DeMon++: A framework for designing and implementing Distributed Monitoring Systems based on Hierarchical Finite State Machines

Master of Science (Tech) Thesis
University of Turku
Department of Computing
Robotics and Autonomous Systems
2023
Lorenzo Morelli

Supervisors:
Prof. Tomi Westerlund
Christian Kern

UNIVERSITY OF TURKU
Department of Computing

Lorenzo Morelli: DeMon++: A framework for designing and implementing Distributed Monitoring Systems based on Hierarchical Finite State Machines

Master of Science (Tech) Thesis, 60 p.
Robotics and Autonomous Systems
July 2023

In today's interconnected world, the proliferation of diverse and numerous devices has become increasingly common. This phenomenon is particularly evident in the field of industrial computing, which has experienced rapid growth. With this rapid expansion, monitoring an industrial control system (ICS) consisting of a large number of devices becomes a critical activity. To evaluate our approach, we chose the CERN ICS as a suitable case study for our research. The CERN ICS is a complex network of thousands of heterogeneous control devices, including PLCs, front-end computers, supervisory control and data acquisition systems. Our approach resulted in DeMon++, a framework for designing and implementing distributed monitoring systems. DeMon++ uses the concept of hierarchical finite state machines to model the system, capturing the hierarchical relationship between devices. In particular, DeMon++ aims to be a flexible, scalable and maintainable monitoring framework to abstract, aggregate and summarise the health state of industrial control systems composed of a heterogeneous set of devices. As part of the CERN OpenLab programme, this thesis provides a flexible and maintainable approach to monitoring complex and distributed ICS, with a particular focus on the demanding environment of CERN.

Keywords: Industrial control system, industrial device monitoring, hierarchical finite state machine, PLC, IoT, CERN ICS, CERN OpenLab, Edge Computing

Acknowledgements

I would like to express my sincere gratitude to my supervisor at Siemens, Christian Kern, who made this thesis possible. His guidance and advice carried me through all the stages of writing my project. I am sincerely grateful to Abhit Patil from CERN and his team for their strong support, patience and faith in me. Finally, I would also like to thank my university professor Tomi Westerlund for his brilliant comments and suggestions.

Figure 1: We at CERN

Contents

List of Figures

List of acronyms

DCSs distributed control systems

ERP enterprise resource planning

HMIs human-machine interfaces

ICS Industrial Control System

IEAP Industrial Edge App Publisher

IED Industrial Edge Device

IEH Industrial Edge Hub

IEM Industrial Edge Management

IoT Internet of Things

PLC Programmable Logic Controllers

RTUs remote terminal units

SCADA supervisory control and data acquisition

1 Introduction

In today's interconnected world, the proliferation of diverse and numerous devices has become increasingly common. This phenomenon is particularly evident in the field of industrial computing, which has experienced rapid growth. With the rapid expansion of the Internet of Things (IoT) and edge computing phenomena, the use of such devices in enterprises has reached enormous proportions and continues to grow unabated. The management and control of such a large number of intelligent industrial devices fall within the domain of Industrial Control System (ICS) 3.1. However, due to the large number of devices involved in an ICS, one of the biggest challenges organisations face is how to effectively monitor and diagnose them. The ability to accurately track and locate events and incidents is critical for organisations to avoid unexpected production disruptions, minimise costs and reduce overall time wasted. The findings of this thesis have profound implications for the industrial sector, where effective health monitoring of smart devices is essential to maintain optimal performance and prevent downtime.

In this chapter, we look at the problem of ICS monitoring and the direction we are taking to develop a solution.

1.1 Motivation

The problem

The integration of smart devices into industrial operations has indeed revolutionised the way the industry operates. These devices monitor their health in real-time through diagnostic logs, providing valuable insight into their performance and potential problems. However, the heterogeneous nature and distribution of these devices in industrial environments presents significant challenges when it comes to aggregating health information. While understanding the health status of individual devices may be relatively straightforward, achieving overall observability of an ICS is a challenging task. Industrial operations often involve a large number of heterogeneous devices from different manufacturers, each with its protocols, data formats and interfaces. These devices can include programmable logic controllers (PLCs), sensors, actuators, human-machine interfaces (HMIs) and more. Integrating and collecting data from these diverse devices can be complex, requiring careful consideration of compatibility and interoperability. An important consideration is that industrial plants are typically spread over large areas and consist of many subsystems and devices, such as production lines, assembly lines and processing units. These subsystems may be physically dispersed, making it difficult to collect health information from all devices in a centralised manner. In addition, the interdependencies and interactions between devices can have a profound effect on the overall functionality and resilience of the ICS. It is difficult to determine the overall health of an ICS based solely on the health of its devices. It is critical to consider how these devices are organised and interconnected within the system. This requires the intelligent combination and interpretation of metrics, logs and traces to assess both the health of individual devices and their collective behaviour within the system. A comprehensive assessment of ICS health should include both the individual device

states and their collective behaviour to ensure satisfactory system observability.

Our solution: DeMon++

This thesis aims to address the challenges of achieving interoperability between heterogeneous devices by developing a state-of-the-art monitoring framework named DeMon++ (Device Monitoring ++). The main objective of DeMon++ is to provide a versatile and adaptable solution for monitoring complex and widely distributed industrial devices. It acts as a higher-level abstraction layer that encompasses various pre-existing monitoring agents and facilitates the seamless integration and consolidation of monitoring information from different sources. By acting as a compatibility layer, DeMon++ streamlines the integration and consolidation of monitoring information from different monitoring agents. Merging diagnostic logs from disparate devices presents significant challenges, often making automation extremely difficult or even impossible. In response, DeMon++ enriches diagnostic logs with a state-based abstraction layer that serves as a unifying mechanism that allows devices to be monitored without being constrained by individual device specifications, thereby promoting a cohesive network environment. The process of transforming diagnostic logs into health states can be implemented via DeMon++ plug-ins or delegated directly to the monitoring agents themselves. In particular, the state-based abstraction layer is fully user-definable, allowing users to define states that exactly match the requirements of the monitoring agents, ensuring full compatibility. Another important aspect of DeMon++ is the possibility to organise monitored devices into logically related groups using a hierarchical tree structure model. This not only provides a higher level of representation that makes the health state of the system understandable and interpretable by non-specialists but also allows the application of the "divide et impera" concept. Specifically, this concept consists of partitioning the entire system, including the monitoring process, into smaller subsystems, known

in DeMon++ as domains. Each domain is monitored by a dedicated instance of DeMon++, resulting in a scalable, flexible and simplified monitoring process. In essence, this approach aims to provide a tool for creating and modelling a digital twin of the system with the specific purpose of achieving real-time intelligent system observability. The automation process is fully configurable through a user-defined logical set of conditional rules to automatically aggregate each device state and derive the overall system health.

