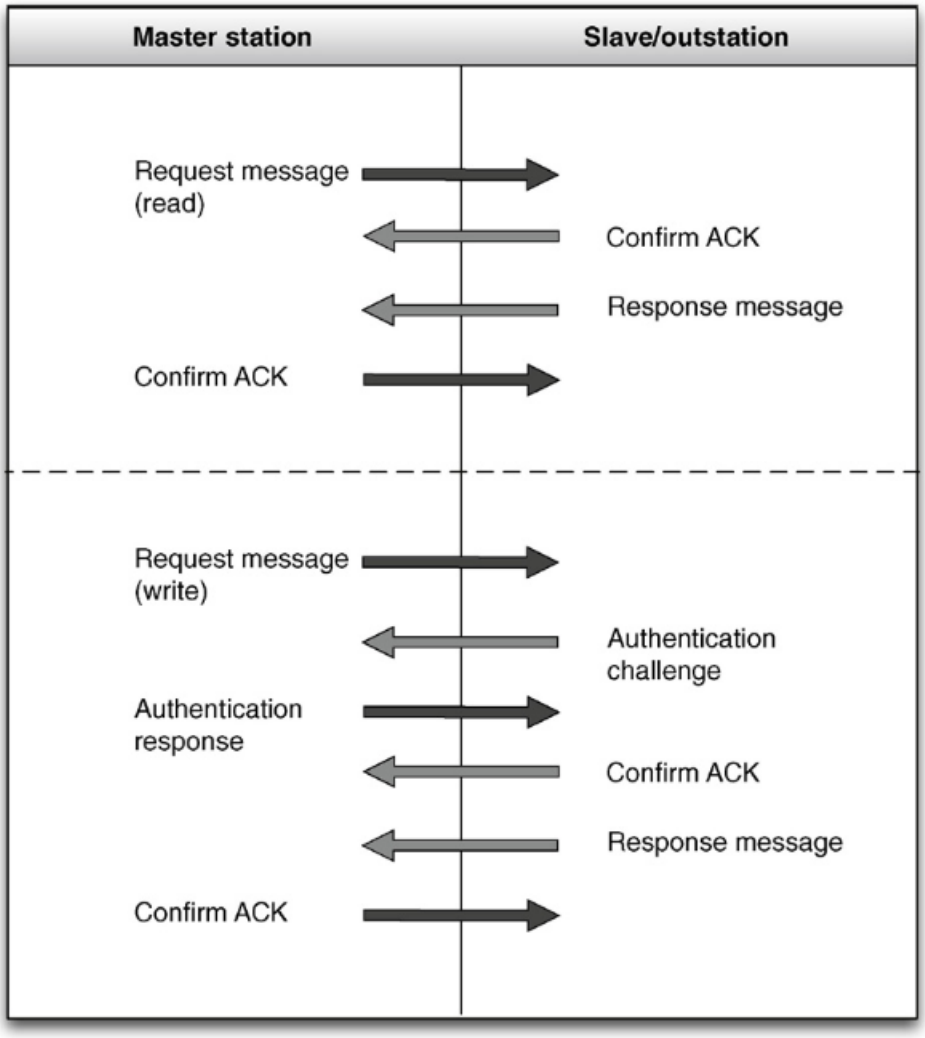

Figure 2.13: Message confirmation and secure DNP3 authentication [43]

for security. The use of common Required Objects for device identification can facilitate device identification and enumeration, facilitating a targeted attack. The use of common Application Objects for device information exchange and control can enable broader industrial attacks, able to manipulate a broad range of industrial devices. EtherNet/IP's use of UDP and Multicast traffic—both of which lack transmission control—for real-time transmissions facilitate the injection of spoofed traffic or (in the case of multicast traffic) the manipulation of the transmission path using injected IGMP controls.

Many other industrial communication protocols exist. Some alternatives to Modbus are Profibus and HART which were also originally fiedlbus protocols. Protocols designed with routing capabilities like EtherCAT and Ethernet/IP. With the integration of Industrial 4.0, Wireless protocols like ZigBee, WirelessHART, Bluetooth to provide mobility, scalability and accessibility and Internet protocol to facilitate data sharing, analysis, and visualization for ICS like HTTP, MQTT, CoAP and OPC UA increase considerably the attack surface on network protocols. They also come with many variants. For example, DNP3 Secure adds challenge-response authentication to DNP3. Each protocol presents its vulnerabilities raising the difficulties to achieve overall ICS security.

Chapter 3
Related work

3.1 Testbeds

This chapter presents some works in the field of ICS simulation. Different sectors have their specificities, and it would not be possible for us to cover all of them and some of them would be far from our research aim. Most solutions are private or only available to insiders, but some interesting open-source projects are available.

We start with the survey of Holm et al. [40] that presents general commonalities between 30 ICS testbeds. Most testbeds are based on real hardware or make use of virtual devices and sometimes use a mix of the two. The most common devices inside ICS are RTU, PLC, MTU, HMI and IED. For the protocols, Modbus, DNP3, OPC, IEC 60870 and IEC 61850 are the most discussed. When there is a simulated physical process, they are simulated via models from Matlab/Simulink, LabView, PowerWorld or other libraries. Only a few testbeds cover the fidelity of the proposed solution against real-world systems.

Then we searched for more recent surveys and found one proposed by Conti et al. [15]. This survey not only provides a newer view of more than 80 different ICS testbeds and datasets. The most present communication protocols in testbeds are Modbus, DNP3 and EtherNet/IP, while in the market, EtherNet/IP has dethroned Modbus/TCP to be the most employed one. Profinet is widely used in ICS but is only present in 3.2% of the analysed testbeds. Some protocols such as BACnet, TridiumFox, NiagaraFox and EtherCAT which gain in market share are not present in the analysed testbeds. They also highlight key challenges of ICS testbeds: Design guidelines, Real World Representation, Replication in Safety, Complexity, Cost, Lack of documentation, Reproducibility, Scalability and Data Collection. Finally, they suggest good practices in the development of testbeds, datasets and IDSs.

These surveys helped us understand what are the possible approaches to the problem. The difficulties and limitations of simulating an ICS environment.

The following sections will present related works, separated into multiple categories although some projects actually can be present in more than one category.

3.1.1 Hardware

In this master thesis, we decided to focus on fully virtual solutions as buying real industrial PLCs or devices was not an option due to cost and lack of infrastructure. But as research, it is interesting to have a global overview of different testbed solutions, including hardware testbed. And we found that testbeds that use physical devices are generally mixed with simulated components for the sake of cost saving. These testbeds are usually called Hybrid testbeds, while some of them use the term Hardware in the Loop (HIL) as some hardware devices are embedded in the process loop.

Several countries have established government-sponsored SCADA testbeds at the national level to promote security research and development in the energy sector. For instance, the U.S. Department of Energy established the National SCADA Test Bed (NSTB) [53] in 2003, while in Japan, the Control System Security Centre [65] Tohoku

Tagajo Headquarter Testbed has been in operation since 2012. Additionally, the Idaho National Laboratory constructed a groundbreaking 61-mile 138kV transmission testbed, which is the world's first grid-tested environment with full-scale replication of real hardware and software equipment, encompassing seven substations and over 3,000 monitoring sites.

The testbed from Haihui G. et al. [35] employs a combination of physical and simulated components in its research. Real physical PLCs and RTUs were utilized, communicating with the emulated corporate network through a specialized hardware device. To minimize costs and maximize the reusability of input/output layer components, the researchers chose not to employ actual industrial boilers or gas pipelines. Instead, they simulated various control processes using Matlab/Simulink and mathematical models derived from the analysis of an industrial oil-fired boiler. This approach enabled them to effectively investigate and test their proposed methods while maintaining cost-efficiency and flexibility.

The Secure Water Treatment(SWAT) testbed [37] is a testbed founded by the Ministry of Defence of Singapore, created by the iTrust lab of Singapore University of Technology and Design. The testbed is a scaled-down, high-fidelity, industry-compliant emulation of a modern water treatment facility. The testbed consists of multiple physical and cyber components that combine to form a modern six-stage water treatment process. A high-level diagram of the treatment process is documented in Figure 3.1. This physical testbed has generated numerous datasets by collecting network traffic and all the values obtained from all the 51 sensors and actuators. Some datasets are data obtained under some attack scenarios, allowing the development of attack detection systems.

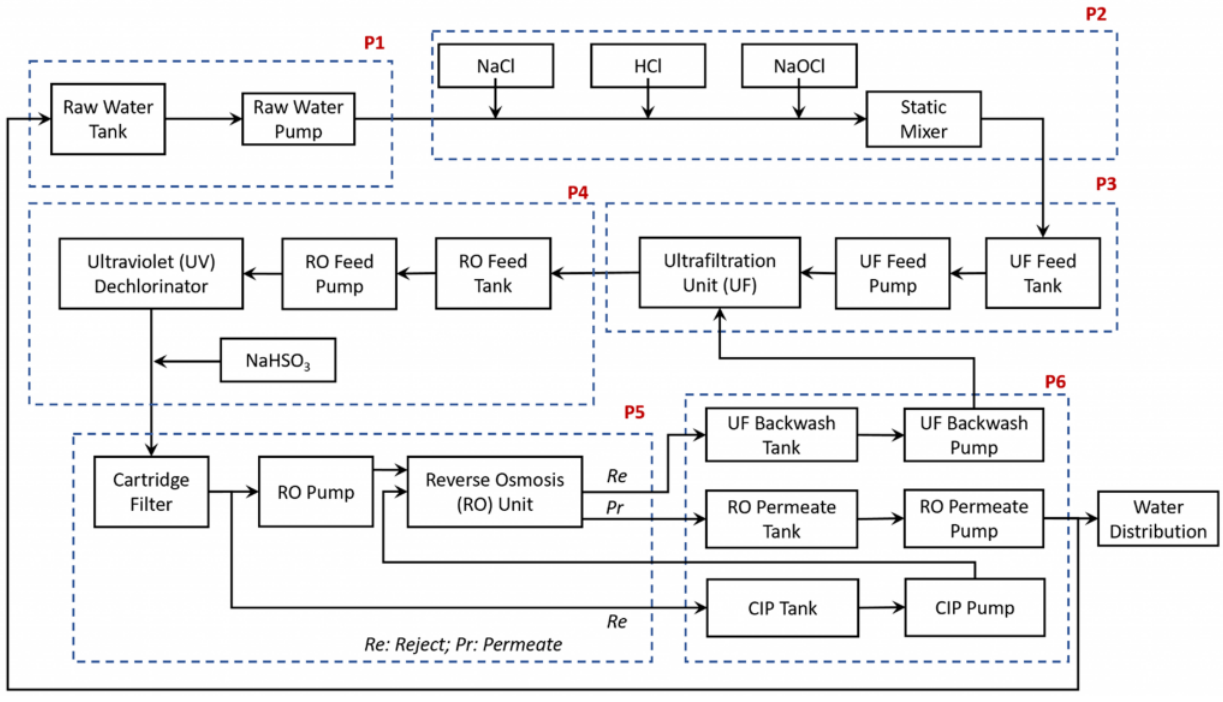

Figure 3.1: Block Diagram of SWaT Physical Process [37]

3.1.2 Hybrid Testbeds

L. Faramondi et al. [31] introduces a dataset designed to support researchers in validating solutions such as Intrusion Detection Systems (IDS) using artificial intelligence and ma-

chine learning techniques for detecting and categorizing threats in Cyber-Physical Systems (CPS). The dataset was obtained from a hardware-in-the-loop Water Distribution Testbed (WDT), which simulates water flow between eight tanks using solenoid valves, pumps, pressure sensors, and flow sensors. The testbed consists of a real subsystem virtually connected to a simulated one. The simulated subsystem was implemented using minicps [6], a CPS simulation toolbox based on mininet, a network simulation tool. Four machine learning algorithms, namely KNN, RF, NB, and SVM, were evaluated on both network and physical datasets. The results indicated that classification algorithms alone cannot detect all attack types when applied separately to physical and network datasets. Thus, better performance is achieved when both network and physical data are considered.

Felix et al. developed LICSTER [62], an innovative open-source low-cost ICS testbed, that offers an affordable solution for researchers and students seeking hands-on experience with industrial security, priced at around 600 Euro. Leveraging OpenPLC_v3 on Raspberry Pi to simulate PLCs, ScadaLTS as a logging system, and a modified PyModbus as an HMI, LICSTER provides a comprehensive educational and research platform. Additionally, a switch and a Fischertechnik punching machine are integrated into the system to create a physical process. This emphasis on low-cost and open-source tools makes LICSTER an accessible option for educational and research purposes. The inclusion of a physical process component in LICSTER is particularly noteworthy, as it allows users to gain a tactile understanding of the impacts of cyber attacks on ICSs.

Smadi et al. survey on Cyber-Physical SmartGrid (CP-SG) testbeds [63] highlight their key criteria and challenges encountered during the survey on the different ICS simulations.

Criteria:

- Fidelity: Accurately replicate the devices and processes from a real-world ICS.
- Repeatability: Allows other researchers to reproduce the findings and compare other solutions on the same system
- Flexibility: Ability to be redefined for alternative test cases or scenarios.
- Safety: Be able to run different tests and scenarios in a safe manner.
- Cost-effectiveness: Must be within the financial budget and affordable for research purposes.
- Scalability: Ability of the testbed to increase the size of the setup
- Diversity: Ability of a testbed to incorporate a wide range of components and devices without affecting its scalability

Challenges:

- Design and Operation: Latency issues due to speed requirements.
- Cybersecurity: Need for identity management, risk control, and operationalization.
- Internetworking: Lack of built-in security in power grid communication.
- Efficiency and Scalability: Importance of system availability.
- Security Objectives: Focus on reliability and equipment protection for CP-SG, and data protection for enterprise networks.

Vulnerabilities:

- Cyber: Communication, software, and privacy vulnerabilities.
- Cyber-Physical: Vulnerabilities in network communication and smart meters. Addressing these challenges is crucial for the development of secure and efficient CP-SG systems.

3.1.3 Emulation

Emulation of PLC is hard and requires way too much work, but would be the most accurate solution to virtualize devices. Few projects interest themselves in this type of virtualization. But there are still some interesting ones.

Awlsim [47] is an open-source Python project designed to emulate the execution of PLC logic, mimicking the behaviour of a real Siemens S7 CPU. It is capable of running logic encoded in STL (Statement List), which corresponds to the Instruction List (IL) language from the IEC 61131-3 standard, with additional vendor-specific features. Particularly for Siemens, STL closely resembles the machine code running on their S7 CPUs, allowing for optimizations. Awlsim supports a large subset of S7 instructions and strives to replicate the behaviour of real Siemens hardware, such as S7-300 and S7-400 CPUs. To interact with the running logic, it provides functionality to simulate inputs and read output values. Additionally, Awlsim can run on PiXtend V1.3, a Raspberry Pi-based PLC, and interface with its built-in I/O modules, including digital/analogue inputs/outputs and relays. Furthermore, Awlsim includes PROFIBUS-DP and LinuxCNC communication modules, used in the exchange of messages.

In their research, Babu and Nicol [9] developed a network simulation using Awlsim PLCs. The network simulation was managed by the discrete network simulator S3F, with modifications made to Awlsim to support the Modbus protocol. They introduced a kernel modification known as TimeKeeper to maintain precise control over the Awlsim emulators and the network simulator by providing a virtual clock. This virtual clock, replacing the hardware clock, ensures synchronization under different configurations. The speed of the virtual clock depends on the chosen Time Dilatation Factor (TDF) which represents the additional time given to the system to compute a step. With this additional time, the system can run multiple instances of Awlsim PLC on commodity hardware while keeping a steady behaviour. The authors compared packet delays, throughputs, and compute times between scenarios controlled by TimeKeeper and best-effort scenarios. Their findings revealed that the controlled environment consistently produced accurate results, even with 7 to 63 PLC nodes, whereas the best-effort environment lost accuracy when increasing the number of nodes. As a future research direction, the authors suggested developing an Intrusion Detection System (IDS) on the testbed, leveraging precise network traffic patterns within this accurate environment.

3.1.4 Virtualized(software) Testbeds

This category interests us the most as our objective is to create a novel virtualized ICS sandbox. The significant advantages of such virtual solutions include portability, ease of modification, and the potential to collaborate with the community in open-source projects. During our research for ICS simulations, we encountered numerous projects specializing in

smart grids. Smart grids are electric supply networks that rely on modern digital communications. They differ from typical ICSs as their extensive networks comprise thousands of devices, and remote substations, serving thousands of customers. Despite these differences, they hold great relevance for ICSs, offering valuable insights and potential cross-pollination of ideas between the two domains.

In our exploration of smart grid solutions, we encountered a standout: Mosaik [52]. Mosaik is an open-source co-simulation framework written in Python, designed to integrate various simulators such as power plants, photovoltaic, and wind turbines into a unified scenario with thousands of simulated entities.

What sets Mosaik apart is its direct relevance to power grids. It not only provides simple simulators and scenarios but also interfaces with PYPOWER, a power flow solver. This connectivity enhances its utility in modelling complex power grid scenarios. One of Mosaik's key features is its synchronization mechanism, which operates in discrete time. This ensures that all simulators advance together step by step, facilitating seamless interaction between connected simulators. Mosaik's ability to orchestrate interactions between diverse simulators makes it a valuable tool for exploring and analyzing complex smart grid scenarios.

Thiago et al. paper discusses a modular approach to virtualization of various components of SCADA systems [4]. For physical systems, modelling software like Simulink/-Matlab, HyperSIM, or EMegaSIM can models the physical process, and then Simulink can be used to perform data transmission over TCP/IP and UDP networks to DCS stations. Virtualization of DCS is achieved using VPLC. They use OpenPLC as a fully functional PLC, which was developed by the same author, and run on a Hypervisor for hardware-efficient virtualization. Higher fidelity network virtualization can employ software-defined networking (SDN) and network functions virtualization (NFV) for flexible management and emulation of network node functions. Alternatively, lower fidelity virtualization uses the hypervisor's virtual network or the host machine's virtual network, but which may lack additional network devices like routers or switches. Remote monitoring and control can be virtualized with ScadaBR, a Java-based application that runs on Windows and Linux. According to the example cited in the paper, this novel virtualization approach has proven efficient and portable, reducing SCADA testbed size and cost while maintaining high fidelity.