The employed use-case

In order to develop an effective solution for monitoring ICSs, it is essential to identify a suitable use case. ICSs can range in size from small setups to large, complex systems with thousands of devices. In this context, the CERN use case is an ideal choice. In particular, the CERN environment is characterised by a vast and intricate network known as the CERN ICS. This complex network manages various critical systems such as cooling, ventilation, cryogenics, gas and electricity through thousands of different control devices. These devices include PLCs, sensors, front-end computers and supervisory control and data acquisition (SCADA) [1] systems. Their purpose is to control various aspects of the plant's operation, including experiments and data collection. However, the large number of devices and the complexity of their interactions make it difficult to understand what is happening in the ICS. As part of the CERN OpenLab [2] programme, this thesis provides a flexible and maintainable approach to monitoring complex and distributed ICS, with a particular focus on the demanding environment of CERN. This programme consists of a collaborative platform where CERN researchers and engineers work closely with experts from partner companies, including major players in the IT industry, for the benefit of the scientific and industrial communities. The collaboration between Siemens and CERN, culminating in this thesis, therefore addresses the challenge of

monitoring ICSs in such a complex and diverse environment as CERN. The aim is to develop an effective solution that can provide insight into what is happening, where and when within the ICS, despite its inherent complexity.

1.2 Organization

We begin with a discussion of related work in Chapter 2, which provides an overview of the literature on monitoring methods and the existing framework for monitoring ICSs. Chapter 3 defines the basic concepts needed to understand the development of the thesis, so that people without a strong background in these areas can easily understand the work. Chapter 4 explains the methods used to develop the DeMon++ architecture. Each software component that is part of the architecture is analysed in detail, as well as the relationships between them. Finally, a proof of concept implementation of this architecture is presented and supported by proper UML diagrams and code snippets derived from the built prototype. This is done to give the possibility to use this thesis, especially this chapter, as a DeMon++ software architecture documentation. In Chapter 5 we will present our proof of concept implementation and we are going to discuss possible directions for further development of this work. Starting with an overview of our findings, in which we discuss the strengths and weaknesses of our work, we will present a proof of concept implementation. Chapter 6 concludes the thesis with a summary of the final result, followed by a list of possible directions for future development.

1.3 Contributions

The main contributions of this thesis are:

- Analyses of the complex large scale ICSs composed of heterogeneous devices and identifying problems thereof.

- Design and development of software architecture for a flexible, scalable and maintainable monitoring framework.
- Implementation of a monitoring framework to abstract, aggregate and summarise the health state of large scale ICSs.
- Application of the implementation in the ICS at CERN.

2 Related Work

In this chapter, we review the existing related literature on monitoring frameworks for ICSs. We will also highlight their limitations in achieving interoperability and consolidating monitoring information from heterogeneous devices.

2.1 Similar Approaches to Monitoring Frameworks

Several monitoring frameworks have been proposed to address the challenges of device interoperability and information aggregation in monitoring. These frameworks aim to provide a higher level of abstraction that allows monitoring agents to collect and consolidate data from different devices, without being constrained by device-specific requirements, to provide a comprehensive model of the health of the system. A notable approach used at CERN is SMI++ [3], which views the real world as a collection of objects that behave as finite-state machines. These objects may represent real entities such as hardware devices or software tasks, or they may represent abstract subsystems. SMI++ supports hierarchical state machines, allowing the composition of smaller state machines to model complex behaviour in a modular and manageable way. It is also available for a wide range of platforms, making it suitable for distributed environments. However, SMI++ focuses primarily on ICS control rather than observability, which introduces complexity in state aggregation mechanisms and overhead for monitoring devices and inferring system health. Another project, Arrowhead [4], aims to bring device status information from heteroge-

neous field devices and sensors to enterprise-level applications and services using the OPC UA protocol. Similar to SMI++, Arrowhead proposes hierarchical aggregation and the use of the OPC UA protocol [5] to share information with higher-level nodes. It provides a good level of standardisation and compatibility with heterogeneous devices. In addition, it provides a general model for representing the aggregation of information, making it an attractive aspect that goes towards the same attempt that we are pursuing in this work. However, as they pointed out, device-specific type representations are not fully supported in the generic model, which may limit the full system observability we are aiming for. This limitation may be addressed in future developments. Other studies, such as Cloud-based standardised device diagnostics for optimised operability of plants in the process industry [6], propose moving device information to the cloud. They emphasise the feasibility of this approach with modern edge computing capabilities. They highlight the importance of a state-based abstraction for standardising device diagnostics, using the states defined in the Namur Ne107 standard [7]. This approach is interesting cause it allows diagnostic logs to be transformed into standard states, facilitating aggregation and providing a unified representation of system health. However, it only presents the rationale for moving device information to the cloud and using standard states, without providing a concrete approach neither to how to transform logs into states nor how to apply a state aggregation process. Despite this, we also think the Namur Ne107 standard is a suitable method for devices to outsource a simple interpretation of their logs. An interesting approach for achieving conditional heterogeneous device monitoring has been devised at Siemens AG, as outlined in [8]. This method involves a function block approach for recording and processing condition monitoring data uniformly. It supports both analogue and digital device signals, converting them into states using a threshold-based method. Eventually, information is assessed and aggregated across user-defined hierarchical levels, allowing for the grouping of de-

vices into digital entities known as blocks. This process contributes to the creation of an abstract model of the system, aligning with the objectives of our current work.

2.2 Interoperability in Heterogeneous Device Monitoring

In industrial environments, devices and equipment from different vendors are typically connected using different fieldbuses and protocols. This proliferation of technologies makes integration difficult and hampers automation efforts. In addition, system interfaces, device descriptions and provided data can vary, leading to a lack of interoperability and hindering comprehensive monitoring of system health. To address these challenges, researchers and practitioners have recognised the need for advanced monitoring frameworks. The CHARIOT project [9] proposes a scalable IoT middleware that enables the integration of heterogeneous IoT devices. It aims to achieve device interoperability by abstracting each device, making the platform agnostic to the underlying devices and protocols. While the approach provides compatibility with heterogeneous devices through an adapter protocol, it lacks scalability and sufficient focus on distributed environments. It also focuses on extrapolating information from heterogeneous devices rather than aggregating device health information and automatically deriving the overall system health state. Another approach to achieving interoperability between heterogeneous embedded devices is through the use of blockchain, as proposed in [10]. The study examines the benefits of using blockchain technology to ensure the reliability of information. However, blockchain has disadvantages such as high storage overhead, scalability issues and encryption processing time. Therefore, this approach may not be the most appropriate for the purposes of this work. In [11], a hardware-based method for monitoring heterogeneous devices is proposed. By using an additional hardware

component, observability of device metrics can be achieved with minimal impact on execution performance. However, this solution does not include a mechanism for automatically aggregating device information, which is necessary to achieve the desired goals. While these approaches focus primarily on data collection, they lack advanced mechanisms for consolidating monitoring information and inferring the health state of the system. However, a lot of methods from these works are really promising to be adopted. Moreover, some of them can also serve as suitable monitoring agents that DeMon++ could use to achieve compatibility with a wide range of devices.