Israel Barbosa et al. developed a Nuclear Power Plants (NPP) testbed [13]. It employs the Asherah NPP simulator in Matlab/Simulink for physics simulation, along with GNS3 for network simulation, all running on Windows OS. OpenPLC is used as the software PLC on an ESP8266 Arduino board. ScadaBR serves as SCADA/HMI. A Man in the Middle (MitM) attack was conducted using Ettercap to assess its reliability. The simulation demands substantial resources, particularly with the GNS3 VM. The computer used to make the simulation required an AMD Ryzen 7 3700X, 32GB RAM. They allocate 6GB to the Windows/ANS(Nuclear) VM and 2GB for each Linux VM.

Queiroz et al. introduced SCADASim [60], a framework for building SCADA simulations. The SCADASim framework developed at the Royal Melbourne Institute of Technology, Melbourne, Australia, provides predefined modules for building SCADA simulations, employing the OMNET++ discrete event simulation engine to recreate typical SCADA components while providing an underlying inter-model communications layer.

Alirea et al. developed ICSSIM [24], a framework to build ICS security testbeds. Multiple technologies were used, including Docker, Memcached, Python and GNS3. Docker

allows running containers within GNS3, creating a flexible simulation environment. A file-based database (SQLite4) ensured the connection between Docker containers as the hardware layer connection is not an IP-aware communication. Physical process simulators and PLCs are configured to access sensor and actuator values accordingly. For GNS3 simulations, a 'Memcached' service is provided due to the limitation of GNS3, not allowing the share of file-based database between nodes. The communication between devices uses the ModbusTCP protocol, more precisely the pyModbusTCP library. Additionally, support is provided for internal Python scripting and third-party software like Matlab or Simulink for HIL software simulation through different connectors.

GRFICS [33] is an open-source graphical ICS simulation tool, based on the Tennessee Eastman process. It's primarily designed for educational purposes, offering pre-defined functions and simulated ICS devices. OpenPLC [5] simulates the PLCs, while the HMI Virtual Machine emulates an HMI using AdvancedHMI [7] software. The testbed supports various pre-defined attacks, including MitM, Command Injection, False Data Injection, PLC Reprogramming (e.g., Stuxnet), Malicious Binary Payload (e.g., TRITON), and Common IT attacks (e.g., password cracking, buffer overflow). Once the attacks are launched, the interface allows monitoring attack consequences, logging process information, and how much cost is wasted through the purge. Additionally, the testbed supports Snort detector [61] installation and customization with new rules. Communication within the testbed is based on Modbus protocols.

VISCORT [28] [29] is an extension of the GRFICS_v2 project cited above. This research rebuilt VMs with lightweight container hypervisor(LXD) and Kernel-based Virtual Machine(KVM) to consume fewer resources, upgrade numerous components of the simulation, and add an attacker node to perform common attacks on ICS.

The master thesis of Paolo Trungadi [66] propose an implementation of the SWAT [37] process with MiniCPS and GNS3. He also performed some common attacks such as ARP poisoning, DoS, and analysed the damage caused on the developed testbeds.

The master thesis of Masset Bertrand and Olivier Taburiaux proposes a fully virtualized testbed [45] using Mininet to emulate an SDN network with POX controller, a modified version of OpenPLC_v2 as PLC and MTU. The SCADA system is implemented with PySCADA and a CloudFirewall acts as a firewall. They also modified MiniNAM to support a visualization of the ICS components. MineEdit is adapted to provide a visual way to create the ICS system, with scripts to generate some examples. Some attacks were also performed on it. However, there was no physical system simulation, and it only supported ModbusTCP.

3.2 Virtual PLC

Most of the production lines are still controlled by hardware PLCs, however, with the market continuously pushing towards Industry 4.0, needing lower operational costs and developing more agile solutions, virtualization of PLC is a promising solution.

Mikael Peltonen thesis [57] explored different virtualization approaches including Hypervisor-based virtualisation(KVM/QEMU) and Container-based virtualization(Docker). Coupled with Node-RED as a control system and OpenPLC as virtual PLCs, using Modbus TCP as communication protocol, he gives a comparison of the two solutions. The results show that KVM solutions have too high latencies, while Docker latencies are acceptable. However, in both cases, the max latencies are really high compared to those without virtualization,

the author did not find out the reason that causes this high max latencies.

Michael et al. [38] in 2020 confirmed that docker is more suitable for virtualization of PLC. They also tested the impact that different docker drivers have on latencies and found out that macvlan network driver is best suited for applications that have high latency requirements and require L2 and L3 communications, while the overlay driver should be used when a higher level of security needs to be achieved and the application can handle RTTs that are 20% higher than those of other configurations.

A recent research on the commercial Siemens vPLC CPU1582u(equivalent to S7-1500) by Massimiliano et al. [34] conclude with similar results, however this time, they find that the reason for that high latency is due to the IP stack implementation.

Another commonly cited virtualization tool is Jailhouse [19], a Linux-based partitioning hypervisor developed by Siemens. Instead of virtualizing some components, it only partitioned the hardware into cells that can be attributed to different virtual PLCs, this solution is even more lightweight than Docker, but jailhouse offers way fewer functionalities than other VM/Containers approaches.

3.3 Honeypot

In the pursuit of enhancing Industrial Control System (ICS) security, honeypots have emerged as increasingly utilized tools. Honeypots represent either physical or virtual systems designed to emulate real devices within a network, luring attackers into thinking that it is an actual real system. These systems serve dual purposes: aiding researchers in understanding new threats and acting as indicators of malicious activities on the network. Within the realm of honeypots, three classifications exist: low-interaction, high-interaction, and hybrid. Low-interaction honeypots mimic the operating system and basic network services, limiting interaction with attackers or malware. For example, a honeypot simulating a Cisco router may appear authentic but prevents attackers from successfully logging in. Conversely, high-interaction honeypots allow attackers unrestricted access, they need to resemble as much as possible like the actual device. Despite their effectiveness, high-interaction honeypots pose significant challenges, such as the need for complete rebuilding if compromised, and increased risk due to providing genuine environments for attackers. The hybrid category seeks to mitigate these drawbacks by combining elements of both low and high interaction, offering a balanced approach to honeypot deployment in ICS security.

Ondrej et al. paper [58] discusses multiple implementations of physical and virtual testbed, highlighting limitations in flexibility, size, and simulation scope in chemical processes for physical testbeds. Among the virtual testbeds, the Honeypot Testbed stands out. It comprises MiniCPS for physical process simulation, MiniNet for network topology simulation, HonSSH for recording bash activity, and Docker for containerization to isolate attackers. The honeypot emulates protocol services connected to a virtual private server through MiniNet, with SSH port 22 exposed to the Internet. HonSSH monitors incoming traffic, forwarding attackers to a Docker container where MiniCPS processes run. Modbus and Ethernet/IP protocol communication is facilitated by pyModbus and CPPPO libraries. The testbed, deployed on a public IP address, recorded 1786 connection attempts, revealing repeated attempts from various IP addresses and additional activity from ten connections. Despite its effectiveness, the testbed's flexibility is limited by the number of communication protocols and simulated devices. Extending the testbed requires implementing new device images to maintain realism and security against potential attackers.

HoneyPLC by Efren et al. [44] is introduced as an extensible, high-interaction, and malware-collecting honeypot tailored for Industrial Control Systems (ICS). Unlike existing honeypots for PLCs, HoneyPLC provides advanced protocol simulations, such as TCP/IP, S7comm, HTTP, and SNMP, enabling interactions comparable to real PLCs. It effectively camouflages itself as a real device, identified by various reconnaissance tools with high confidence. Deployed on Amazon AWS, HoneyPLC records significant interactions over the Internet, demonstrating its ability to engage and deceive attackers while collecting valuable data. However, there are limitations to consider: it lacks support for modelling physical interactions of PLCs, and testing against sophisticated malware like Stuxnet is challenging due to difficulties in injecting Ladder Logic code.

3.4 Intrusion detection system

As industrial systems share common vulnerabilities with IT systems, implementing IT solutions into Industrial systems could be a good solution to palliate risk due to the introduction of those systems, however, industrial systems tend to have more constraints due to the interaction between cyber and physical systems. Hence there is a need to tailor these systems to be more performant on ICS.

Some commonly used IDS are Snort [61] an open-source, network-based IDS, which has a rule language for specifying misuse and attack signatures. Suricata [3] has been created to be an open-source enhancement of Snort. It has native multithreaded operations to handle a larger network volume than Snort. Zeek [56] (previously known as Bro) It uses detection methods based on signatures and anomalies. Its scripting language allows users to define the analysis and detection policies at a much higher level of abstraction.

The paper of Monzer et al. [51] presented a solution to facilitate the design of a model-based IDS for ICSs and by using this model, to create a rules generator that converts the input model into IDS rules. This solution solves the problem of creating and maintaining consistent rules for the anomaly-based process-aware IDS.

Aboah Boateng et al. [1] propose a one-class support vector machine, one-class neural network interconnected in a feed-forward manner, and isolation forest approaches for verifying PLC process integrity by monitoring PLC memory addresses. A new histogram-based approach is introduced to visualize anomaly detection algorithm performance and prediction confidence. However, this approach is limited to anomaly-detection algorithms with signed output results and the choice of the parameters of different ML techniques.

Chapter 4
ICS Simulation

This chapter presents in depth the decisions we made to build the ICS simulation. It will discuss the design choices taken, the tools used and the modification performed to support our needs.

4.1 Design

There are a lot of concerns when creating our ICS simulation. First, We needed to decide which type of simulation we wanted. In our case, to have an as general as possible simulation, we choose to go on a fully virtualized simulation. In this way, we are not dependent on hardware. In this way, we won't be limited by the price of such hardware, and a full software solution can offer wat more flexibility in terms of components, size of simulation, and portability. From the related works on virtual solutions, most projects use a time-step mechanism to synchronize the whole simulation. Although this is an interesting approach, after some debates, we decided not to use synchronisation mechanism.

As a new start on an ICS simulation, we wanted to try a new approach to the subject and use this thesis as a start for future projects. A real-time simulation was thus decided, it is a much simpler approach that can benefit from tools already available. The drawback is the lack of accuracy compared to real systems, due to limited hardware in heavy simulation with multiple simultaneous running processes. ICSs are composed of multiple devices that communicate to each other, we chose to use realistic networks that could carry real communications compared to solutions that use fake messages to simulate interactions. This is important to support existing tools, variable network topologies and future project development. We decided to create a configuration tool that uses device templates to generate ICS instances instead of focusing on a single specific example. This configuration file will follow an easy-to-understand workflow that can be adapted to many different scenarios and support easy modification. We considered using existing tools and modified them if needed to support our needs so we could avoid duplicate works and go as far as possible on the subject itself. Open-source projects are the best choices in this case. Finally, an automatic installation and configuration of the project in a virtual machine is needed for easy setup. This Virtual machine can also be pre-configured and thus be portable to anyone who wants to make a try via the vagrant config file.

4.2 Tools

In this section, we will briefly introduce tools we used to reach our objectives, to have a clear overview of all tools before detailing them in the following sections. First of all, we had to select a tool to create virtual networks. From previous experience and ICS simulation projects, Mininet [49]and IPMininet [18], its non-Software Defined Network (SDN) versions were chosen. IPMininet is easy to use and allows the user to create custom topologies easily by writing simple Python scripts. The hosts created by IPMininet should also be able to run the different components of the ICS simulation. However, experimentation revealed

that the PLCs' flask server will be frozen after seconds of non-interaction. Thus we changed direction and used Docker [26] network. For the simulation of PLCs, after a comparison of different PLC simulation solutions, we selected OpenPLC [54], which is an open-source Programmable Logic Controller that supports popular SCADA protocols such as Modbus, DNP3 and EtherNet/IP. OpenPLC can play the role of a PLC and MTU which is really convenient in our case as it kills two birds with one stone. Even if OpenPLC can be used to monitor sensors and PLCs data, having a central view of the ICS simulation with a SCADA system is necessary, thus we used SCADALTS [21](SCADABR) [22], an open source SCADA system to fulfil that role. All these tools are used in a Linux environment, ubuntu 20.04 LTS to be more precise and are mostly programmed in Python or Bash. For portability, everything is automatically installed and configured in a virtual image in VirtualBox via a Vagrant script.

4.3 Architecture overview

To be as realistic as possible, we try to follow the PURDUE model, which divided the organization into 5 levels:

- Level 5: Internet DMZ, where an organization's corporate network is separated from the internet
- Level 4: The organization's corporate network or enterprise zone.
- Level 3: Separated by a DMZ, the manufacturing zone or the main control centre, which is the start level of the ICS.
- Level 2: Regrouping a lot of Local HMI, and control points
- Level 1: Field controllers
- Level 0: Field I/O devices, Bus, Sensors

The figure 4.1 shows the architecture of the testbed we have created. The system is separated into 3 levels, the physical simulation, written in Python, is equivalent to levels 0 and 1 of the Purdue model. The OpenPLCs instances will be considered as level 2, as they not only control a bunch of devices of the physical simulation but can also be used as an MTU or local HMI to monitor and control the physical simulation. Then the Historian (ScadaBR/ScadaLTS) will be in the level 3. All of these nodes are containerized and interconnected with each other as they were put in the same user-defined docker network.

The following sections will describe in detail each node of the architecture, including the network simulation done through Docker networking, and a dedicated chapter will be given on the physical simulation.

4.4 PLC simulation

The primary component required for developing an ICS simulation is the PLC. Extensive efforts were dedicated to identifying and evaluating available solutions, as well as understanding their limitations. The objective was to identify a free, Linux-compatible software capable of simulating a standard PLC, supporting communication protocols like Modbus/TCP, being general enough to support different behaviours and being part of a

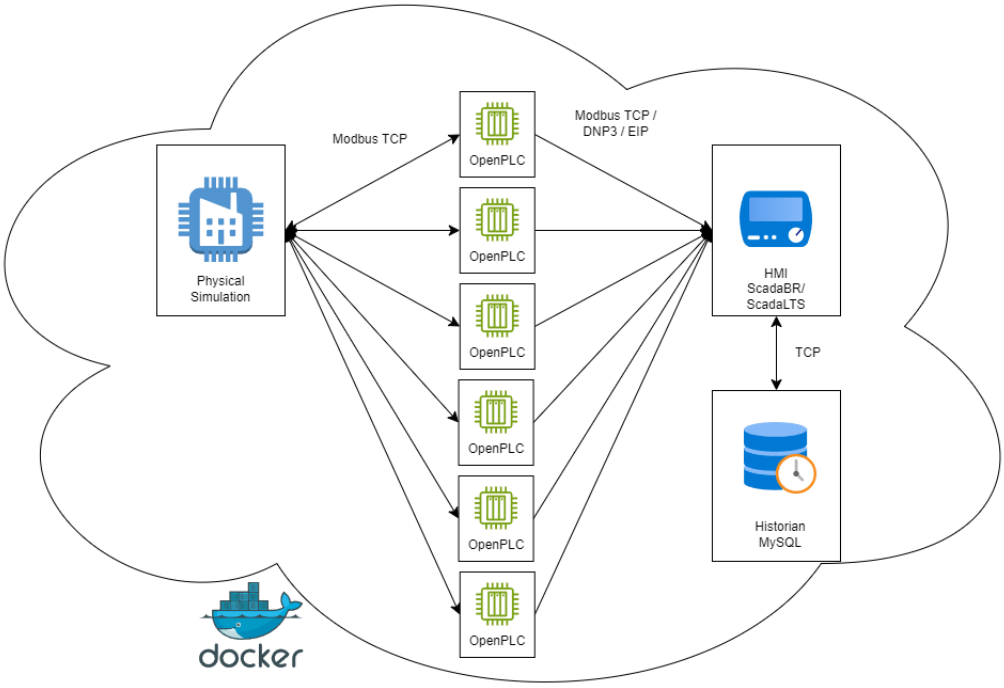

Figure 4.1: Overview of the testbed architecture

bigger simulation to represent an ICS. The preference was for an open-source project to facilitate modifications and circumvent restrictions typically encountered with closed-source solutions.

From the previous project [45] on ICS and further research, commercial solutions such as RSLogix by Rockwell Automation and S7-PLCSIM by Siemens were directly out of our view as these solutions are likely to be either non-free or offer limited trial versions, exclusive to Windows operating systems, closed-source, and lack support for external communications. Additionally, Matlab/Simulink, although capable of directly programming PLCs for testing and code generation, was deemed overly specific and resource-intensive for inclusion in a comprehensive ICS simulation.

Subsequently, several smaller projects including ClassicLader, pyModSlave, and EasyModbus were investigated but did not provide us with all the needed functionality. Eventually, a suitable project meeting the specified criteria was identified.

OpenPLC [54] is a project aimed at providing a standardized open-source PLC. Started in 2014 [5], the project is now in its third version (OpenPLC_v3) and is still under development. It can run on open-source hardware like Raspberry and Arduino or more commercial ones like UniPi. Recently they also released an *Underwriter Laboratories* certified (UL-listed) industrial Raspberry Pi (Figure 4.2), as a certified PLC, it can be used in real-world ICS. It is also possible to run it as a virtual PLC in complete software mode on Linux and Windows. We were looking for a virtual PLC to avoid investing money in hardware and run a high number of instances simultaneously to mimic a typical ICS composition. The core of OpenPLC stays the same between platforms with only minor compilation changes. What needs to be adapted is the code to access the hardware layer, more specifically the code to access the I/O of the platform (example for a Raspberry in Figure 4.3). It can

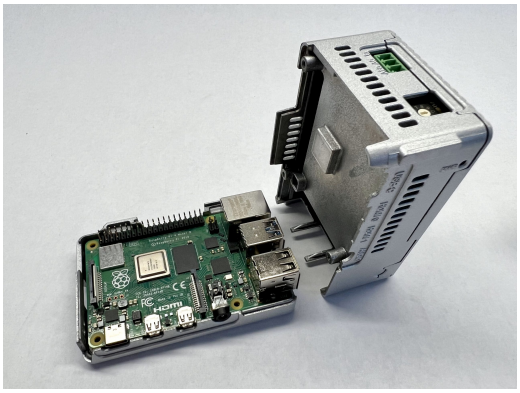

Figure 4.2: SL-RP4 [8]

Figure 4.3: Raspberry pinout

thus be ported to new platforms without too much effort.

The project adheres to the standard IEC 61131-3, which means that it supports the five programming languages discussed in Section 2.1. OpenPLC Editor is recommended as an editor supporting programming in these languages. It is an open-source Python software derived from the Beremiz project [10], aimed at providing a comprehensive development environment for automation. OpenPLC Editor is recommended for its user-friendly interface, the ability to save projects in the recognized standard PLCOpen XML format, and the option to export any of the languages into Structured Text (ST) logic code. This latter feature is particularly useful as OpenPLC is capable of compiling and running ST logic, enabling simulation of real PLC operations using the same logic.

The project also supports variant communications protocols such as Modbus/TCP, DNP3 through open-source libraries libmodbus and opendnp3, respectively It also support EtherNet/IP through custom implementation. This enables communication with a running instance of OpenPLC to read current values from its memory and send write requests to modify them. If the modified values are integrated into the logic, they can influence its execution to trigger new actions.

The figure 4.4 presents an example of a ladder logic in OpenPLC Editor. The variables are declared on the top and the logic can be drawn on the bottom. Each variable has a type and a specific memory location in OpenPLC. In the example, the variables are the 3

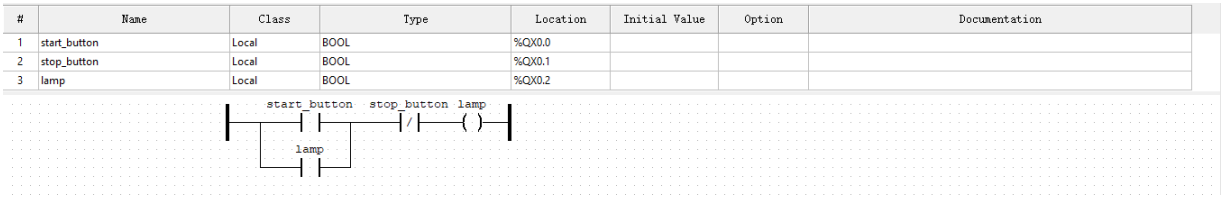

Figure 4.4: Ladder logic example (presented in section2.1) inside OpenPLC Editor

Data Type	Size	Access	PLC Adress	Address Range
Discrete Input Contact	1 bit	Read-only	%IX0.0 - %IX99.7	0 - 799
Discrete Output Coil	1 bit	Read-write	%QX0.0 - %QX99.7	0 - 799
Input Register(include slave)	16 bits	Read-only	%IW0 - %IW99	0 - 1023
Holding Register(include slave)	16 bits	Read-write	%QW - %QW99	0 - 1023

Table 4.1: OpenPLC I/O - Modbus binding

first cols of the PLC mapping. The memory mapping of OpenPLC are presented in Table 4.1,4.2, 4.3. A Modbus master (OpenPLC instance) can be used as an input extension of another OpenPLC instance. Table 4.4 details the naming convention of PLC addresses.

OpenPLC also provides an extended memory address space for larger registers.

OpenPLC is mostly composed of three languages, the core of OpenPLC is written in C++, the web GUI is a Python flask web server. There is also a Python SubModule (PSM) that acts as a driver for OpenPLC, and finally, some bash scripts that allow the installation of OpenPLC and other functionalities. The web GUI is started by the flask server that creates an interface accessible from the web browser (Figure 4.5). It is recommended to use the web GUI to interact with the start/stop of PLC, check logs, monitor PLC I/O state, change the PLC driver, add/remove slaves PLC, and upload new ST logic that must be compiled and started.

In the following subsections, we will discuss each tab and what we can do in them.

4.4.1 PLC program

As shown in Figure 4.6, here is the page where we can upload, choose and compile our PLC logic. OpenPLC is provided with a blank program that is here to assist the launch of OpenPLC. The ladder logic that we will use, as shown in Figure 4.7, only transfers the data from input contact/register to the output coil/register without any specific processing. The choice of this simple ladder logic is driven by our aim at simplicity and straightforwardness to learn. A user that does not want to bother with IEC 61131-3 PLC logic, can use the ICS simulation configuration to generate those PLC logics. The generation of that PLC logic will be further discussed in the configuration5.1.3 subsection. Of course, we still need to control our PLCs. To do so, we will write the control logic directly in the Hardware Driver of OpenPLC. One characteristic of OpenPLC_V3 is the introduction of the PSM driver. As Python has grown as one of the most used languages by scientists and students,

Data Type	Size	Access	PLC Adress	Address Range
Slave Discrete Input Contact	1 bit	Read-only	%IX100.0 - %IX199.7	800 - 1599
Slave Discrete Output Coil	1 bit	Read-write	%QX100.0 - %QX199.7	800 - 1599

Table 4.2: OpenPLC slave I/O - Modbus binding

Data Type	Size	Access	PLC Adress	Address Range
Holding Registers	16 bits	Read-write	%MW0 - %MW1023	1024 - 2047
Holding Registers	32 bits	Read-write	%MD0 - %MD1023	2048 - 4095
Holding Registers	64 bits	Read-write	%ML0 - %ML1023	4096 - 8192

Table 4.3: OpenPLC extended I/O - Modbus binding

Prefix	Description	Prefix	Description
I	Input	Q	Output
M	Memory location	X	Bit
B	Byte (8 bits)	W	Word (16 bits)
D	Double word (32 bits)	L	Long (64 bits)

Table 4.4: OpenPLC Naming convention

we think it would be easier to learn for people who might not be interested in the inner logic of PLCs. This Python driver is easy to use and can be used to program the logic control of our PLC. A bash script also automatically uploads the generated ladder logic to the according container/PLC.

4.4.2 Hardware Layers

OpenPLC offers several hardware drivers for interfacing with different types of hardware or software as shown in Figure4.8

Most of them are hardware drivers, they are used to map OpenPLC contacts/coils/registers to the board input/output/memory. A particular one is the Simulink driver. This driver through another small program called SimLink, can communicate with a physical process simulated in Matlab/Simulink. The Matlab simulation provides components that send and receive data via UDP, and as shown in Figure 4.9, via UDP to the Simulink Driver which uses a shared Memory with the OpenPLC instance. The Matlab/Simulink solution can provide much higher accuracy on the physical simulation but at a high cost in resource consumption. Thus we decided to not use this driver, and opt for PSM.

the PSM driver stands out due to its simplicity and versatility. Python is widely used by researchers and students, making it an accessible choice for programming PLCs. Writing PLC logic in the PSM driver is straightforward.

The PSM driver is designed with simplicity and efficiency in mind. It consists of three main functions: hardware initialization, data input and data output.

Firstly, the hardware initialization function can be used to initialize the Modbus connection with the physical simulation, initialize connection status, and initialize some sensor values.

Following initialization, the PSM driver enters an infinite processing loop. It will first get input values from the physical simulation. From then, in the update_inputs function, based on the logic written, write the processed data in the according input contact/register.

The ladder logic executes the programmed logic. In our case, transfer the input to the output coils/registers. Once the processing is complete, the output values are read from the ladder logic and made available to the PSM update_outputs function. Where output values can be read. Finally, the output values are available to the Modbus protocol to be transmitted to external devices or systems. This completes the cycle, ensuring seamless communication between the PLC logic and external hardware.

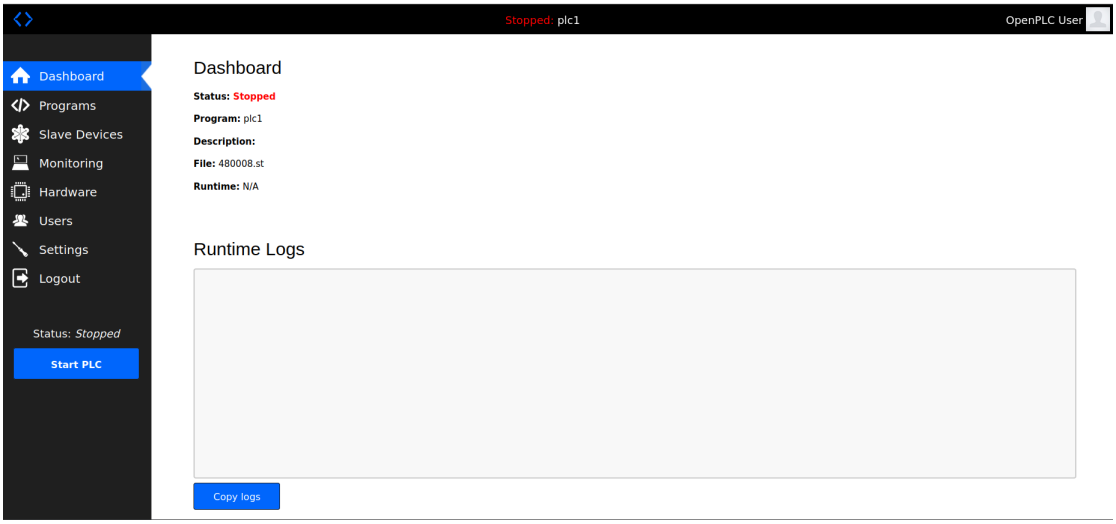

Figure 4.5: openPLC main page

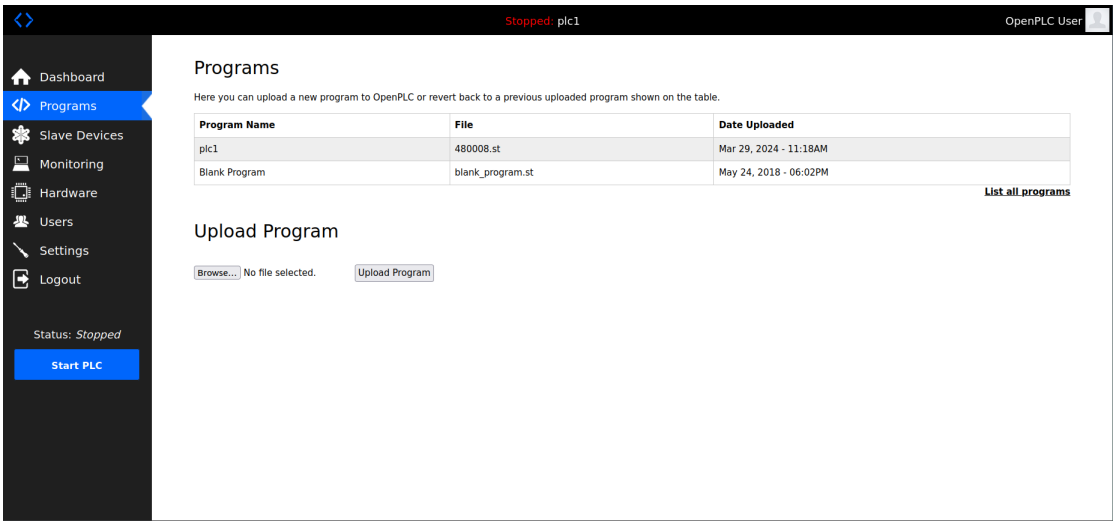

Figure 4.6: OpenPLC program

It has to be noted that OpenPLC works in a "consuming" manner, meaning that after each loop, the input contact/register value will be consumed, and the output coil/register will be overwritten. Another worth mentioning is that the PSM driver does not have access to the OpenPLC memory, meaning that we are limited to writing to input contacts and registers. A workaround will be to write proper ladder logic utilising Arithmetic blocks to compute a value from two registers and output the result to a Memory variable.

Overall, the PSM hardware driver in OpenPLC provides a user-friendly and efficient solution for interfacing with various hardware components, making it an ideal choice for both beginners and experienced users.

4.4.3 slave

OpenPLC can attach Modbus slave devices to the OpenPLC Runtime to expand the number of I/O points. This is particularly useful for systems that don't have any I/O points at all, like OpenPLC Runtime running on desktop computers or servers for example. The OpenPLC Runtime supports slave devices using either Modbus/TCP (network) or Mod-

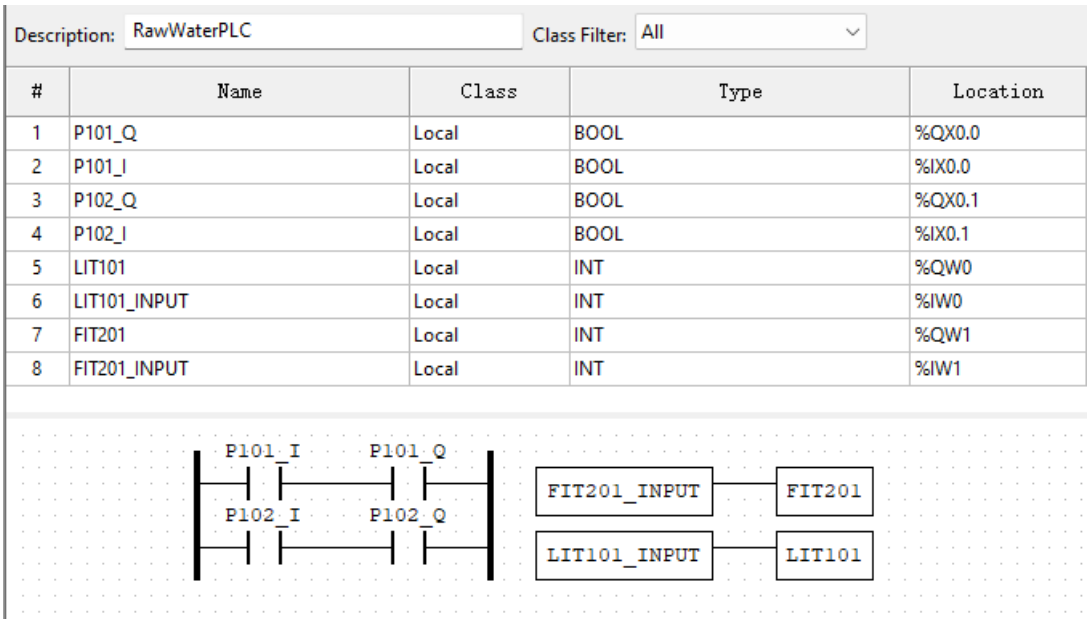

Figure 4.7: Example of PLC ladder logic

bus/RTU (serial). Devices with wireless capabilities that can transmit Modbus packets over TCP/IP are also supported. To add a new slave device, click on "Add New Devices", select "Generic Modbus TCP Device" and define the address of the first register and the range, which means how many registers after the first one you want to be read/written. These inputs, coils and registers will then be mapped to OpenPLC internal variables, located at %IX100.0, %QX100.0, and so on...

However, the capability of PSM is limited when we are working on PSM, the PSM driver does not allow read/write to these slave addresses as the values should already be in the memory, and not being processed by the PSM driver. As a workaround, our physical simulation implements a Modbus slave, and each OpenPLC instance, in the PSM driver, will initialize a Modbus master. Thus the PSM driver will be able to read and write data to the Modbus slave device (physical simulation). More details concerning the Modbus slave will be given in the physical simulation section 4.7.

4.4.4 Monitoring

The OpenPLC monitoring page (Figure 4.11) allows users to observe and interact with the PLC.

This page displays the different values in the PLC's memory values, the refresh time rate can be adjusted, forcing the values of the PLC to manually control the PLC.

4.4.5 Settings

On the OpenPLC settings page, you can enable DNP3, EtherNet/IP server. Using this server, we can make connections with SCADA software, permitting us to have a centralized monitoring system. The Persistent Storage Thread should allow retaining memory variables when restarting the PLC device, however, this function is buggy. It cannot be enabled, but some people make a workaround [36]. Enabling OpenPLC RUN mode will restrict the edition of the PLC program.

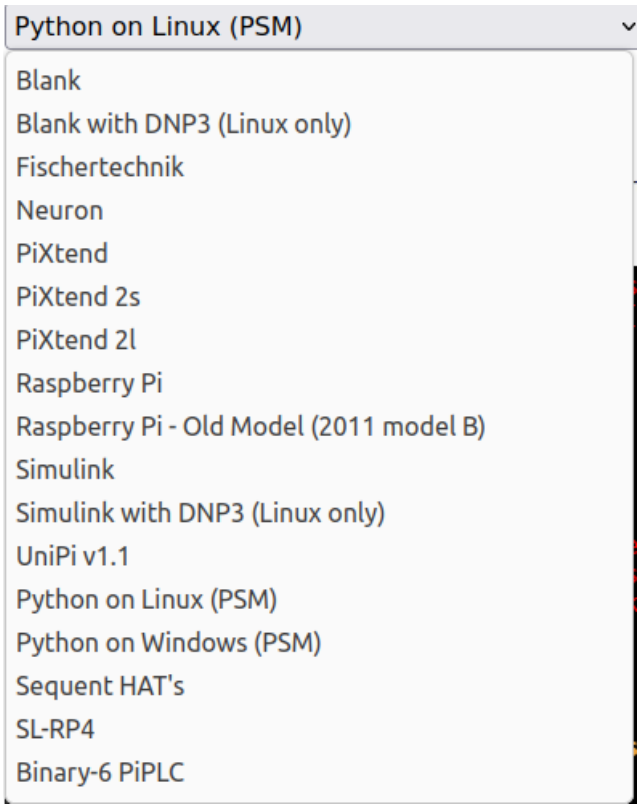

Figure 4.8: OpenPLC drivers

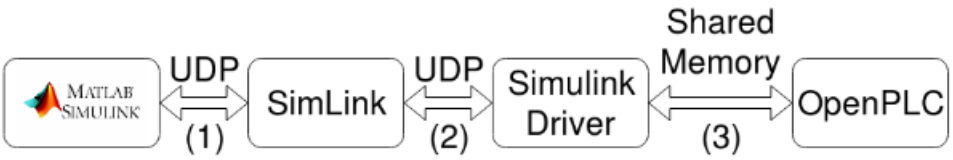

Figure 4.9: Interactions between OpenPLC and Matlab/Simulink [45]

4.5 Multiple instances

Running multiple instances of OpenPLC on a unique computer is quite complex as it needs to:

1. Install multiple OpenPLC instances (OpenPLC is installed at sudo level, with root user).
2. Change the webserver port of each OpenPLC instance so that there is no socket being used by another one.
3. OpenPLC python version cannot conflict with other Python interpreter (physical node)

To resolve this problem, many solutions have been discussed on the OpenPLC forum. Well-known KVM solutions like Virtual Box are too resource-consuming, and even if our ICS simulation does not require high real-time requirements, we want it to be more lightweight.