3 Fundamentals

In this chapter, a brief introduction to computer science topics relevant to our studies will be provided. We will start with an overview of what is an ICS. After that, we outline the main concept needed to develop this thesis. In the end, we introduce the edge computing concept as, from our point of view, a suitable application for our monitoring tool.

3.1 Industrial Control Systems

In order to explain our method, it is necessary to introduce what we mean by ICS [12]. An ICS is a type of computer-based system that is used to manage and control industrial processes and machinery. The primary purpose of an ICS is to automate and optimize industrial processes by using sensors, controllers, and other devices to monitor and control various aspects of the system. This can include monitoring temperature, pressure, flow rate, and other variables, as well as controlling the operation of machinery and equipment.

As also described in the figure 3.1, ICSs typically consist of four main layers:

- `Enterprise Layer:` The enterprise layer encompasses the integration of the ICS with higher-level business processes and systems. It involves the exchange of data and information between the ICS and enterprise resource planning (ERP) systems, asset management systems, supply chain management systems, and other business applications.

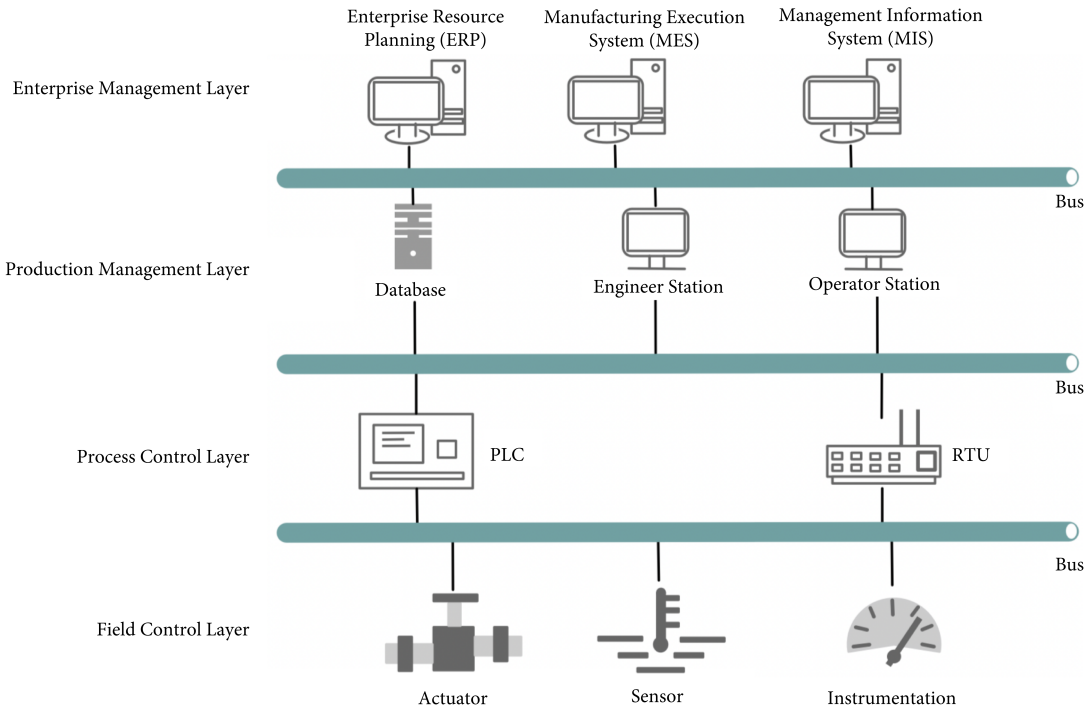

Figure 3.1: A typical architecture of ICS [13]

- `Supervisory Layer:` The supervisory layer consists of software and hardware that enables operators to monitor and control the industrial processes in real-time. It includes human-machine interfaces (HMIs), operator workstations, and SCADA [1] systems.

- `Control Layer:` The control layer includes the PLCs, which receive sensor data from the field layer and issue control signals to the field layer devices. The control layer also includes other devices such as remote terminal units (RTUs) and distributed control systems (DCSs) that perform similar functions.

- `Field Layer:` This layer includes sensors, actuators, and other devices that directly interact with the physical process being controlled. The sensors collect data about the process, and the actuators modify the process in response to control signals.

It's important to note that an ICS may vary depending on the specific archi-

tecture and requirements of the system. For this reason, we want to provide an overview of the CERN ICS since, as already mentioned in the Introduction, it is the use-case we have chosen.

3.1.1 The CERN Industrial Control System

The CERN ICS consists of a complex network comprising thousands of heterogeneous control devices, including programmable logic controllers, front-end computers, supervisory control and data acquisition systems.

Some examples of ICS at CERN include LHC Cryogenics, machine protection systems such as QPS (Quench Protection System), WIC (Warm Interlock Controller), and PSEN (Powering Safety Elements), electrical systems such as POPS (Powering of Superconducting Circuits), HVAC (Heating, Ventilation, and Air Conditioning) and cooling systems for both the technical infrastructure and experiments, gas control systems for experiments such as LHC GCS (Gas Control System) and primary gas systems such as EAPGS (Experimental Area Primary Gas System), and control systems for experiments such as DSS (Detector Safety System) [14].

3.2 Fundamentals of Industrial Protocols

Industrial protocols serve as the communication backbone of ICSs, enabling the exchange of data and control commands between devices, controllers, and other components within the system. Understanding the fundamentals of industrial protocols is crucial for designing, implementing, and managing efficient communication networks in industrial environments. In particular, they define the rules and standards for data exchange, device control, and system coordination. Industrial protocols facilitate interoperability and synchronization among heterogeneous devices, enabling seamless integration and coordination of industrial processes. Before introducing the

most used IoT devices intra-communication protocols, it is necessary to introduce the various existing communication models:

- `Point-to-Point:` The point-to-point communication model involves direct communication between two devices, typically a master and a slave. In this model, the master device initiates communication and controls the data exchange with the slave device. It is commonly used in protocols like Modbus, where a master device communicates with one or more slave devices.

- `Publisher-Subscriber:` The publisher-subscriber communication model, also known as the publish-subscribe model, involves data distribution from publishers to multiple subscribers. Publishers generate and distribute data, while subscribers receive and consume the data they are interested in. This model is commonly used in protocols like OPC UA, MQTT, and DDS, allowing for flexible and scalable data exchange in industrial systems.

- `Client-Server:` The client-server communication model involves a client device requesting data or services from a server device. The client sends a request, and the server processes the request and sends back the response. This model is used in protocols like HTTP (Hypertext Transfer Protocol) and RESTful APIs for communication between clients (e.g., web browsers or mobile apps) and servers (e.g., cloud-based systems or web servers). While not specific to ICSs, this model can be adopted in certain industrial applications for data exchange with external systems.

3.2.1 Industrial Devices Communication Protocols

To meet the evolving needs of industrial communication, several protocols that offer advanced features and capabilities have been developed based on the communication model just introduced. Some widely used high-level industrial protocols are

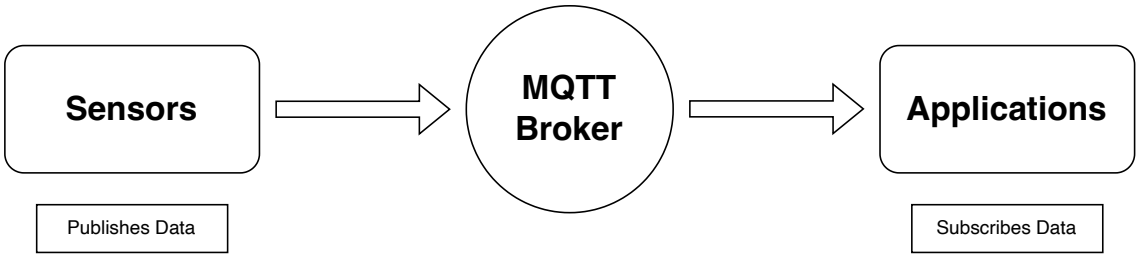

Figure 3.2: The MQTT protocol

discussed below:

OPC UA (Unified Architecture)

OPC UA [5] is a platform-independent industrial protocol that provides secure and reliable communication in industrial environments. It offers a standardized and interoperable framework for data exchange, device control, and information modelling. OPC UA supports a wide range of platforms, including Windows, Linux, and embedded systems, making it suitable for diverse industrial applications. It incorporates modern security features, including encryption and authentication, to ensure the integrity and confidentiality of data.

MQTT (Message Queuing Telemetry Transport)

MQTT [15] is a lightweight publish-subscribe messaging protocol designed for efficient and reliable communication in constrained environments, such as industrial IoT. As also represented in figure 3.2, it follows a publisher-subscriber communication model, where publishers send messages to a central broker, and subscribers receive relevant messages based on their subscriptions. MQTT's lightweight design minimizes network bandwidth and device resource requirements, making it ideal for industrial applications with limited resources. It supports various Quality of Service (QoS) levels, enabling flexible trade-offs between message delivery reliability and

network overhead.

CoAP (Constrained Application Protocol)

CoAP [16] is a lightweight application-layer protocol designed for constrained devices and networks, particularly in the context of IoT applications. It enables resource-constrained devices to communicate over the internet using RESTful principles. CoAP supports efficient request-response interactions, resource discovery, and group communication. It is suitable for industrial scenarios that require low-power and low-bandwidth communication, such as smart building automation and industrial sensor networks.

DDS (Data Distribution Service)

DDS is a data-centric publish-subscribe protocol designed for real-time and scalable communication in distributed systems. It provides a high-level abstraction for data-centric publish-subscribe communication, enabling the exchange of large amounts of data with low latency and high reliability. DDS supports various Quality of Service policies, such as reliability, durability, and deadline enforcement, to meet the requirements of different industrial applications. It is commonly used in industries such as aerospace, defence, and healthcare, where real-time data sharing and system scalability are critical.

3.3 Data Collection and Acquisition

This section focuses on the fundamentals of data collection and acquisition in ICSs. It covers the types of data collected, the sensors and devices used for data acquisition, and the methods employed for acquiring data. An overview of the types of data, sensor devices and methods of data acquisition has been provided.

3.3.1 Types of Data

ICSs collect various types of data to monitor and control industrial processes effectively. Some common types of data include:

- `Process Data:` Process data represents the parameters and variables that describe the state and behaviour of industrial processes. It includes measurements such as temperature, pressure, flow rate, voltage, and current. Process data provides insights into the operation and performance of industrial systems, enabling effective monitoring and control.

- `Diagnostic Data:` Diagnostic data encompasses information related to the health, status, and diagnostics of industrial equipment and devices. It includes data such as error codes, fault indicators, system logs, and diagnostic measurements. Diagnostic data enables proactive maintenance, troubleshooting, and fault detection in industrial systems.

- `Event Data:` Event data captures significant occurrences or incidents in industrial processes. It includes events such as alarms, system events, operator actions, and process state transitions. Event data is crucial for analyzing system behaviour, identifying anomalies, and investigating critical events in ICSs.

3.3.2 Sensors and Devices

Various sensors and devices are employed for data collection in industrial environments. These sensors and devices are specifically designed to measure and monitor physical quantities and phenomena. Some commonly used sensors and devices include:

- `Temperature Sensors:` Temperature sensors measure the ambient or process temperature in industrial applications. They are crucial for monitoring

thermal conditions and ensuring optimal operating parameters in industrial processes.

- `Pressure Sensors:` Pressure sensors measure the pressure levels of liquids or gases in industrial systems. They are used to monitor and control pressure conditions, detect leaks, and ensure safe and efficient operation of industrial equipment.

- `Flow Sensors:` Flow sensors measure the flow rate of liquids or gases in industrial processes. They are essential for monitoring and controlling fluid flow, optimizing energy consumption, and ensuring the proper functioning of industrial systems.

- `Level Sensors:` Level sensors measure the level of liquids or solids in tanks, containers, or vessels. They provide information about the fill levels, enabling effective inventory management, process control, and preventing overflows or shortages.

- `Control Devices:` Control devices include actuators, valves, switches, and relays that are responsible for controlling industrial processes based on the acquired data. These devices receive control signals and actuate physical changes in the system, such as opening or closing valves, starting or stopping motors, or adjusting process parameters.

3.3.3 Methods of Data Acquisition

Data acquisition involves the process of collecting and digitizing data from sensors and devices. Several methods are used for data acquisition in ICSs. Some commonly used methods include:

- `Analog-to-Digital Conversion (ADC):` Analog-to-digital conversion is a fundamental method for converting analog sensor signals into digital format.

ADCs sample the analog signal at regular intervals and convert it into discrete digital values that can be processed and stored by digital systems. ADCs are widely used for acquiring data from analog sensors in ICSs.