Another solution is to use Jailhouse, a solution developed by Siemens. Jailhouse is a lightweight, small and simple open-source Linux-friendly hypervisor for real-time and

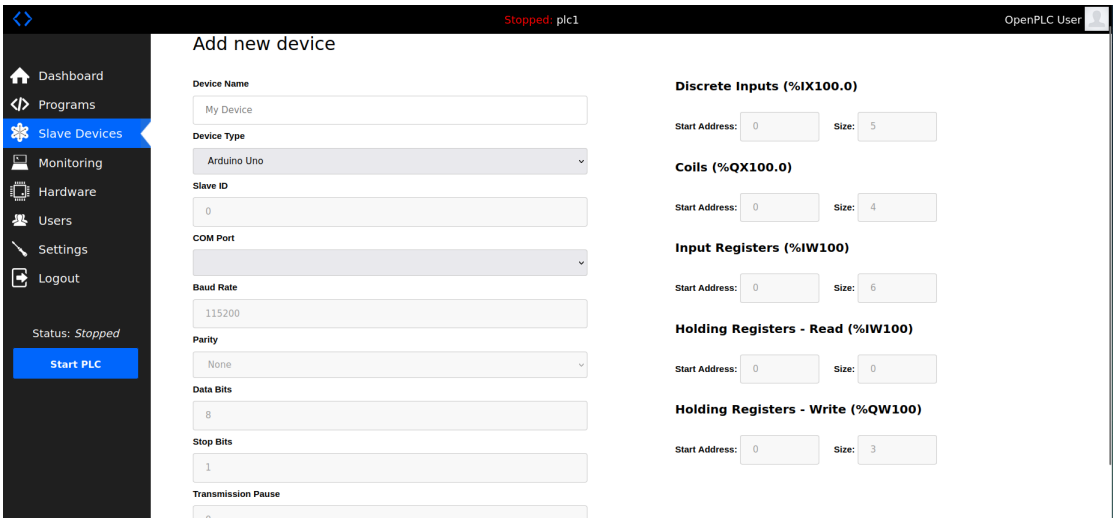

Figure 4.10: OpenPLC slave devices

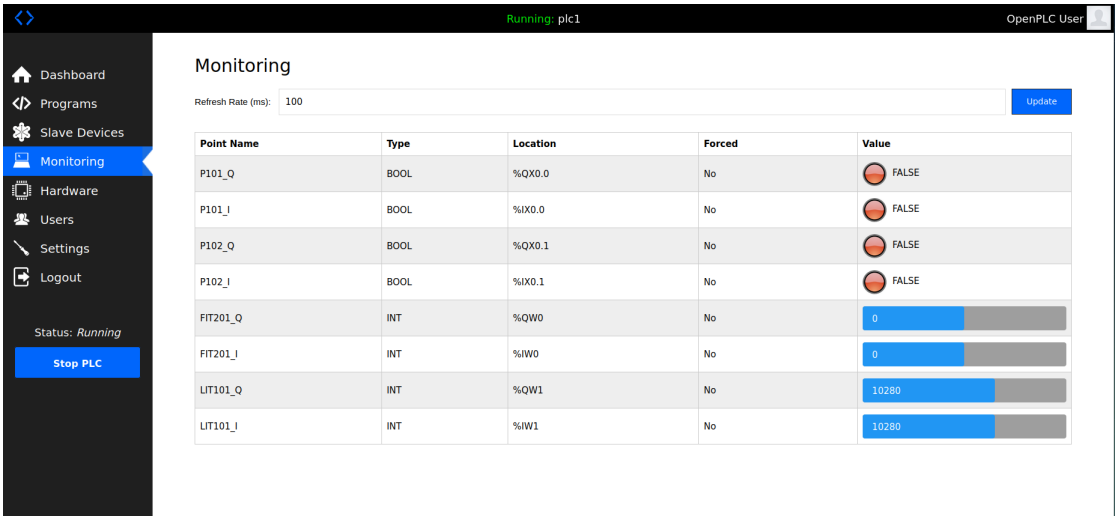

Figure 4.11: OpenPLC monitoring page

certifiable workloads. From the technical point of view, Jailhouse is a static partitioning hypervisor that runs bare metal but cooperates closely with Linux. It just splits your hardware into isolated compartments called "cells" that are wholly dedicated to guest software programs called "inmates". One of these cells runs the Linux OS and is known as the "root cell". Other cells borrow CPUs and devices from the root cell as they are created. In this way, each PLC will have its own dedicated CPU and memory spaces. However, due to a lack of documentation, and non-straightforward setup manner, we decided to opt for another solution.

Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Several reasons push us to use Docker container as an isolation solution:

- Open source

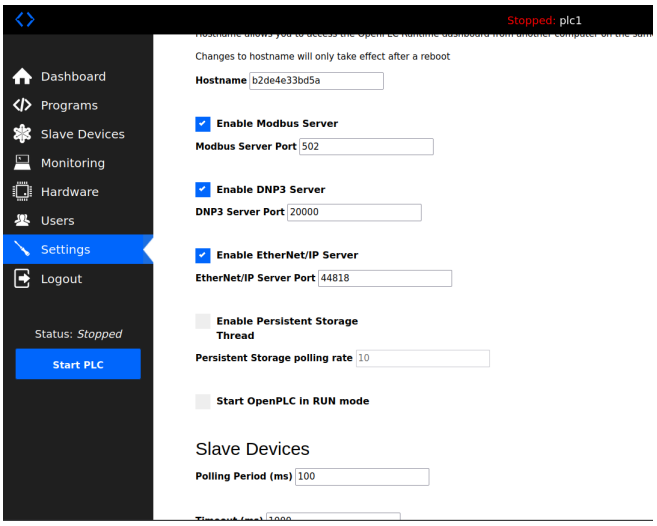

Figure 4.12: OpenPLC settings page

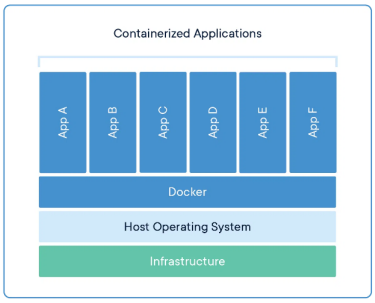

Figure 4.13: Docker Cotainers

- Well-known, well documented, easy to use
- Lightweight, container-based solutions overhead are acceptable for real-time applications.(Section 3.2)
- Networking capabilities, docker provides multiple drivers, and container network capabilities and allows future extension of the project with firewalls, DNS service, IPv6
- Well supported for the different ICS components (Python, OpenPLC, ScadaBR/ScadaLTS)

The deployment and launch of a docker container can be fully automatized via DockerFile and docker-compose config file. An example of a DockerFile 4.1, which will download a Ubuntu container, install Python, IProute, ping, all requirements for the physical simulation, and finally launch the physical simulation.

```
1 FROM ubuntu:20.04
2 COPY physic_simulation .
3 RUN apt-get update
4 RUN apt-get install -y python3
5 RUN apt-get install -y python3-pip
6 RUN apt-get install iproute2 iputils-ping -y
7 RUN pip3 install -r ./requirements.txt
8
9 ENTRYPOINT [ "python3" ]
```

```
10
11 CMD ["run.py","-c", "test.yml", "-v", "2", "-m", "proportional"]
```

Listing 4.1: DockerFile example

4.6 Network Emulation

Docker networking refers to the ability for containers to connect to and communicate with each other, or to non-Docker workloads. Containers have networking enabled by default, and they can make outgoing connections. A container has no information about what kind of network it's attached to, or whether their peers are also Docker workloads or not. A container only sees a network interface with an IP address, a gateway, a routing table, DNS services, and other networking details. Docker use the Container Networking Model (CNM) to manage networking for containers. The implementation of the CNM concept used by Docker is Libnetwork [20]. Then, with the help of drivers, various network topologies can be created.

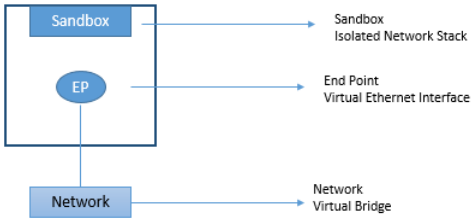

Figure 4.14: Docker CNM model

CNM is based on the following concepts: (Figure 4.14)

- Network sandbox: It is an isolated network stack, which includes Ethernet interface, DNS, Ports, routing tables.
- Endpoint: It is a virtual interface, which is used to provide network connection to make communication successful.
- Network: A network is a group of endpoints that allow containers to communicate with each other.

Figure 4.15 is an example of the container A with a sandbox, has an endpoint connected to network X, and the container B with a sandbox, has two endpoints, respectively connected to Network X and Y. The containers A and B which are connected to Network X will be able to communicate with each other, whereas the other endpoint in container B will not be able to communicate with the container A as they are not in the same network.

Here is a short description of different docker network drivers:

- Host: Containers share the network namespace with the Docker host. Offer high performance, but lack isolation between containers and the host.
- Bridge: Creates a virtual bridge that connects containers to the host network, each container is assigned an IP. Containers on the same bridge network can communicate with each other using their IP addresses.

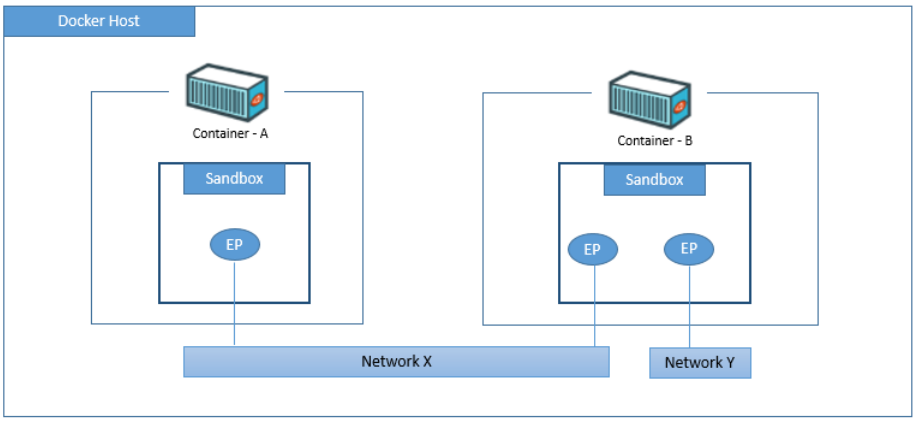

Figure 4.15: Example container architecture

- Overlay: Allows communication between containers across multiple Docker hosts. It's useful for deploying multi-host applications and services in a Docker Swarm cluster.

- MacVLAN: Assigns a unique MAC address to each container, making them appear as separate physical devices on the network. Containers can be directly connected to the physical network, allowing them to communicate without NAT (Network Address Translation). Making them the most performant ones over the different drivers. However, it requires configuration at the network level.

The creation of our network is done via a bash script that will automatically create the network, build the different containers, run them, and add them to the previously created network. 4.2

We first start to make sure the network does not already exist. If it does not exist, we create the network called "swat" with a fixed subnet, using a bridge driver. Then, with the predefined DockerFile, we can build a container named "plc1:oplcv3". Once the container is built, we can start it. The *–net swat* precise the network we want the container to join. *–ip 172.18.0.11* give a static IP address to the container in the network, *-d* for detached, to run this container in the background, *–rm* to tell the Docker Daemon to clean up the container and remove the file system after the container exits. *-p 10011:8080* will map port 10011 from our local machine to port 8080 inside our container.

```
1  sudo docker network inspect swat || sudo docker network create --subnet
    172.18.0.0/16 --driver bridge swat
2  sudo docker build -t plc1:oplcv3 .
3  sudo docker run --net swat --ip 172.18.0.11 -d --rm --privileged --name
    plc1 -p 10011:8080 plc1:oplcv3
```

Listing 4.2: Snippet code to create docker network and add containers to network

To simulate a more realistic network, we decided to install netem [48], a Linux package that provides network emulation functionality for testing protocols by emulating the properties of real-world networks. The queue discipline provides one or more network impairments to packets such as delay, loss, duplication, and packet corruption. Here is an example of possibles use of netem 4.3, more example can be found on the tc-netem manpage [48]:

```
1  docker exec plc1 tc qdisc add dev eth0 root netem delay 100ms # add 100
   ms latencies
```

```
2    docker exec plc1 tc qdisc change dev eth0 root netem delay 100ms 20ms
     distribution normal # This delays packets according to a normal
     distribution (Bell curve) over a range of 100ms +- 20ms.
3    docker exec plc1 tc qdisc change dev eth0 root netem loss 0.1% # This
     causes 1/10th of percent (i.e 1 out of 1000) packets to be randomly
     dropped.
```

Listing 4.3: Netem example commands

4.7 Physical simulation

To not be dependent on an existing ICS dataset, we prefer to have our own simulation that will be able to generate data. There are a lot of possible sectors that can be implemented in simulation, however, some of them are more critical as they can impact a lot of life in case of disruption, for example, smart grids which are electric networks that use digital technologies, sensors and software to better match supply and demand, which have a high real-time requirement and a huge number of connected devices. Chemical/Pharmaceutical sectors, which require high precision, Transportation, which have to deal with a lot of unexpected situations etc. In our case, we choose to simulate a water treatment process. Numerous reasons for this choice are :

- Procedural process: There are several well-defined steps to process raw water into the final product.
- Low real-time requirement: Water treatment process can support latencies of up to a second.
- Multiple variables to monitor: Outside of device states monitoring, the state and chemicals inside the water can be added to the monitoring.
- Existing physical testbed: We can inspire ourselves of existing testbed [37] to implement ours.
- Adaptation: We can easily adapt the water treatment process to another one, or even change the type of "data" to proceed in the treatment.

Once we have decided which physical process to simulate. We need to decide by which tools/languages it would be implemented. As said in the PLC program section 4.4.1, OpenPLC offers multiple drivers, and one of them is Simulink, which means, we could build our water treatment process in Matlab as these articles did it [35] [13] [24]. Having a complex Matlab simulation for the physical simulation can produce way more accurate generated data, and sustained by a great open source community, will reduce the development time of the sandbox. However, this solution is not lightweight, and would not suit our requirements. Other high-accuracy simulations are often done in Modelica, an object-oriented language to model cyber-physical systems. An example is the *WasteWater* implementation done by Dietmar, Martin and Lennart [25]. However, as Simulink, it is a heavy simulation. As OpenPLC have a PSM driver, we can use any language with at least a ModbusTCP implementation to communicate with OpenPLC instances. From this, many languages respond to these requirements but, 5 of them are more suitable: Python, C++, Java, Go and Rust. All of them have libraries that implement Modbus, DNP3, EtherNet/IP, OPC UA, and EtherCat.

All of these languages have their advantages and disadvantages. Rust claims to be more secure and is as fast as C/C++, but uses more memory, and has much fewer libraries than others. Go is reputed for its concurrency management, and will be great for multi-threaded applications, but in our case, as we physical simulation is a step-by-step application, multi-threaded implementation is not required. C++ can offer the best performance, but have a limited amount of libraries, which may not be suitable for future extension of the project. Java is the best in terms of scalability and has a consequent amount of libraries, but we choose Python for its ease of learning, wide range of libraries, popularity among scientists, data science and portability. An open-source Python package developed by AguaClara [16] is available for designing and performing research with high accuracy on AguaClara water treatment plants. However, as this project is pretty advanced, it does not offer customization as we would like due to its focus on AguaClara water treatment process. An interesting implementation of a water treatment simulation is done by Peter Maynard [46]. He split the simulation into 3 parts, devices, fluids, and sensors, the main loop will make devices work, adding/removing fluid in each device, and measure the result of each step through sensors, which will send data through ModbusTCP to an HMI which is a python Modbus Client. Where all data will be logged. We decided to follow a similar approach to this implementation.

For the replication of the physical simulation, multiple famous water treatment process exists, such as the Tennessee Eastman (TE) process 4.16. It simulates an actual chemical process widely used as a benchmark in test fault diagnosis and process control. The overall process consists of five operating units: reactor, condenser, vapour-liquid separator, recycle compressor and product stripper. Datasets of this simulated process are available and used as standard training and test data sets for applications such as diagnosis, classification fault detection, etc. A simplified version has been implemented by GRFICS [33] and VISCORT [29]. Although implementing this process is possible, and can produce a dataset that we can compare to the official one [14], however, the number of chemical products in it adds a lot of complexity in the implementation of reaction between those chemicals.

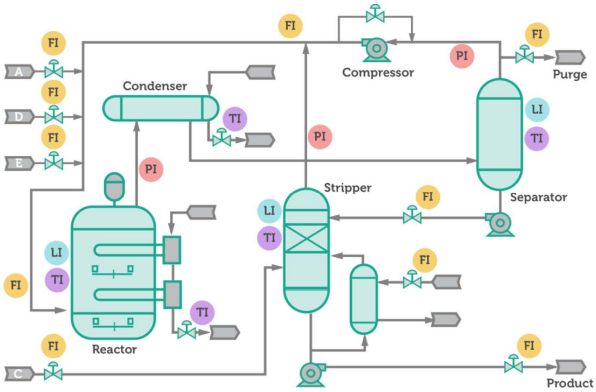

Figure 4.16: Tennessee Eastman process [14]

Another available water treatment process dataset is the one developed by the iTrust lab of Singapore [37]. The lab has developed its own simplified and smaller-scale water treatment testbed 4.17, and accomplished cyber attacks on it. Details of the physical devices, process logic, and connections between devices are also available. Only a few chemicals are present in the process and are removed through some devices such as an ultraviolet dechlorinator, and filters that remove deposits and big particles.

We will implement a slightly simplified version of the process. The way it is imple-

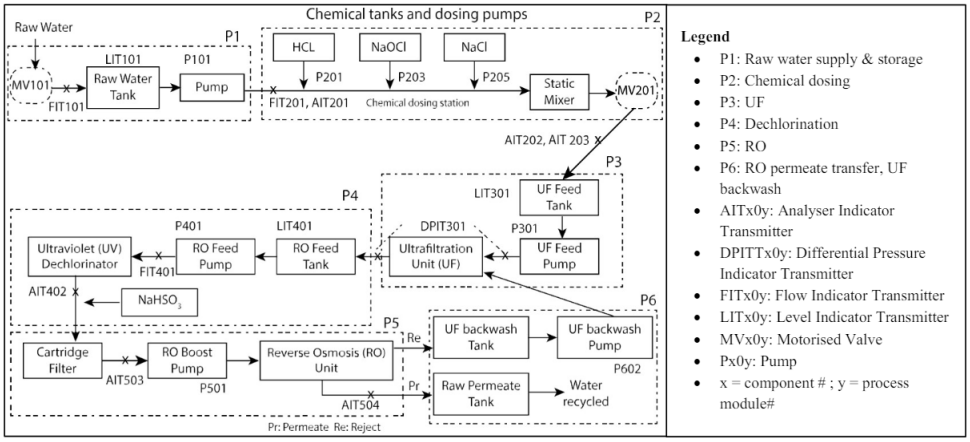

Figure 4.17: iTrust Lab Secure water treatment process(high level) [37]

mented is similar to the one by Peter Maynard [46]. We implemented devices such as tanks, reservoirs, pumps, filters, vessels, and valves. There are sensors such as state sensors to monitor and control device states(on/off, open/close), volume sensors to measure the amount of fluid present in a device and flow rate sensors. Only one kind of fluid is implemented by simplicity, to avoid managing complex chemical reactions. Finally, PLCs will read data from sensors and write it to Modbus databank, and communicate with OpenPLC instances that can write back to the Modbus databank, from which the PLCs will react and control devices (through state sensors).

Figure 4.18 is an overview of the simplified version of the SWAT process that we implemented as an example to validate our physical simulation, the different devices are codified as follows: the first letter indicates the device type, the first digit, at which stage the process is related, and the two last digits are indexes to differentiate the different devices.

The devices starting with the letter T are tanks and derivate devices (i.e. reservoir), P ones are pumps, and MV are valves. The UF-301 (an ultrafiltration membrane), and UV-401(UV dechlorinator) are implemented as a simple filter. RO (reverse osmosis) devices are implemented as a vessel.

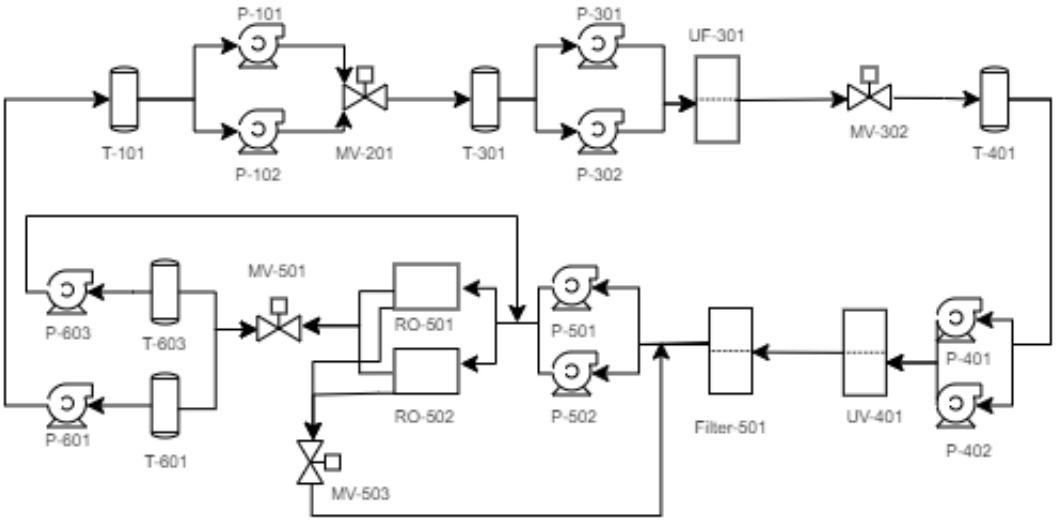

Figure 4.18: Implemented simplified version of SWAT

A dedicated chapter gives the details of the implementation of the physical simulation.

4.8 SCADA and HMI software

ScadaBR [22] and its long-time support version ScadaLTS [21], is an Open Source, web-based, multi-platform solution to build a SCADA system. Developed in Java, it can be run on many architectures (PC/Mac/Linux), its user interface runs from a standard web browser, and have well documented SOAP and REST API´s for custom integration. It supports features such as :

- Data Acquisition Engine (for many popular pushed and polled protocols)
- Watchlist - see datapoints updating in realtime
- Datasources and Datapoint Hierarchies - to organize data
- Graphical Views Builder (also known as HMI: Human-Machine Interface)
- Data Reports with charts
- User-based access with detailed permission settings
- Scripting Engine for on-the-fly value calculations, setpoints and commands issuing
- Multi-language with English, Portuguese, Spanish...

It also supports many industrial communication protocols such as Modbus TCP/IP, DNP3, IEC 101, OPC DA 2.0, and implemented other communication protocols such as HTTP listeners, an SQL connector etc.

Templates of dockerfile [12] and compose-file [17] to easily deploy ScadaBR and ScadaLTS into docker containers. Some small changes have been made to include the container directly in the network.

In the following subsections, we will detail ScadaBR(ScadaLTS) features and the settings we have made. If no specific mention was written for ScadaLTS, they use the same way of functioning.

4.8.1 Data sources

We will start by presenting the binding of data sources. We can access the ScadaBR webpage via the URL 'IP:8080/ScadaBR' or for ScadaLTS 'IP:8080/Scada-LTS', in our case, The IP address respectively are 172.18.0.9 and 172.18.0.8. After login(default credential is admin admin), we can enter the portal and go to the data sources panel. In the data sources panel, we will be able to define data sources (OpenPLC instances). For simplicity, we will detail how it was done with ModbusTCP.

The drop-down panel 4.19, lists all supported communication protocols of ScadaBR. To set sources using ModbusTCP/IP protocol, we will select the Modbus IP sources and click on the add button to add a new source.

Once in the configuration panel, we need to define the Modbus communication properties. In the properties panel, to have a more real-time update, the update period is changed to 1 second, the transport type is set to 'TCP with keep alive' to not disconnect after each fetch of data. Reducing a lot the overhead generated by multiple connections.

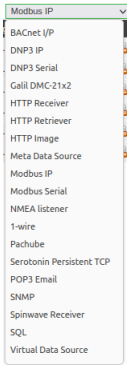

Figure 4.19: Source protocols

Finally sets the Host IP and Port of OpenPLC instances. In our case, OpenPLC IP varies from 172.18.0.11 to 172.18.0.16 (included), and the port is the default Modbus port 502. Once set, we can assess that we can read from the running OpenPLC instances by using the 'Modbus read data' panel. Select the type of register and the number of registers you want to read.

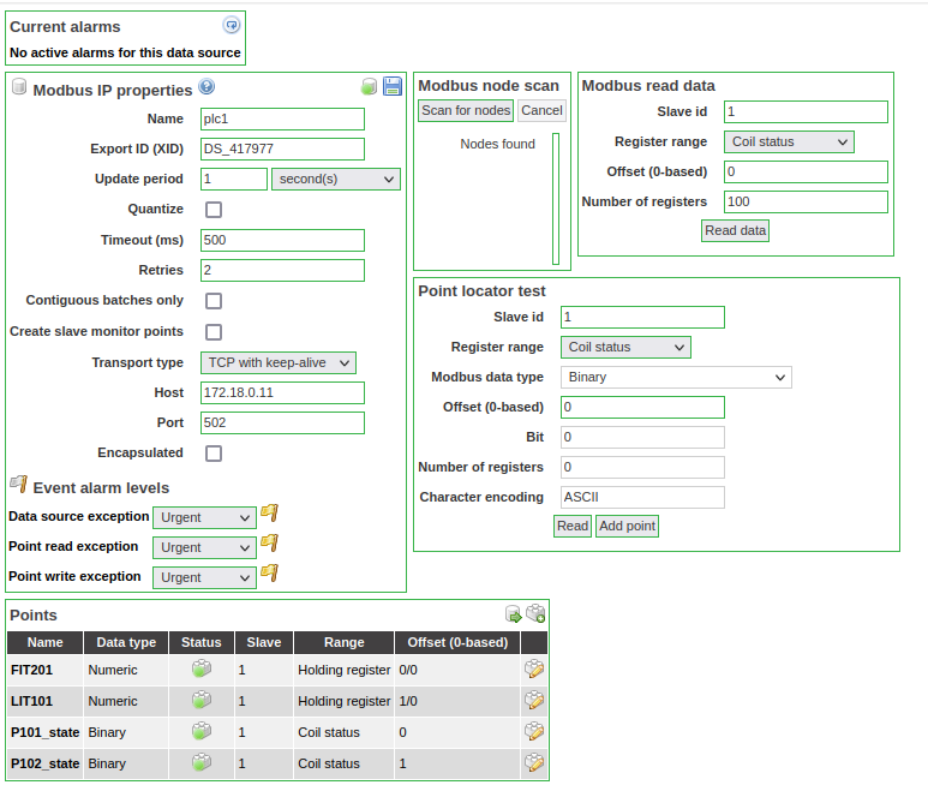

Figure 4.20: Setting Source protocols

Once we check that the Modbus proprieties are correct, we can save the setting, and we will be able to add data points. A data point is the binding of a coil/register raw value to a processed value in ScadaBR. A coil value will be registered as Binary, and a register integer value as 2 Byte Signed Integer. You will have to add an offset to read the correct coil/register. For example, the coil 'QX1.2' in OpenPLC, or X10 in the YAML configuration file, will have an offset of 10. Once the data point is set, you will have to enable the data points and the data sources so that ScadaBR will continuously fetch the

data. This process is done for all OpenPLC instances 4.21.

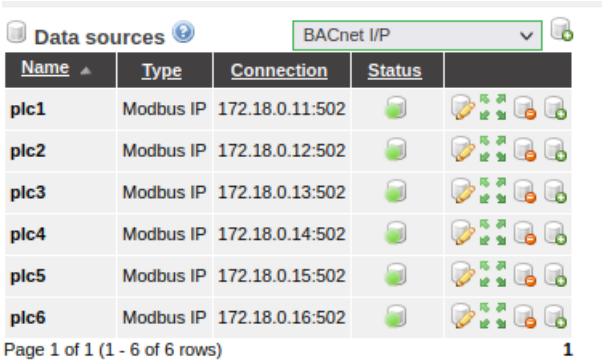

Figure 4.21: Configured data sources

With all these data points set, it is important to be able to organize these data points. Here comes the Hierarchy panel. We can create 'folders' which can regroup all these points, helping us to organize these points into a hierarchy.

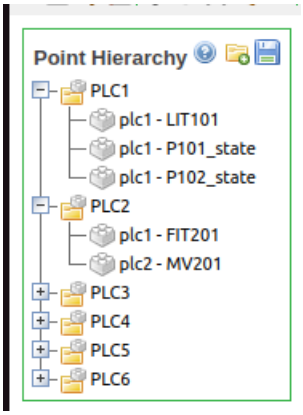

Figure 4.22: Set points hierarchy

4.8.2 WatchList and Graph

One of the main functionalities of an HMI is to be able to centralize, display and monitor all data received. Here comes the watchlist. In the watchlist 4.23, we add the data points that we have previously defined, and we can see them update at the frequency we have set on its data source.

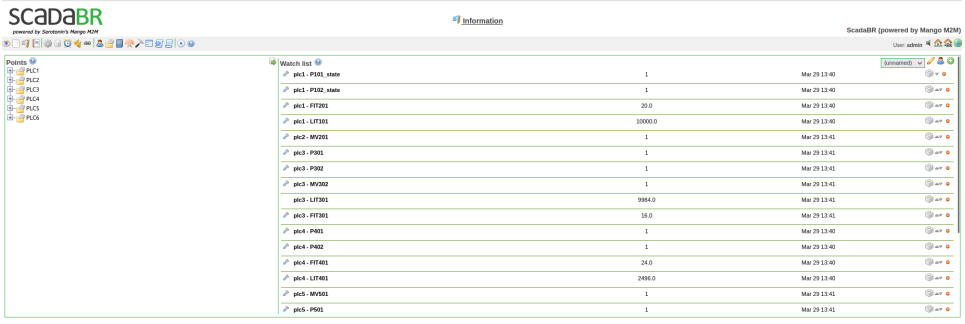

Figure 4.23: ScadaBR WatchList

When clicking on a data point of the watchlist, we can get a history of the data read from it, some statistical information and a chart drawn on these data. In this figure 4.24, as the white noise was deactivated, the sensor reported a linear decrease in fluid level, until its output valve was closed, so that input fluid filled the reservoir.

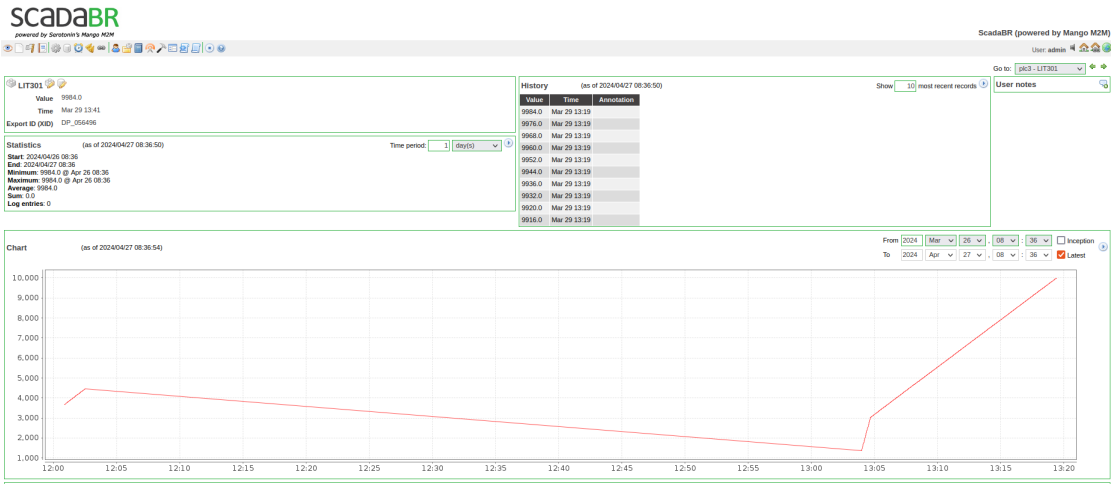

Figure 4.24: ScadaBR Historian graph

Another core functionality of an HMI is to be able to display graphically the process. In the graphical view 4.25, we can set up different images, forms, and javascript code to represent the data received and display them in a more Human-understandable manner. Due to a lack of time, we only represent this tank, bound to a data point, which is filled up based on the data fetched by the data point. Introduction of new images can be done by adding folders of images under '/opt/tomcat6/apache-tomcat-6.0.53/webapps/ScadaBR/graphics/' (Scada-LTS use a similar path, the path might be different depending on the tomcat version)

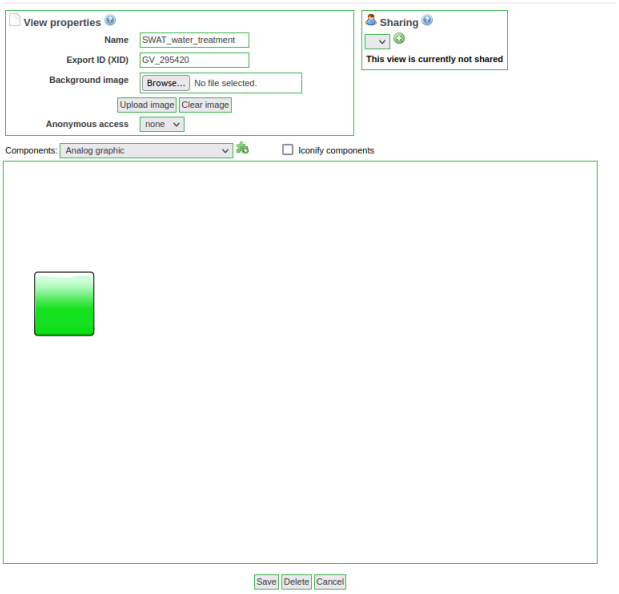

Figure 4.25: ScadaBR Graphical view

ScadaBR can export and import configuration by a simple JSON file, this file can be generated in the configuration panel.

Chapter 5
physical simulation

This chapter will describe the implementation details of the different components of the physical simulation, including the way of writing the configuration file to generate the full physical simulation, and the simplified version of the SWAT testbed we have developed as a sample testbed. How data are logs and some limitations of the current physical simulation.

5.1 Implementation of physical simulation

As stated in the Physic simulation section 4.7, for a lightweight and more flexible physic simulation, we decided to implement our simulation in Python.