- `Fieldbus and Industrial Ethernet:` Fieldbus and Industrial Ethernet protocols provide standardized communication interfaces for data acquisition from sensors and devices. These protocols enable efficient and real-time data exchange between sensors, actuators, and controllers. Examples of fieldbus protocols include PROFIBUS, Modbus, and DeviceNet, while Industrial Ethernet protocols include PROFINET, EtherCAT, and Ethernet/IP.

- `Wireless Data Acquisition:` Wireless data acquisition methods utilize wireless communication technologies to transmit data from remote sensors and devices. Wireless protocols such as Zigbee, WirelessHART, and Bluetooth are used to establish communication links and collect data from sensors located in challenging or inaccessible areas. Wireless data acquisition provides flexibility and mobility in industrial environments.

3.4 Design of the Monitoring Tool

For designing a robust and efficient monitoring tool, choosing an appropriate architecture style and employing a suitable evaluation method is crucial. This section provides an overview of different software architecture styles pointing out which one has been chosen and why. Lastly, an evaluation method for the chosen architecture has been defined.

3.4.1 Architectural Styles

Software architectural styles provide guidelines and patterns for organizing components, defining communication mechanisms, and managing system behaviour. Pop-

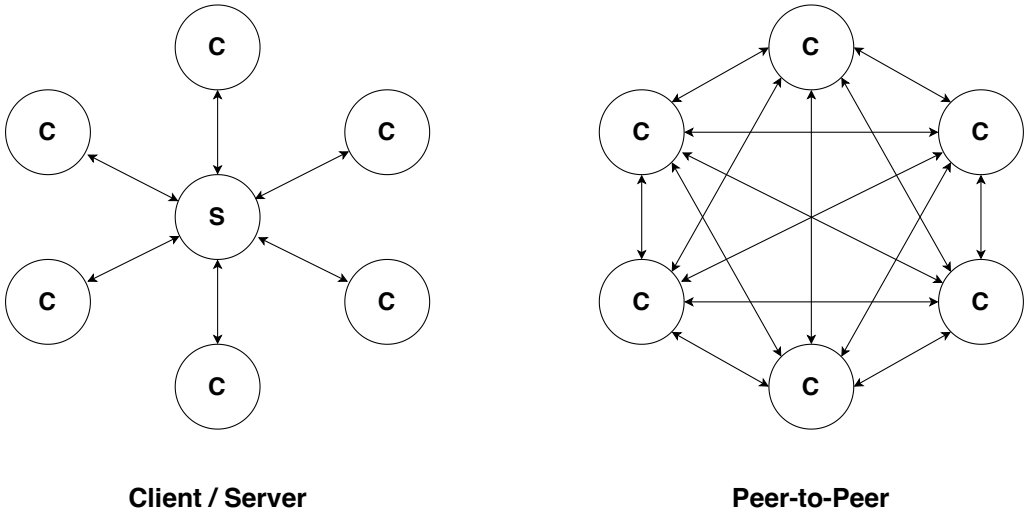

Figure 3.3: Client-Server and Peer-to-Peer Service-Based Architectural Styles

ular architectural styles are:

Partitioning Architectural Styles

Partitioning architectural styles focuses on dividing the system based on structure or functionality:

- `Layer/Tier Architectures:` This style divides the system into distinct layers or tiers based on their functionalities and responsibilities. It promotes modularity, separation of concerns, and ease of maintenance.

- `Pipes and Filters Architecture:` This style decomposes a system into sequential processing stages connected by pipes. Each stage, known as a filter, performs a specific function and passes the processed data to the next filter. It emphasizes the functional decomposition and reusability of filters.

Service-Based Architectural Styles

Service-based architectural styles focus on the interaction and communication between components:

- `Client-Server Architectures:` As can also be seen in the left part of the figure 3.3, this style divides a system into client and server components, where clients request services from servers. It enables distributed computing, scalability, and separation of presentation and business logic.

- `Peer-to-Peer Architecture:` As also shown on the right side of the figure 3.3, this style allows equal participation of all nodes in a network. It enables direct communication and collaboration between peers without relying on a centralized server. It supports decentralization, fault tolerance, and scalability.

- `Message-Passing Architectures:` This style involves components communicating through messages. Publishers send messages to specific topics or channels, and subscribers receive relevant messages based on their subscriptions. It provides loose coupling, event-driven communication, and scalability.

Special Architectures

Special architectures address specific design patterns or requirements. Some of the most popular are:

- `MVC Architectures:` This style separates an application into three interconnected components: the model, the view, and the controller. It promotes separation of concerns, modularity, and flexibility in designing user interfaces and handling user input.

- `Interpreter Architectures:` This style allows for the interpretation of user-defined rules to define system behaviour dynamically. It separates the rule interpretation logic from the core functionality of the system, providing flexibility in adapting the system's behaviour based on specific requirements.

Chosen Architectural Style

In this thesis, the chosen software architecture style for the monitoring tool is the interpreter one, from Special Architectures. This architectural style has been selected since it provides the necessary flexibility to let the user define the behaviour of the monitoring tool by interpreting the user-defined configuration file (System Descriptor File). The Interpreter Architecture allows for the dynamic interpretation of aggregation rules, which can be expressed in a domain-specific language or a configurable format. This flexibility enables users to define and customize the monitoring tool's behavior based on specific requirements, without the need for extensive software development or recompilation. Moreover, the monitoring tool can adapt to various ICS environments, accommodating different Monitoring Agents and industrial devices.

3.5 Edge Computing

Edge computing has gained significant popularity in recent years, revolutionizing data processing and analysis. The term edge computing refers to the paradigm of processing and analyzing data closer to where it is generated, typically at the network edge. This approach stands in contrast to traditional centralized computing models where data is transmitted to a remote data centre or cloud for processing. Edge computing leverages distributed infrastructure and intelligent devices to bring computing resources closer to the point of data generation, reducing latency, improving reliability, and enhancing security.

In light of the significant popularity and transformative impact of edge computing, it is crucial to explore how our project aligns seamlessly with the Siemens Industrial Edge ecosystem.