The simulation is modular and can be divided into 3 main parts:

- Devices: the devices available(including Device/Actuator, PLC, Sensor).
- Fluid: the fluid passing through the different devices.
- Simulator: the main loop which initializes devices, controls the running process, acts as Modbus Server and logs numerous data.

Here is a list of used libraries:

- pyyaml: To generate Python object from yaml configuration file.
- sympy: To parse mathematics equation into executable python code if desired
- pymodbus: Complete Modbus protocol library, including serial, TCP, TLS and UDP versions of Modbus protocol.
- pymodbusTCP: This library provide an easy way to implement the Modbus TCP protocol
- numpy: Used to introduce white noise into the system.

The code follows PEP recommendations and uses PEP 484 - Type-Hints for all the functions and classes, making it easily maintainable and customizable.

The following sub-sections will give a more detailed description of the main loop, each component of the physical simulation, and the choice of each library.

5.1.1 Main Loop

Firstly, we will introduce the main loop of the physical simulation. When the physical simulation docker container is launched, it will launch the 'run.py' file, containing the entry of the physical simulation, this file defines accepted arguments of the simulation and the entry functions. Numerous arguments can be passed:

- –config(-c): The YAML configuration file to load
- –verbose(-v): Set verbosity level of the logs

- –math-parser(-m): Type of math expression parser
- –generate(-g): Generate openPLC ladder logic files

The YAML configuration file contains the general simulation settings(including the address and port needed for Modbus TCP), all Devices(Device/actuator, PLC, Sensor) configurations, the type of Fluid contained in tanks, and the connection between Devices. The structure details of this configuration file will be given in the configuration section 5.1.3.

In our simulation, to give the maximum liberty to researchers, we proposed different methods to compute the input/output of fluid of a device.

- 'proportional': When there are multiple inputs/outputs, the fluid passed by each connection should be even.
- 'sympy': By the use of the SymPy [23] library, we can parse mathematical expressions into Python executable code, and use it as the input/output function of devices.
- 'wolfram': SymPy is also able to parse wolfram expressions into Python executable code. Its use is similar to 'sympy'

The 'generate' argument can be used to generate ladder logic based on the configuration file. As our use of PLC is static (The PLC should only move input values to output), it is easy to generate the according ladder logic based on the configuration file. Details about how it was done will be given in the configuration section 5.1.3.

Once the settings of the Simulator are loaded, we set the logging system and instantiate the different devices with the help of PyYAML, the most popular third-party YAML library. It will parse the YAML document in a stream and produce the corresponding Python object. Once the different Devices are instantiated, we can launch the Modbus Server of the simulation, and initialize the Modbus databank. After the Server is running, we will wait for all PLCs to be connected to the server before launching the main loop. The main loop is straightforward:

1. Reset the temporary values: The flow rate computed in each device should be re-computed after each cycle.
2. Call devices worker: This will call devices whose state is True. These devices should actively work (i.e. pump)
3. Check Tanks volume: We want to make sure there is the same quantity of fluid in the whole loop. A loss means that some fluid has disappeared due to some miscomputation in the distribution of fluid between devices.
4. Call sensors worker: The sensors will read the data of the Devices they are attached to and make some numeric process if required.
5. Call PLCs worker: The python PLC will read the Modbus databank and update the device state if necessary, then update the Modbus databank accordingly to the value read from the sensors attached to that PLC.
6. Log data: each device, sensor, and PLC data from this loop will be written in a CSV file

7. Sleep: The simulation will sleep a certain amount of time until the next cycle starts.

This pseudo-code 1 summarises the physical simulation's main loop:

Algorithm 1: Main loop

```
Result: None
Initialize Logging system;
Initialize Devices and global configuration ;
Initialize Devices ;
Initialize Modbus Server ;
Wait until all PLCs are connected ;
while max_ cycle not reached do
    reset current_flow_rate of Devices ;
    call Devices Worker ;
    check Tanks volume ;
    call Sensors worker ;
    call PLCs worker ;
    log data of each Devices, Sensors, PLC ;
    sleep() ;
end
```

5.1.2 Devices

Our "Devices" category can be separated into 3 classes:

- Device: The simulation of real devices, this includes multiple kind of tank, pump, filter, valves and vessel.
- Sensor: Discrete and analogue sensors that will measure and take control of the devices.
- PLC: The PLCs will gather data from the Sensors, and make it available for the Modbus Server. It will also read data received from OpenPLC PLCs via ModbusTCP and delegate the control of devices to the sensors.

We can construct the modular simulation using these three classes, resulting in a simplified version of the SWAT process. More details will be provided within respective subsections, However, only a subset of the attributes and functions will be detailed in the following points as detailing each attribute and function in the thesis is unreasonable.

Device

We first have the abstract Device class 5.1. This class is the base class of any device and defines attributes and methods that all devices should share. The device_type is one of the derivated classes of this abstract Device. In our simulation, this includes: 'pump', 'valve', 'filter', 'tank', 'reservoir', 'vessel'. The input_devices dictionary is used to save the previous devices that will output fluid to the actual devices. Similarly, output_devices saves the next devices which our actual devices will output fluid. The active attribute serves as an indicator to know if the device is working, while the state attribute lets the simulator know if the worker method of the device should be called or not. The state attribute avoids lots of calls of worker functions as most devices do not actively pull fluid

by themselves. The math_parser is the type of math parser used by the sympy library to parse mathematical expressions into Python code, which can be used to generate a more realistic input and output volume of fluid when there are multiple input/output devices. More details about this functionality will provided in the Expression formula subsection 5.1.3. As you can infer, the output_devices_expr stores the mathematical expressions used to compute the output fluid volume to each output device. Similarly, it can also be done for the input. The precision parameter will limit the number of digits of a float value.

The from_yaml method is used to override the default loader to add custom YAML tags and be able to load Python objects from the YAML configuration file. The simulator will repetitively call the worker method. Only devices that should actively pull fluid should define this function. i.e. In the case of a pump, this function will trigger the output function of its input devices, which their turn trigger the output function of its input devices, until they reach a device similar to a Tank, which will try to satisfy the request, and trigger the input function of its output devices, until reaching another Tank similar devices which can store the provided amount of fluid.

```
class Device(yaml.YAMLObject):
    def __init__(self, device_type=None, fluid=None,
                 label='', state=None):
        self.device_type = device_type
        self.label = label
        self.input_devices = {}
        self.output_devices = {}
        self.fluid = fluid
        self.current_flow_rate = 0
        self.active = False
        self.state = state
        self.math_parser = None
        self.output_devices_expr = {}
        self.input_devices_expr = {}
        self.symbol_dict = {}
        self.precision = 10

    @classmethod
    def from_yaml(cls, loader, node):
        fields = loader.construct_mapping(node, deep=False)
        return cls(**fields)

    @abstractmethod
    def worker(self):

    @abstractmethod
    def input(self, fluid: Fluid, volume: Union[int, float]):

    @abstractmethod
    def output(self, to_device, volume: Union[int, float]):
```

...

Listing 5.1: Devices

In the following text, we will call devices from which we can input fluid 'input devices', and inversely, devices from which we can output fluid 'output devices'. Only a subset of the devices and functions will be presented as they have some particularities, others will be described briefly.

We start the introduction of devices by the one that makes the simulation run: A Pump 5.2. This is a typical device that needs to implement the worker method. As it is the pump that should initiate a movement in the physical simulation. The pump inheriting from Device, implements the worker, input and output method.

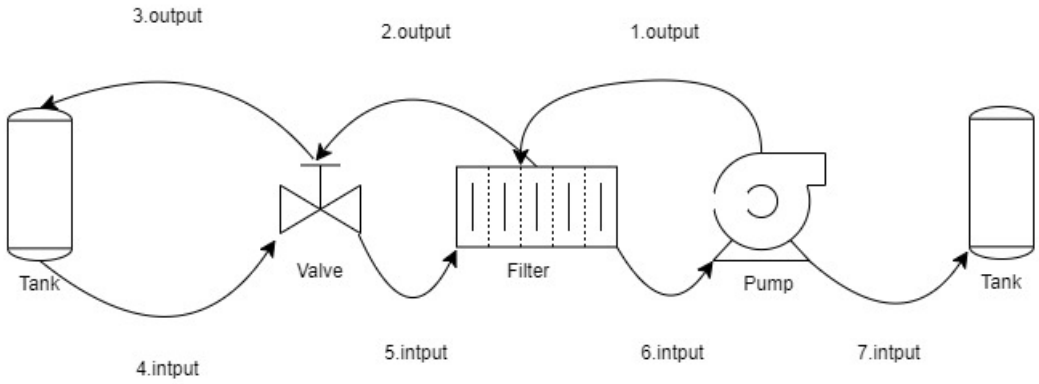

Figure 5.1: Flowchart of fluid process

The output method will call the output method of input devices. A warning can be displayed if there is more fluid that flows through the device than its maximum capacity (volume_per_cycle). A typical scenario would be in the case of pumps in series, the second pump, pulling water from the previous one will augment the flow rate of the previous one, which can result in an overwhelming of the device. Similarly, the input method will update the fluid processed by the device, add the quantity of fluid processed to the current flow rate, and call the input method of output devices. Finally, the worker method will just call the output method of devices that input fluids to itself. An illustration can be found here 5.1.

```
class Pump(Device):
    yaml_tag = u'!pump'
    yaml_loader = yaml.Loader

    def __init__(self, device_type='pump', state='off',
                 volume_per_cycle: Union[int, float] = 1, **kwargs):
        state = bool(['off', 'on'].index(state))
        self.volume_per_cycle = volume_per_cycle
        super(Pump, self).__init__( \
        device_type=device_type, state=state, **kwargs)

    def worker(self):
        if self.state:
            self.output_fluid(self.volume_per_cycle)
```

```
    def input(self, fluid: Fluid, volume: Union[int, float] = 1)
            -> Union[int, float]:
        if self.state:
            self.fluid = fluid
            self.current_flow_rate += volume
            self.input_fluid(fluid, volume)
            return volume
        else:
            return 0

    def output(self, to_device: 'Device', volume: Union[int, float] = 1)
            -> Union[int, float]:
        if self.state:
            if self.current_flow_rate + volume >= self.volume_per_cycle:
                log.warning(f"EXCEED {self} volume_per_cycle")
            self.output_fluid(volume)
            return volume
        else:
            return 0
...
```

Listing 5.2: Pump

The output_fluid function is described here 5.3. It functions as follows: We first assess that there are device(s) from which we can get an input of fluid. If the 'math_parser' is 'proportional', then it will just call the different input devices with the volume computed as a division of the total volume per the number of input devices. For 'sympy' and 'wolfram' 'math_parser', the same applies, but the volume requested from each input device will be computed by a mathematical expression given by the user. Inversely, the input_fluid will call the input method of the output devices.

```
def output_fluid(self, volume: Union[int, float]) -> Union[int, None]:
    open_device_number = 0
    for o in self.input_devices:
        if self.input_devices[o].active or \
           self.input_devices[o].active is None:
            open_device_number += 1
    # When no tank can take the input of fluid
    if open_device_number == 0:
        log.error(f"Device {self} has no device to get input fluid")
        return 0
    if self.math_parser == Allowed_math_type.proportional.value:
        for o in self.input_devices:
            # Request the fluid from all inputs devices equally
            self.input_devices[o].output( \
            self, sp_evalf.N(volume / open_device_number, self.precision))
    else:
        self.symbol_dict['requested_volume'] = volume
```

```
        if self.math_parser == Allowed_math_type.sympy.value:
            for devices_label, expr in self.input_devices_expr.items():
                self.symbol_dict[devices_label].output(self, \
                sp_evalf.N(sp.parse_expr(expr, \
                    local_dict=self.symbol_dict), self.precision))
        elif self.math_parser == Allowed_math_type.wolfram.value:
            for devices_label, expr in self.input_devices_expr.items():
                self.symbol_dict[devices_label].output( \
                self, sp_evalf.N(mp.mathematica(expr).subs( \
                self.symbol_dict), self.precision))
```

Listing 5.3: output_function

Another worth mentioning device is the Tank device. The tank will provide and receive fluid from other devices. When a tank receives fluid, the input function will be called. Before accepting the input fluid, it will first check if he can accept it, if not a warning log will be generated, and the excess fluid will be thrown away. Similarly for the output, of fluid, before proceeding to the output, the tank will first check if it has enough fluid to output before actually outputting.

```
class Tank(Device):
    yaml_tag = u'!tank'
    yaml_loader = yaml.Loader

    def __init__(self, volume: Union[int, float] = 0,
                 max_volume: float = float('inf'),
                 device_type: str = 'tank',
                 state: bool = True, **kwargs):
        self.volume = volume
        self.max_volume = max_volume
        super(Tank, self).__init__(device_type=device_type,
                                   state=state, **kwargs)

    def __increase_volume(self, volume: Union[int, float]):
        self.volume += self.__check_increase_volume(volume)
        return volume

    def __decrease_volume(self, volume: Union[int, float]):
        self.volume -= self.__check_decrease_volume(volume)
        return volume

    def __check_increase_volume(self, volume: Union[int, float]):
        if self.volume == self.max_volume:
            volume = 0
            log.warning(f"{self} full")
        elif self.volume + volume < self.max_volume:
            volume = volume
        else:
            volume = self.max_volume - self.volume
```

```
            log.warning(f"{self} max volume reached")
        return volume

    def __check_decrease_volume(self, volume: Union[int, float]):
        if self.volume <= 0:
            volume = 0
            log.warning(f"{self} empty")
        elif self.volume > volume:
            volume = volume
        else:
            volume = self.volume
            log.warning(f"{self} empty")
        return volume

    def __update_fluid(self, new_context):
        self.fluid = new_context

    def input(self, fluid: Fluid, volume: Union[int, float] = 1):
        self.__update_fluid(fluid)
        accepted_volume = self.__increase_volume(volume)
        self.current_flow_rate += volume
        return accepted_volume

    def output(self, to_device: 'Device', volume: Union[int, float] = 1):
        accepted_volume = to_device.input(self.fluid,
        self.__check_decrease_volume(volume))
        self.__decrease_volume(accepted_volume)
        self.current_flow_rate -= volume
        return accepted_volume
```

Listing 5.4: Tank device

Two other devices are inherited from Tank, the Reservoir is a tank that can get a periodical input of fluid, while the Vessel will allow fluid to pass through it once the device is full. The Filter device will just let fluid pass through it. And finally, Valve can be used to enable and disable connections between devices.

Sensors

Similarly to devices, we also defined a base class for sensors. The Sensor class define the basic attributes required of a sensor, including which type it is, a unique uid and label to recognize it, the state attribute to decide if we call the worker method or not, the device to monitor, and finally the location attribute. The PLC uses this attribute to determine where in the databank the Sensor is reading/writing. As we do not need any input register/coil, we decided that output coils should start by an X(binary), and output registers by a W (Word).

The AnalogSensor 5.5 class inherits from Sensor and defines the number of digits to round and a multiplier to scale the read data. To reproduce the inaccuracy of real-world sensors, the random_generator is used to introduce white noise into the measure of ana-

logue sensors. We choose a standard deviation as it is the most common one.

```
class AnalogSensor(Sensor):
    def __init__(self, sensor_type: str = None, precision: int = 5,
                 multiplier: Union[int, float] = 1, seed: int = 123,
                 standard_deviation: int = 0, **kwargs) -> None:
        self.precision = precision
        self.multiplier = multiplier
        self.random_generator = random.default_rng(seed)
        self.standard_deviation = standard_deviation
        super().__init__(sensor_type=sensor_type, **kwargs)
```

Listing 5.5: Analogue sensors

From these abstract Sensors, we then implemented three types of sensors

- FlowRateSensor: It is an AnalogSensor that will read the current flow rate of the attached Device
- VolumeSensor: It is an AnalogSensor that will read the volume stored in a Tank.
- StateSensor: It is a (discrete) Sensor that can read the state of the attached Device and turn on/off its state.

PLCs

Finally, the last type of device is the PLC. The Base_PLC besides uid, precision, label, a list of controlled sensors and state attributes, defines a connection_ established_coil, this area in the databank is reserved so that the simulator can call the PLC check_connected function so that it is sure the OpenPLC ModbusTCP client is already operational. As for other devices and sensors, the PLCs worker will be called at each simulation loop, collecting data from its attached sensors, and writing it to the right location in the ModbusTCP databank. When the sensor is a StateSensor, the PLC also have to, via the sensor, change the state of the device. When it is an Analogue Sensor (Volume, FlowRate), as basic Modbus TCP only supports 16-bit integers, we must assess the measured data before writing it into the databank. If this measure is too high/low, we will convert it to the maximum/minimum value ModbusTCP supports.

```
class PLC(Base_PLC):
    yaml_tag = u'!plc'
    yaml_loader = yaml.Loader

    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def check_connected(self) -> bool:
        coil_state = \
        self.data_bank.get_coils( \
        self.connection_established_coil, 1)
        if coil_state is not None and coil_state[0]:
            return True
        else:
```

```
            return False

    def worker(self) -> None:
        for sensor in self.controlled_sensors.values():
            if type(sensor) == StateSensor:
                if "X" == sensor.location_tuple[0] and sensor.active:
                    coil_data = \
                    self.data_bank.get_coils( \
                    sensor.location_tuple[1], 1)
                    if coil_data is not None:
                        if coil_data[0]:
                            sensor.device_to_monitor.activate()
                            self.data_bank.set_coils( \
                            sensor.location_tuple[1], [True])
                        else:
                            sensor.device_to_monitor.deactivate()
                            self.data_bank.set_coils( \
                            sensor.location_tuple[1], [False])
                    else:
                        log.error(f"Error reading coil {sensor.location}")
            elif type(sensor) == VolumeSensor or \
                 type(sensor) == FlowRateSensor:
                if "W" == sensor.location_tuple[0]:
                    sensor_value = \
                    int(sensor.read_sensor() * sensor.multiplier)
                    if sensor_value > _16BITS:
                        sensor_value = _16BITS
                    elif sensor_value < _ZERO:
                        sensor_value = _ZERO
                    self.data_bank. \
                    set_holding_registers( \
                    sensor.location_tuple[1], [sensor_value])
```

Listing 5.6: Python PLC

5.1.3 Configuration File

To make a simulation as generic as possible, we opt for generating the physics ICS by a configuration file. Many formats can be used to generate Python Objects from text files, the most common ones are JSON, XML and YAML. We opt for YAML for its simplicity of writing and clear structure.