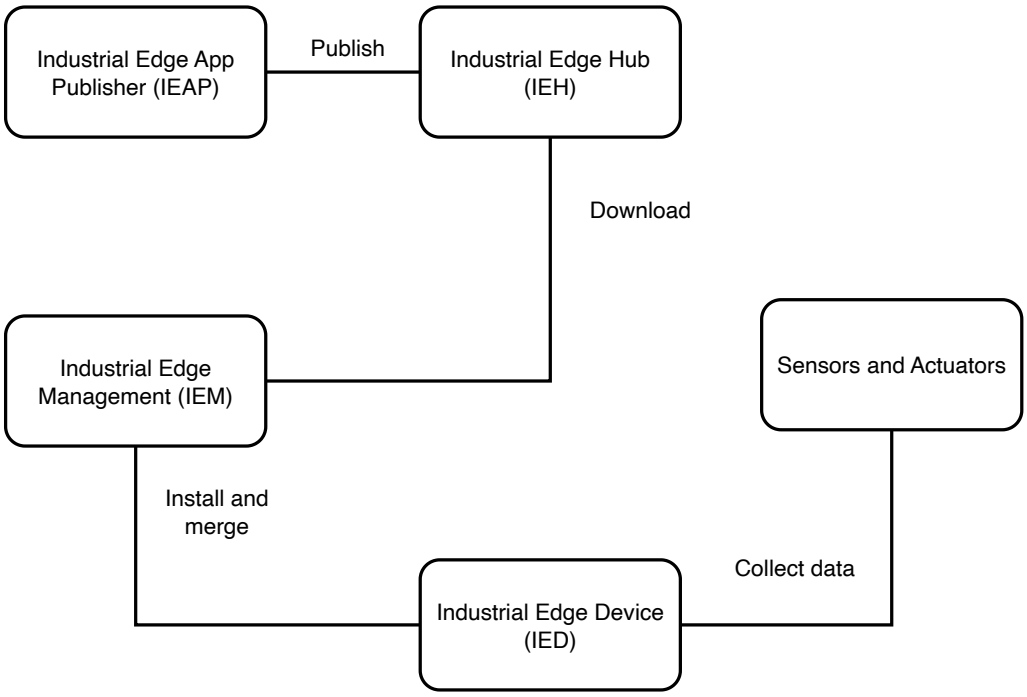

Figure 3.4: Siemens Industrial Edge Overview

3.5.1 Siemens Industrial Edge Ecosystem

According to Siemens documentation [17], Siemens has developed their own Industrial Edge ecosystem, which as illustrated in the figure 3.4, includes several components to enable the deployment and management of edge applications in industrial environments. These components are:

- `Industrial Edge Hub (IEH)`: The Industrial Edge Hub is the central cloud platform that serves as a central platform for deploying and managing edge applications. Designed to operate in harsh industrial environments, it provides a secure and scalable infrastructure for running edge applications. The Industrial Edge Hub is based on standard industrial hardware, including an industrial-grade CPU, memory, storage, and connectivity options.

- `Industrial Edge Management (IEM)`: The Industrial Edge Management system is a software platform that enables the centralized management of Indus-

trial Edge devices and applications. It provides tools for deploying, configuring, and monitoring edge applications, as well as managing security and software updates. The Industrial Edge Management system is designed to be easy to use and can be integrated with existing industrial automation systems.

- `Industrial Edge App (IEA)`: The Industrial Edge Applications are software applications designed to run on Industrial Edge Devices, allowing for the implementation of various use cases and business models in industrial environments. These applications leverage the power of Edge Computing to enable real-time data processing and analytics, as well as local decision-making capabilities. Industrial Edge Applications can be easily deployed, updated, and managed through the Industrial Edge Management system, and can be customized to meet the specific needs of individual industrial use cases. Examples of Industrial Edge Applications include condition monitoring, predictive maintenance, quality control, and energy management, among others.

- `Industrial Edge App Publisher (IEAP)`: The Industrial Edge App Publisher is a software tool that enables developers to create and package edge applications for deployment on the Industrial Edge Hub. It provides a graphical user interface for creating and configuring edge applications and can generate packages that can be deployed on the Industrial Edge Hub. The Industrial Edge App Publisher supports a wide range of programming languages and frameworks, including C++, Python, and Java.

- `Industrial Edge Device (IED)`: Finally, the Industrial Edge Devices are a crucial component of the Siemens Industrial Edge ecosystem and are designed to work seamlessly with the so-called Field Devices, such as sensors and actuators. These edge devices are responsible for gathering and processing data at the network edge in industrial environments. Industrial Edge Devices are

designed to be compatible with the Industrial Edge Hub and can be easily integrated with the Industrial Edge Management system. In fact, the Edge Devices are the actual devices on which the individual Edge applications, and of course, DeMon++, will be executed.

Together, these components provide a comprehensive solution for deploying and managing edge applications in industrial environments. The Siemens Industrial Edge ecosystem is designed to be flexible, scalable, and easy to use, making it an ideal platform for industrial automation and digital transformation.

4 Methods

This chapter discusses the methods that are going to compose a novel monitoring framework, DeMon++. After a first overview of system requirements, we are going to explain deeply the methods involved in our work.

4.1 System Requirements

In order to make this method practical in industrial environments, certain key requirements must be met. In this section, we are going to outline why and what requirements DeMon++ wants to satisfy.

- **Flexibility:** Industrial operations are characterized by dynamic and evolving environments with frequent changes in device configurations, network topologies, and operating conditions. The monitoring system must be flexible enough to adapt to these changes without compromising its effectiveness.

- **Maintainability:** The monitoring system in industrial environments should be easy to maintain and update. Downtime can result in significant financial losses, so the system should seamlessly integrate advancements in smart devices and new technologies.

- **Scalability:** Industrial environments often expand or change over time, with the addition of new devices or modifications to existing infrastructure. The

monitoring system should accommodate scalability requirements without requiring significant reconfiguration or redevelopment. It should be capable of scaling up to thousands of devices.

- **Compatibility and Extensibility:** The monitoring system must prioritize compatibility and extensibility with the wide range of devices found in the industrial world. It should effectively communicate with different types of devices, support industry-standard protocols, and adapt to device-specific requirements and configurations. The system should also be easily extensible to incorporate new compatibility features and support additional protocols for achieving interoperability.

- **Human Expertise:** Apart from technical complexities, the human factor plays a crucial role in interpreting the health status of monitored devices. Personnel involved in the operation and maintenance of the monitoring system should possess specialized knowledge and skills to accurately understand and analyze the behaviour and potential problems of industrial equipment. This includes expertise in device-specific diagnostics, protocols, and operating characteristics, enabling prompt action in critical situations. With DeMon++ we want to overcome human expertise by reducing complexity and consequently increasing the system observability accessibility.

By addressing these requirements, the monitoring system can be made practical and effective in industrial environments.

4.2 Use-Case

In this section, we present the use cases for our monitoring tool by exploring the activities and benefits of using it.

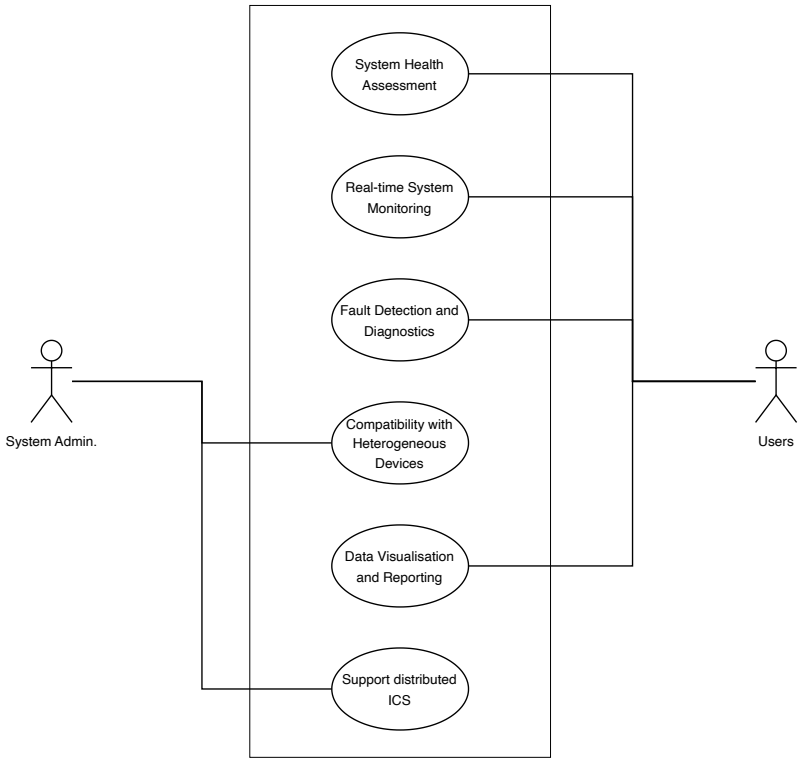

Figure 4.1: Monitoring tool - Use-case Diagram

The increasing complexity and size of ICSs have highlighted the need for a robust monitoring tool. The reasons why such a tool is essential are varied. As shown in the 4.1 use case diagram, we have highlighted the main use cases for why such a tool is needed:

- `System Health Assessment:` The tool should be able to assess the overall health and performance of the ICS. It should aggregate information from multiple devices and provide a comprehensive view of the health of the system, highlighting any anomalies.

- `Real-time System Monitoring:` The monitoring tool continuously collects and analyzes real-time data from devices, sensors, and components within the ICS. It provides operators with a live view of the system's performance, allowing them to identify any abnormal conditions, deviations from set points, or potential failures.

- `Fault Detection and Diagnostics:` The monitoring tool detects and diagnoses faults within the ICS. It analyses data patterns, compares them to pre-defined thresholds and triggers alerts or notifications when anomalies or faults are detected. This helps operators to take immediate corrective action and minimise system downtime.

- `Compatibility with heterogeneous devices:` ICSs are often made up of different devices from different manufacturers, each with its own communication protocols and interfaces. The monitoring tool should be compatible with a wide range of devices, support different protocols and ensure seamless integration and data collection.

- `Data visualisation and reporting:` The tool should provide intuitive data visualisation capabilities that present system status, trends and performance metrics in a clear and accessible manner. It should also provide reporting capabilities that allow operators to generate comprehensive reports for analysis, audit and compliance purposes.

- `Support distributed ICS:` The monitoring tool should provide support for such a distributed ICS context as the CERN case study documented in the The CERN Industrial Control System section.

4.3 System Overview

In this section, we introduce an overview of the DeMon++ system, which is inspired by ideas and concepts presented in the above use-case 4.2.

A use-case diagram, depicted in Figure 4.2, illustrates the interaction between users and the DeMon++ system. The use-case involves multiple actors categorized into three groups: "Users", "System Administrators", and "Monitoring Agents".

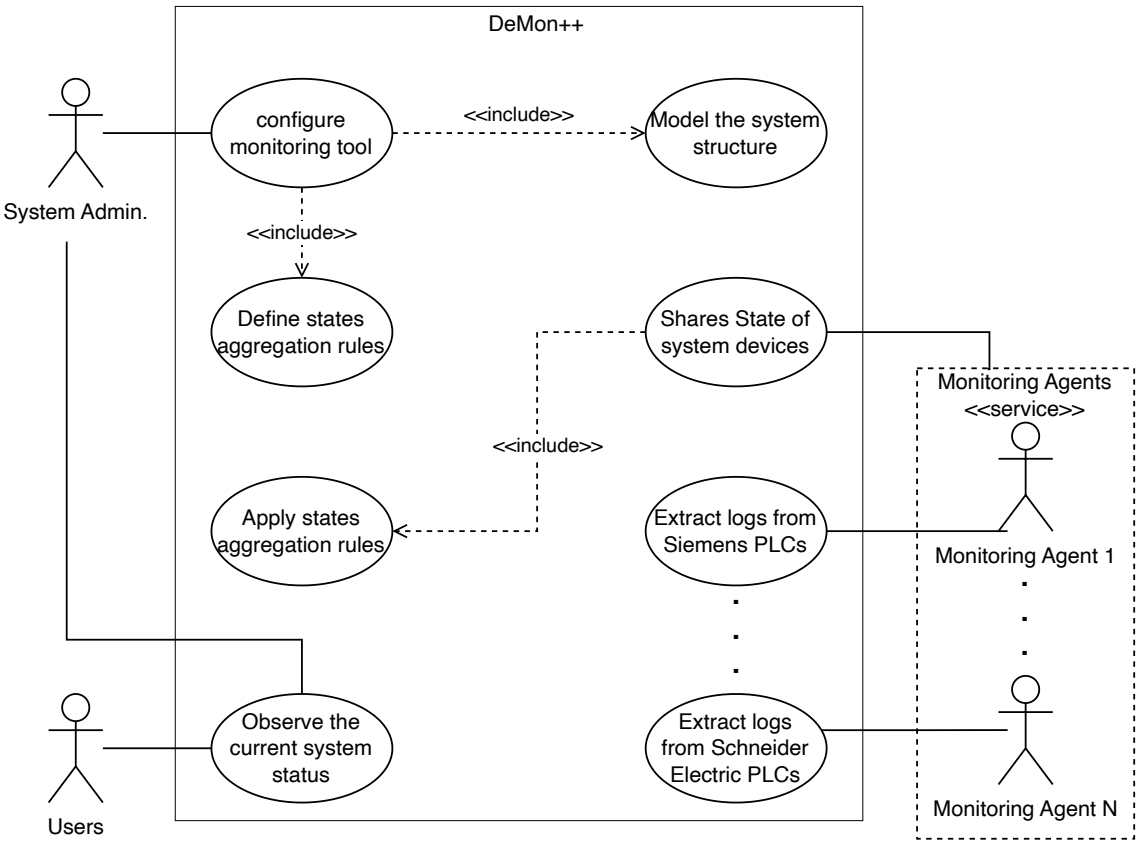

Figure 4.2: DeMon++ - Use-case Diagram

Employees, who are CERN personnel, rely on acquiring knowledge about ongoing events through a Graphical User Interface which represents the CERN ICS topology. This ensures the smooth progress of experiments and the accuracy of results. As already mentioned, the primary use is to check the overall system state and the status of individual machinery, but additionally, the system allows employees to temporarily disable the monitoring of specific devices to facilitate equipment maintenance, improvements, or repairs.