YAML is a human-readable data serialization language that is often used for writing configuration files. YAML files are simpler to read as they use Python-style indentation to determine the structure and indicate nesting. Tab characters are not allowed by design, to maintain portability across systems, so whitespaces—literal space characters—are used instead. Comments can be identified with a hash symbol (#). It has to be noted that YAML is case-sensitive and whitespace-sensitive.

The file structure can be separated into 6 sections:

- Settings: Global settings of the simulation
- Devices: Defines Devices used in the simulation
- Connections: Defines connections between devices
- Symbols: Defines symbols used in mathematical expression when the 'sympy'/'wolfram' math parser is used.
- Sensors: Defines Sensors used in the simulation
- PLCs: Defines PLC used in the simulation

In the following text, we will detail each section, the attributes that we can (should) set, and their meaning.

Settings

The Settings section 5.7 defines global settings of the simulator, including the sleeping time main loop, the default precision used by devices and sensors, the number of cycles the simulation should run, the IP address and port used by the ModbusTCP server, and the time constraint of the OpenPLC ladder logic.

```
settings:
  # Sleeping time between simulation cycles in milliseconds
  sim_speed: 200
  # When generating PLC ladder logic
  # the maximum time of a PLC ladder logic loop
  plc_speed: 20
  # The number of digits of numbers
  precision: 5
  # The number of cycles the simulation will run
  # If 0, it will run infinitely
  max_cycle: 0
  # IP of the physic simulation, default is "172.18.0.10"
  host_address: "172.18.0.10"
  # Port of the ModbusTCP server, default is 502.
  port: 12345
```

Listing 5.7: Sample Simulation settings

Devices

The Devices section 5.8 defines Devices used in the simulation. Besides default tags, the tags defined in the Device class can be used to initiate different Devices. Available tags are '!pump', '!valve', '!filter', '!tank', '!reservoir' and '!vessel'. The label key is mandatory and should be unique to avoid surprises when we connect devices between themselves.

A Tank (Reservoir) device should set a maximum volume (max_volume), and it is possible to set the current volume(volume) of fluid present in the device and the type of fluid contained in it. A reservoir can take fluid input without a tank and pump, pumping fluid into it, to do that you can set the 'self_input' at 'yes', and define the volume added at each cycle('input_per_cycle), by default, the 'self_input' is set as 'no'.

The Pump ('!pump') device should set a pumping capacity('volume_per_cycle') and the state of the device ('on'/'off'), the default pumping capacity is 1 and the state is 'off'.

The Valve ('!valve') device state is by default 'closed', but can be opened if set to 'open'

A Vessel ('!vessel') is modelled as a combination of filter and tank. Once the maximum volume of the Vessel is reached, the fluid can pass through the vessel. Similarly to a Tank(reservoir), it should be set a maximum volume(max_volume) , and can set a current volume('volume')

The basic Filter('!filter') only requires a label to be used.

```
devices:
  - !reservoir
    label: T-101
    max_volume: 10000
    volume: 1000
    fluid: !water {}
    self_input: 'yes'
    input_per_cycle: 20
  - !pump
    label: P-101
    volume_per_cycle: 10
    state: 'on'
  - !valve
    label: MV-201
    state: 'open'
  - !vessel
    label: RO-501
    volume: 0
    max_volume: 40
  - !filter
    label: Filter -501
```

Listing 5.8: Example of Devices settings

Connections

It is in this section that we define the connections between devices. In the example 5.9, the Reservoir with label 'reservoir1' have two outputs, which are respectively 'pump1' and 'pump2'. These two pumps respectively output their fluid into a Tank, which label is 'tank'. In the example, as we are using the sympy math parser, we need to define the input and output expression of all connections. It works as follows: Before the colon (:) is the label of the output/input device, and after is the mathematical expression, which results in the volume of fluid that should be pulled/pushed from/to.

All attributes of devices in the simulator can be accessed with the same syntax as you want to access it in Python ('device_label.attribute').

For input devices, in the 'output_devices_expr' the following extra variables can be used:

- 'requested_volume': The volume of fluid that the device is requested to provide or pull.

- 'open_output_devices_number': The number of devices directly connected to the actual device to which we may pull fluid.

Similarly, for output devices, in the 'input_devices_expr' the following extra variables can be used:

- 'accepted_volume': The volume of fluid that is allowed by the tank to pass through.
- 'open_input_devices_number': The number of devices directly connected to the actual device to which we may push fluid.

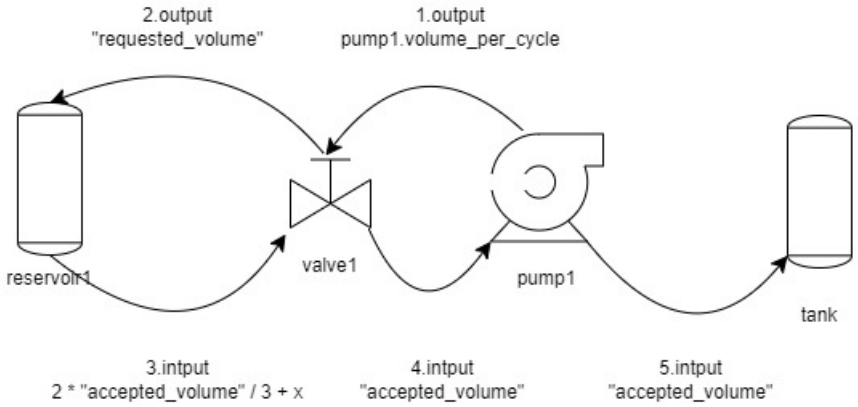

Figure 5.2: Flowchart of fluid process with sympy

A simplified illustration 5.2 of the configuration 5.9 can be described as follows:

When the pump1 worker is called, it will firstly compute the fluid quantity it wants to request, which can be accessed as 'pump1.volume_per_cycle' or 'requested_volume'. Then will trigger the output function of the input device (valve1 in our case), which in turn will request a 'requested_volume' volume of fluid to the reservoir1. As the reservoir1 is the input device of valve1, we have reached a Tank, the Tank will provide an 'accepted_volume' of fluid, and call the input function of valve1, which volume is computed by the formula: $2*"accepted_volume"/3+x$. 'x' is a symbol defined in the Symbols section, you can consider it as a constant. Then pump1 will call the input function of the pump1, which will let the "accepted_volume" pass through it, and finally reach tank1. It works similarly for pump2.

```
connections:
  reservoir1:
    outputs:
      - valve1
  valve1:
    outputs:
      - pump1
      - pump2
    input_devices_expr:
      reservoir1: "requested_volume"
    output_devices_expr:
      pump1: 2 * "accepted_volume" / 3 + x
      pump2: 2 * "accepted_volume" / 3 + x
  pump1:
```

```
    outputs:
      - tank1
    input_devices_expr:
      valve1: pump1.volume_per_cycle
    output_devices_expr:
      tank: "accepted_volume"
  pump2:
    outputs:
      - tank1
    input_devices_expr:
      valve1: pump2.volume_per_cycle
    output_devices_expr:
      tank: "accepted_volume"
  tank1:
    input_devices_expr:
      pump1: "accepted_volume"
      pump2: "accepted_volume"
```

Listing 5.9: Sample Connection settings with sympy

Symbols

The Symbols section 5.10 is where we define symbols that are used in the Connections if we choose to launch the simulation with sympy or wolfram math parser. The declaration of such symbols is the same as we have done for mathematical expression in the Connections part. The left of the colon (:) is the symbol to define, and the right is the constant value attributed to it. Be careful to not use some Python keywords as symbols.

```
symbols:
  x: 10
  y: 100
  z: 1000
```

Listing 5.10: Example of symbols definition

Sensors

The Sensors section 5.11 defines Sensors used in the simulation. Besides default tags, the tags defined in the Device class can be used to initiate different Devices. Available tags are '!state','!flowrate' and '!volume'. The label key is mandatory and should be unique to avoid key problems in the logging system. There are two types of location in Modbus, the binaries and 16-bit integers. Binary values defined by discrete sensors such as the StateSensor should be located in values from X0 to X65535. Similarly, AnalogueSensor such as the VolumeSensor and the FlowrateSensor should write their data to locations from W0 to W65535. The location of binary data and word data are independent, however, having the same location for two different sensors will make one override the data of the other.

Similarly to devices, Sensors also require the state to be set as 'on' if we want its worker method to be called. The sensor is attached to a Device by 'connected_to' which requires the Device label.

To introduce White noise to the measure taken by the sensor, we can define the standard_deviation. The number given to the standard_deviation is the scaling parameter of normal (Gaussian) distribution. For example, with a scale of 1, 70% of sample drawn from the normal distribution will be within -1 and 1, while 27.2% of them will be between -2 and -1, or 1 and 2, and rarely extremely rarely more than 3. When adding white noise, it is recommended to set the seed to be able to reproduce the result. To have better results, setting the multiplier can also be considered. Due to the limitation of Modbus for 16-bit integers, float values will be rounded, resulting in a loss of such white noise in the OpenPLC PLC or SCADA/historian database. Fortunately, the physical simulation will be able to log these float values without this limitation.

```
sensors:
  - !state
    label: MV101
    state: 'on'
    connected_to: MV-101
    location: X0
  - !volume
    label: P101
    state: 'on'
    connected_to: P-101
    location: X0
    multiplier: 1
  - !flowrate
    label: P101
    state: 'on'
    connected_to: P-102
    location: W1
    precision: 8
    seed: 123
    standard_deviation: 1
```

Listing 5.11: Example of Sensors settings

PLCs

Finally, we will define PLCs 5.12, the tag used by PLC is '!plc'. Similarly to Device and sensors, it requires a label and set the state to 'on' to be working. 'connection_established_coil' is the coil used to make sure an OpenPLC instance is connected and can write to the ModbusTCP server. This value can vary between 0 and 65535. The last parameter 'controlled_sensors_label' is the list of sensors (label) controlled (monitored) by the PLC.

```
  - !plc
    label: PLC1
    state: 'on'
    connection_established_coil: 65001
    controlled_sensors_label:
      - P101
      - P102
```

- **FIT201**
- **LIT101**

Listing 5.12: Example of PLC settings

Generation of PLC ladder logic

Another worth mentioning feature is the generation of PLC logic via the configuration file when launching the simulation with the '–generation' parameter.

As a configuration file, ladder logic has a strict structure. The program part contains variables and logic connections between the different variables. The logic that we want to apply to the PLC is simple. We want data read from the input contact/register to be passed to the output coil/register. As we already defined the location used by each sensor in the Modbus databank, we can translate this address to the OpenPLC mapping one. OpenPLC contact and coils are grouped by 8. For example, we would map a sensor location of X10 to '%IX1.2' and '%QX1.2' ($10 = 1 * 8 + 2$), Analogue values do not require such conversion as they use similar syntax.

For the logic part, as we only move the input to the output, we only need to attribute the output coils/registers values to the input contact/registers values. The end of the file consists of a simple configuration, the INTERVAL is the 'plc_speed' set in the global configuration of the simulation, and the name of this program is the label of the PLCs defined in the configuration file.

This logic created in OpenPLC Editor 5.3 creates the same ladder logic5.13 as generated by the Python simulation.

#	Name	Class	Type	Location
3	P102_Q	Local	BOOL	%QX0.1
4	P102_I	Local	BOOL	%IX0.1
5	LIT101_Q	Local	INT	%QW0
6	LIT101_I	Local	INT	%IW0
7	FIT201_Q	Local	INT	%QW1
8	FIT201_I	Local	INT	%IW1

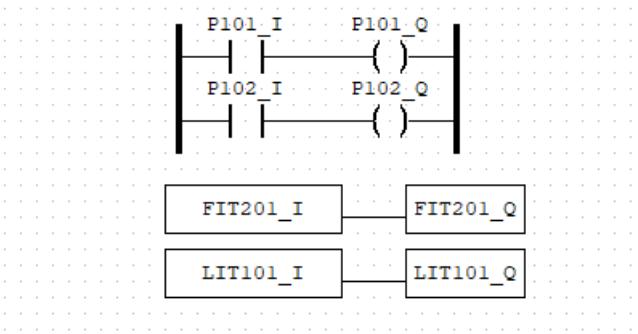

Figure 5.3: OpenPLC Editor graphic visualisation

```
PROGRAM PLC1
  VAR
    P101_Q AT %QX0.0 : BOOL;
    P101_I AT %IX0.0 : BOOL;
    P102_Q AT %QX0.1 : BOOL;
    P102_I AT %IX0.1 : BOOL;
```

```
    FIT201_Q AT %QW0 : INT;
    FIT201_I AT %IW0 : INT;
    LIT101_Q AT %QW1 : INT;
    LIT101_I AT %IW1 : INT;
  END_VAR

  P101_Q := P101_I;
  P102_Q := P102_I;
  FIT201_Q := FIT201_I;
  LIT101_Q := LIT101_I;
END_PROGRAM


CONFIGURATION Config0

  RESOURCE Res0 ON PLC
    TASK task0(INTERVAL := T#20ms,PRIORITY := 0);
    PROGRAM instance0 WITH task0 : PLC1;
  END_RESOURCE
END_CONFIGURATION
```

Listing 5.13: Generated PLC ladder logic

5.2 Limitations

At the actual state, the modularity of the sandbox has a significant advantage and can be easily extended for other use cases. However, the construction of the configuration file can be time-consuming, especially if the sympy/wolfram math parser is used. Another limitation, due to the check of fluid quantity check, no fluid can be leaked or drained out of the physical simulation, it may also be possible that there is some loss due to float number precision, which can also trigger a warning. Of course, this check can be disabled if required, but a better way to assess the integrity of the fluid contained in the simulation should be implemented. Another limitation is the lack of comparison with real-world data, As our simulation only replicates a simplified version of the SWAT process, we cannot completely compare our dataset to the one provided by the iTrust lab.

Chapter 6
Discussion

6.1 Comparison with state of the art/related works

Based on Smadi et al. [63] criteria of an ICS, we make the following table 6.1. The scoring has been attributed from 1 to 5, 1 being the least and 5 being the best in the category.

For Fidelity, we estimate that a score of 1 will be simulations that neither have 'complex' physical simulation, IEC61131-3 compliant logic, nor network simulation. While 5 should implement a complete physical simulation.

For Repeatability, 1 is given to simulations that cannot be reproduced by other researchers due to a lot of factors(cost, equipment, time...), while 5 will given to Open-source solution which does not require any devices other than a personal computer.

Flexibility is complicated to assess as most sandboxes are created to follow a specific process, but those with generative topologies, code-based simulation will receive higher scores than those having a precise goal.

For Cost-effectiveness, the more lightweight and devices independent the simulation is, the higher the score will be. Those requiring extra physical hardware would receive lower scores.

Scalability, In most scenarios, the scalability of the sandbox is not demonstrated, and can only be inferred via the tool used, or based on the number of physical devices involved. From our perspective, Fully software-based, with automatism will receive higher scores than those using physical devices. In terms of technologies, KVM-based solutions will receive lower scores than container-based ones and themselves lower than kernel-based (Jailhouse, Mininet)

For Diversity, multiple factors can be taken into account, PLC diversity (combining different manufacturers), communication protocol diversity, Physical device diversity, network component diversity and the use of HMI, historian and so on.

In the table, we named our sandbox SWAT_V as we attempted to create a virtual sandbox of the SWAT testbed. In the comparison testbed, we achieved first at the total scoring, but we acknowledge that the testbed has room for a lot of improvement, including the need for a more accurate physical system, the implementation of more communication

Testbed	Fidelity	Repeatability	Flexibility	Cost	Scalability	Diversity
SWAT_V	2	5	4	5	4	4
SWAT [37]	5	1	1	1	2	2
LICSTER [62]	4	3	2	3	2	1
NPP [13]	3	5	2	5	3	3
Paolo's [66]	1	5	4	5	4	4
ICSSIM [24]	2	5	4	5	3	3
GRFICS [33]	2	5	3	5	3	4
VISCORT [29]	2	5	3	5	4	4
Masset et al. [45]	2	4	4	5	5	4

Table 6.1: Comparison of related work sandboxes

protocols, and improvement of the HMI.

6.2 Lessons learned

The methodology applied was pretty simple, we started by reviewing state-of-the-art articles before deciding which technologies we wanted to apply and finally started the development of the different components of the ICS one by one. Beginning with the physical simulation, then the network simulation, followed by deploying OpenPLC and HMI into containers. Documentation of the project was done simultaneously, while this thesis was written after all implementation was done.

There are some flaws encountered during the project that can be avoided. One of them is during the deployment of OpenPLC. As this work is my some points based on the work of Masset Bertrand and Olivier Taburiaux [45], we thought that OpenPLC_V3 can be easily deployed into the non SDN version of mininet (ipmininet), but it was not the case. We should have done some proof of concept before rushing into the actual deployment and linkage of all devices used in the simulation.

6.3 Limitations of validity

Our sandbox has some limitations due to its simplistic design, the use of docker networking and lack of real-world data testing:

- Inaccurate physical simulation: The sandbox does not precisely replicate the physical behaviour of industrial control systems, but can be complex enough for a lot of applications.
- Simplistic Design: The sandbox is designed to be straightforward to use, which means it lacks the complexity found in actual industrial control systems.
- Modular and Adaptable: While the modularity of the sandbox is a significant advantage, it also means that the initial setup and configuration can be extensive.
- Docker Network Limitations: The use of Docker networking poses challenges for implementing real-world networking configurations. Docker's network abstraction might not fully capture the complexity of industrial network protocols and interactions, which could limit the sandbox's effectiveness in testing network-specific scenarios.
- Lack of Real-World Data Testing: Our sandbox has not been validated against real-world data, raising questions about the accuracy and usability of the datasets it generates.

Chapter 7
Future work

There are several possible works to go further with our ICS simulator.

- Testing with physical devices: Devices such as Raspberry can be useful to verify the interoperability between simulated devices and real devices which should not encounter any problems as our network is emulated and the Modbus/TCP messages meet the specifications.
- Attack scenarios can be conducted on vulnerable devices: for example, an old version of OpenPLC has a vulnerability under CVE-2018-20818, and an exploit can be found [32].
- Lack of comparison with real-world data: Compared to the iTrust Secure Water Treatment testbed [37] the physical simulation still lacks some devices and does not support multiple fluids well. Thus continuing the development of the physical simulation can also be considered. However, the more the physical process is complicated, the more it will be resource-consuming. Thus things should be balanced.
- Development of a complete HMI: As we did not have time to work on the graphical view, its development can highly improve its educational purpose as having graphic support can greatly increase empathy when there is graphic support.
- Multiple communication protocols between Physical node and PLCs: The communication between the physical simulation and PLCs is done only with Modbus TCP. As both OpenPLC PSM and the physical simulation use Python, implementing other communication protocols than ModbusTCP should be possible.
- Protection (security) of ICS: Research on detection and defence against ICS attacks can be a good research topic with the use of the developed testbed.

Chapter 8
Conclusions

8.1 Conclusions

In this master thesis, we made a dedicated chapter on the different components of an ICS system, with details on PLC and ladder logic, MTU, SCADA, HMI, and common communication protocols such as Profinet, EtherCat, Powerlink, DNP3, EIP and Modbus and its variant.

The state of the art of ICS testbeds, focusing on virtualised testbeds. In the different testbeds, some attire more our attention such as the SWAT [37] which is a fully physical testbed with a complete technical description of the testbed and provides a dataset with normal operation and under attacks that can be used to further to develop IDS solutions, other virtual testbeds such as GRFICS [33], VISCORT [29] that implements a complete virtual testbed with an interesting proof of concept. The ndn-water-treatment-testbed [46] has an interesting modular approach to create the physical simulation which inspires us in the design of our physical simulation design.

The main objective to create a virtual ICS testbed has been accomplished, by using functional standardized open source IEC 61131-3 compliant PLC simulator OpenPLC, both in software and in hardware as PLC.

A modular approach to create and configure a physical simulation, that tries to replicate a physical testbed (SWAT [37]). This physical simulation is created in Python, with a YAML configuration file, and tries to have the simplest code to enable easy adaptation to different ICS scenarios. Multiple ways to compute the input/output between devices have been developed, including adding white noise into measurement to replicate the real-world inaccuracy of sensors. The possibility to create OpenPLC compatible ladder logic from the configuration file.

ScadaBR (or SacadaLTS) is used as an HMI and historian to monitor all data generated by the PLCs.

The network simulation part is assured by Docker, all of the previous components are separated into containers, including a physical node, 6 PLC nodes, a ScadaBR/ScadaLTS node, and a MySQL node used as historian for the ScadaBR/ScadaLTS, all grouped into a Docker user-defined network.

Multiple bash scripts ensure all this deployment to ease the configuration of the components and include a vagrant file to facilitate the installation of a VM with the VM network configured and docker installed.

8.2 Use of AI

In the course of preparing this thesis, I utilized ChatGPT 3.5, a large language model developed by OpenAI, to assist with language correction and refinement. Specifically, ChatGPT was employed to review and suggest improvements to the clarity, coherence, and grammatical accuracy of my writing. The last access was the 2 June. The tool provided valuable feedback, which I carefully reviewed and incorporated as appropriate

while maintaining full responsibility for the content and interpretations presented in this work.

8.3 Source code and vagrant

The source code is available at: `https://github.com/Wherriea2333/ICS_Water_testbed`

Adaptation of Docker containers of OpenPLC: `https://github.com/Wherriea2333/OpenPLC_v3_customized`

Adaptation of Docker files of scadaBR: `https://github.com/Wherriea2333/scadabr`

The readme page explains how to install everything from scratch if you want to reproduce the testbed on your own. A link to the vagrant box may also be provided in the readme.

Bibliography

[1] Aboah Boateng, E., Bruce, J.W.: Unsupervised machine learning techniques for detecting plc process control anomalies. Journal of Cybersecurity and Privacy 2(22), 220–244 (Jun 2022) [Cited on page 21.]

[2] Alanazi, M., Mahmood, A., Chowdhury, M.J.M.: Scada vulnerabilities and attacks: A review of the state-of-the-art and open issues. Computers & Security 125, 103028 (2023) [Cited on page 5.]

[3] Albin, E., Rowe, N.C.: A realistic experimental comparison of the suricata and snort intrusion-detection systems. In: 2012 26th International Conference on Advanced Information Networking and Applications Workshops. pp. 122–127 (March 2012) [Cited on page 21.]

[4] Alves, T., Das, R., Werth, A., Morris, T.: Virtualization of scada testbeds for cybersecurity research: A modular approach. Computers & Security 77, 531–546 (Aug 2018) [Cited on page 18.]

[5] Alves, T., Morris, T.: Openplc: An iec 61,131–3 compliant open source industrial controller for cyber security research. Computers & Security 78, 364–379 (2018) [Cited on pages 19 and 24.]

[6] Antonioli, D., Tippenhauer, N.O.: Minicps: A toolkit for security research on cps networks (2015) [Cited on page 16.]

[7] arj3090: Advancedhmi software (2024), `https://www.advancedhmi.com/`, [Online; accessed 11-May-2024] [Cited on page 19.]

[8] autonomy: Sl-rp4 industrial raspberry pi 4 plc device (2024), `https://autonomylogic.com/product/sl-rp4/`, [Online; accessed 21-April-2024] [Cited on pages V and 25.]

[9] Babu, V., Nicol, D.M.: Emulation/simulation of plc networks with the s3f network simulator. In: 2016 Winter Simulation Conference (WSC). pp. 1475–1486 (Dec 2016) [Cited on page 17.]

[10] beremiz: Beremiz: Open source framework for automation (2024), `https://beremiz.org/`, [Online; accessed 22-April-2024] [Cited on page 25.]

[11] BERR, J.: "wannacry" ransomware attack losses could reach $4 billion (5 2017), `https://web.archive.org/web/20170614105447/http://www.cbsnews.com/news/wannacry-ransomware-attacks-wannacry-virus-losses/` [Cited on page 1.]

[12] bitelxux: scadabr. `https://github.com/Wherriea2333/scadabr` (2024) [Cited on page 38.]

[13] de Brito, I.B., de Sousa, R.T.: Development of an open-source testbed based on the modbus protocol for cybersecurity analysis of nuclear power plants. Applied Sciences 12(1515), 7942 (Jan 2022) [Cited on pages 18, 35, and 59.]

[14] Chen, X.: Tennessee eastman simulation dataset. IEEE Dataport (2019) [Cited on pages V and 36.]

[15] Conti, M., Donadel, D., Turrin, F.: A survey on industrial control system testbeds and datasets for security research. IEEE Communications Surveys & Tutorials 23(4), 2248–2294 (2021) [Cited on page 14.]

[16] Contributors, A.: aguaclara. https://github.com/AguaClara/aguaclara (2024) [Cited on page 36.]

[17] Contributors, A.: docker-compose.yml. https://github.com/SCADA-LTS/Scada-LTS/blob/develop/docker-compose.yml (2024) [Cited on page 38.]

[18] Contributors, A.: ipmininet. https://github.com/cnp3/ipmininet (2024) [Cited on page 22.]

[19] Contributors, A.: jailhouse. https://github.com/siemens/jailhouse (2024) [Cited on page 20.]

[20] Contributors, A.: libnetwork. https://github.com/moby/libnetwork (2024) [Cited on page 33.]

[21] Contributors, A.: Scada-lts. https://github.com/SCADA-LTS/Scada-LTS (2024) [Cited on pages 23 and 38.]

[22] Contributors, A.: scadabr. https://github.com/ScadaBR/ScadaBR (2024) [Cited on pages 23 and 38.]

[23] contributors, A.: Sympy (2024), https://www.sympy.org/en/index.html, [Online; accessed 19-May-2024] [Cited on page 43.]

[24] Dehlaghi-Ghadim, A., Balador, A., Moghadam, M.H., Hansson, H., Conti, M.: Icssim — a framework for building industrial control systems security testbeds. Computers in Industry 148, 103906 (Jun 2023) [Cited on pages 18, 35, and 59.]

[25] Dietmar Winkler, Martin Sjolund, L.O.: Wastewater. https://github.com/modelica-3rdparty/WasteWater (2024) [Cited on page 35.]

[26] Docker Project Contributors: Docker (2024), https://www.docker.com/, [Online; accessed 14-May-2024] [Cited on page 23.]

[27] DRAGOS: Ot cybersecurity the 2023 year in review (2 2024), https://www.dragos.com/ot-cybersecurity-year-in-review/#anchor-report [Cited on page 1.]

[28] Ekisa, C.: aguaclara. https://gitlab.com/ekisac10/vicsort (2024) [Cited on page 19.]

[29] Ekisa, C., Briain, D.Ó., Kavanagh, Y.: Vicsort - a virtualised ics open-source research testbed. In: 2022 Cyber Research Conference - Ireland (Cyber-RCI). p. 1–8 (Apr 2022) [Cited on pages 19, 36, 59, and 62.]

[30] EtherCAT: Safety over ethercat (fsoe) (2024), https://www.ethercat.org/en/safety.html, [Online; accessed 14-April-2024] [Cited on page 7.]

[31] Faramondi, L., Flammini, F., Guarino, S., Setola, R.: A hardware-in-the-loop water distribution testbed dataset for cyber-physical security testing. IEEE Access 9, 122385–122396 (2021) [Cited on page 15.]

[32] Feng, K.: Openplc webserver 3 denial of service / buffer overflow (2024), https://packetstormsecurity.com/files/174582/OpenPLC-Webserver-3-Denial-Of-Service-Buffer-Overflow.html, [Online; accessed 22-May-2024] [Cited on page 61.]

[33] Formby, D., Rad, M., Beyah, R.: Lowering the barriers to industrial control system security with grfics. In: 2018 USENIX Workshop on Advances in Security Education (ASE 18). USENIX Association, Baltimore, MD (Aug 2018) [Cited on pages 19, 36, 59, and 62.]

[34] Gaffurini, M., Bellagente, P., Depari, A., Flammini, A., Sisinni, E., Ferrari, P.: Virtual plc in industrial edge platform: Performance evaluation of supervision and control communication. IEEE Transactions on Instrumentation and Measurement 73, 1–10 (2024) [Cited on page 20.]

[35] Gao, H., Peng, Y., Dai, Z., Wang, T., Han, X., Li, H.: An industrial control system testbed based on emulation, physical devices and simulation. In: Butts, J., Shenoi, S. (eds.) Critical Infrastructure Protection VIII. pp. 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg (2014) [Cited on pages 15 and 35.]

[36] gexod: variable retain function (2023), https://openplc.discussion.community/post/variable-retain-function-9914880?trail=30, [Online; accessed 28-April-2024] [Cited on page 29.]

[37] Goh, J., Adepu, S., Junejo, K.N., Mathur, A.: A dataset to support research in the design of secure water treatment systems. In: Havarneanu, G., Setola, R., Nassopoulos, H., Wolthusen, S. (eds.) Critical Information Infrastructures Security. pp. 88–99. Springer International Publishing, Cham (2017) [Cited on pages V, 15, 19, 35, 36, 37, 59, 61, and 62.]

[38] Gundall, M., Reti, D., Schotten, H.: Application of virtualization technologies in novel industrial automation: Catalyst or show-stopper? (11 2020) [Cited on page 20.]

[39] Hackworth, J., Hackworth, F.: Programmable Logic Controllers: Programming Methods and Applications. Programming Methods and Applications, Pearson/Prentice Hall (2004) [Cited on pages IV and 2.]

[40] Holm, H., Karresand, M., Vidström, A., Westring, E.: A Survey of Industrial Control System Testbeds, vol. 9417, p. 11–26. Springer International Publishing, Cham (2015) [Cited on page 14.]

[41] IATC: Programming (2024), https://in.pinterest.com/pin/565131453215637936/, [Online; accessed April 07, 2024] [Cited on pages IV and 1.]

[42] Knapp, E.: Chapter 5 - how industrial networks operate. In: Knapp, E. (ed.) Industrial Network Security, pp. 89–110. Syngress, Boston (2011) [Cited on pages IV and 6.]

[43] Knapp, E.: Chapter 6 - industrial network protocols. In: Knapp, E. (ed.) Industrial Network Security, pp. 121–169. Syngress, Boston (2011) [Cited on pages IV, V, 7, 8, 9, 11, 12, and 13.]

[44] López-Morales, E., Rubio-Medrano, C., Doupé, A., Shoshitaishvili, Y., Wang, R., Bao, T., Ahn, G.J.: Honeyplc: A next-generation honeypot for industrial control systems. In: Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security. p. 279–291. CCS '20, Association for Computing Machinery, New York, NY, USA (Nov 2020) [Cited on page 21.]

[45] Masset, B., Taburiaux, O.: Simulating Industrial Control Systems using mininet. Ph.D. thesis, UCL - Ecole polytechnique de Louvain (2018) [Cited on pages IV, V, 3, 19, 24, 30, 59, and 60.]

[46] Maynard, P.: ndn-water-treatment-testbed. `https://github.com/PMaynard/ndn-water-treatment-testbed` (2024) [Cited on pages 36, 37, and 62.]

[47] Michael Busch: Awlsim: S7 compatible plc / sps (2024), `https://bues.ch/cms/automation/awlsim.html`, [Online; accessed 9-May-2024] [Cited on page 17.]

[48] Michael Kerrisk: tc-netem(8) — linux manual page (2024), `https://man7.org/linux/man-pages/man8/tc-netem.8.html`, [Online; accessed 1-May-2024] [Cited on page 34.]

[49] Mininet Project Contributors: Mininet an instant virtual network on your laptop (or other pc) (2024), `https://mininet.org/`, [Online; accessed 14-May-2024] [Cited on page 22.]

[50] Modbus Organisation: Modbus/tcp security (2018), `https://modbus.org/docs/MB-TCP-Security-v21_2018-07-24.pdf`, [Online; accessed 17-April-2024] [Cited on pages IV and 10.]

[51] Monzer, M.H., Beydoun, K., Ghaith, A., Flaus, J.M.: Model-based ids design for icss. Reliability Engineering & System Safety 225, 108571 (2022) [Cited on page 21.]

[52] Ofenloch, A., Schwarz, J.S., Tolk, D., Brandt, T., Eilers, R., Ramirez, R., Raub, T., Lehnhoff, S.: Mosaik 3.0: Combining time-stepped and discrete event simulation. In: 2022 Open Source Modelling and Simulation of Energy Systems (OSMSES). pp. 1–5 (April 2022) [Cited on page 18.]

[53] Office of Electricity: National scada test bed (2024), `https://www.energy.gov/oe/national-scada-test-bed`, [Online; accessed 10-May-2024] [Cited on page 14.]

[54] OpenPLC: autonomy (2024), `https://autonomylogic.com/`, [Online; accessed 21-April-2024] [Cited on pages 23 and 24.]

[55] Parsons, D.: Five startling findings in 2023's ics cybersecurity data (11 2023), `https://www.sans.org/blog/five-startling-findings-2023-ics-cybersecurity-data/` [Cited on page 1.]

[56] Paxson, V.: Bro: A system for detecting network intruders in Real-Time. In: 7th USENIX Security Symposium (USENIX Security 98). USENIX Association, San Antonio, TX (January 1998) [Cited on page 21.]

[57] Peltonen, M.: Plc virtualization and software defined architectures in industrial control systems (2017), `http://www.theseus.fi/handle/10024/134029`, accepted: 2017-10-04T10:31:53Z [Cited on page 19.]

[58] Pospisil, O., Blazek, P., Kuchar, K., Fujdiak, R., Misurec, J.: Application perspective on cybersecurity testbed for industrial control systems. Sensors 21(2323), 8119 (Jan 2021) [Cited on page 20.]

[59] PROFINET IP North America: Profinet security guideline (2023), https://www.profibus.com/download/profinet-security-guideline, [Online; accessed 14-April-2024] [Cited on page 7.]

[60] Queiroz, C., Mahmood, A., Tari, Z.: Scadasim – a framework for building scada simulations. IEEE Trans. Smart Grid 2, 589–597 (Dec 2011) [Cited on page 18.]

[61] Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings of the 13th USENIX Conference on System Administration. p. 229–238. LISA '99, USENIX Association, USA (1999) [Cited on pages 19 and 21.]

[62] Sauer, F., Niedermaier, M., Kießling, S., Merli, D.: Licster – a low-cost ics security testbed for education and research (2019), arXiv:1910.00303 [cs] [Cited on pages 16 and 59.]

[63] Smadi, A.A., Ajao, B.T., Johnson, B.K., Lei, H., Chakhchoukh, Y., Abu Al-Haija, Q.: A comprehensive survey on cyber-physical smart grid testbed architectures: Requirements and challenges. Electronics 10(99), 1043 (Jan 2021) [Cited on pages 16 and 59.]

[64] Stouffer, K., Falco, J., Kent, K.: Guide to supervisory control and data acquisition (scada) and industrial control systems security (01 2006) [Cited on pages IV and 4.]

[65] Tokyo Research Center: Control system security center (2024), https://www.css-center.or.jp/en/, [Online; accessed 10-May-2024] [Cited on page 14.]

[66] Trungadi, P.: Exploiting virtual networks for CPS security analysis – The Smart Home Environment. laurea, Politecnico di Torino (Apr 2023) [Cited on pages 19 and 59.]

[67] Wikipedia Contributors: Colonial pipeline ransomware attack (2004), https://en.wikipedia.org/wiki/Colonial_Pipeline_ransomware_attac, [Online; accessed 09-April-2024] [Cited on page 1.]

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**