System administrators, in addition to the functions performed by employees, have the responsibility of configuring and defining the system topology via system configuration files. Through this configuration process, administrators can construct a digital model of the system and define logical rules for automatically aggregating device states. A comprehensive explanation of this aforementioned file will be provided later in this chapter in System Descriptor File section.

Lastly, monitoring agents constitute another group of actors, representing a collection of tools responsible for extracting diagnostic logs from actual devices. Due to device heterogeneity, multiple types of monitoring agents may be required to support different devices from various manufacturers. Moreover, to achieve compatibility and interoperability with heterogeneous devices, a standard language is needed to interpret different diagnostic logs uniformly. Therefore, we opted to abstract diagnostic logs into device states as a common communication layer to summarize and outsource device health information. Thus, monitoring agents are also responsible for converting the acquired diagnostic logs into device states, which are then shared with the DeMon++ system. The DeMon++ system applies user-defined first-order logical rules to automatically generate an overview of the entire system state.

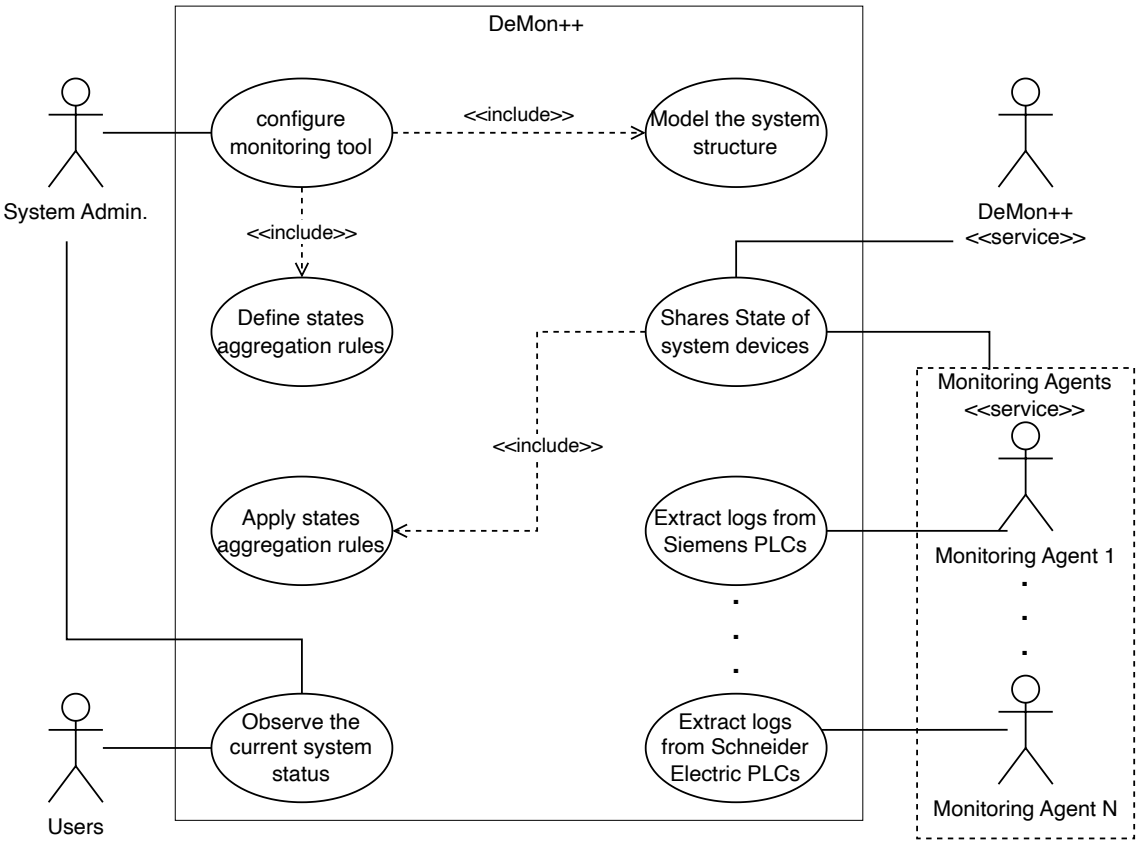

Figure 4.3: Distributed DeMon++ - Use-case Diagram

4.3.1 Distributed ICS use-case

As a distributed industrial system can be fragmented and distributed into several hierarchical structured sub-systems, a DeMon++ instance can also be fragmented and distributed accordingly. This helps for more reliable monitoring in addition to a more spread of the computation load among multiple devices and the consequently decreasing of computation resources.

From the perspective of DeMon++, an ICS is referred to as a monitored domain.

Therefore, as also depicted in Figure 4.3, the use-case can also be extended to support distributed ICS domains. In this scenario, a new actor, the DeMon++ service, comes into play. This actor represents one or more DeMon++ sub-domains,

which, similar to the monitoring agents, share the state of their respective domain devices. This feature allows for a comprehensive overview of the entire distributed system. A more in-depth analysis of how this capability is handled by DeMon++ will be presented later on DeMon++ sub-domains.

4.4 Software Architecture

After conducting an extensive analysis of the proposed use-case presented in the System Overview section, it has become evident that the adoption of architectural styles capable of accommodating dynamic and evolving requirements is necessary. Therefore, we propose the utilization of an interpreter architectural style. As also described in the section Special Architectures, we think it is a promising approach to address the challenges related to the flexibility and extensibility of our monitoring tool. This architectural style offers a modular and adaptable framework, facilitating the efficient interpretation and execution of domain-specific languages (DSLs) or business rules. By implementing this architectural style, our objective is to enhance the agility, maintainability, and scalability of software systems, thereby enabling organizations to effectively respond to evolving business needs. This section provides a comprehensive outline of the DeMon++ architecture, detailing each of its components extensively. The design rationale behind these components will be explained in order to fulfil the aforementioned functionalities.

In Figure 4.4, all the DeMon++ components and their relationships are outsourced. In particular, DeMon++ is composed of four main components:

- System Description File.
- State Manager.
- External Services.

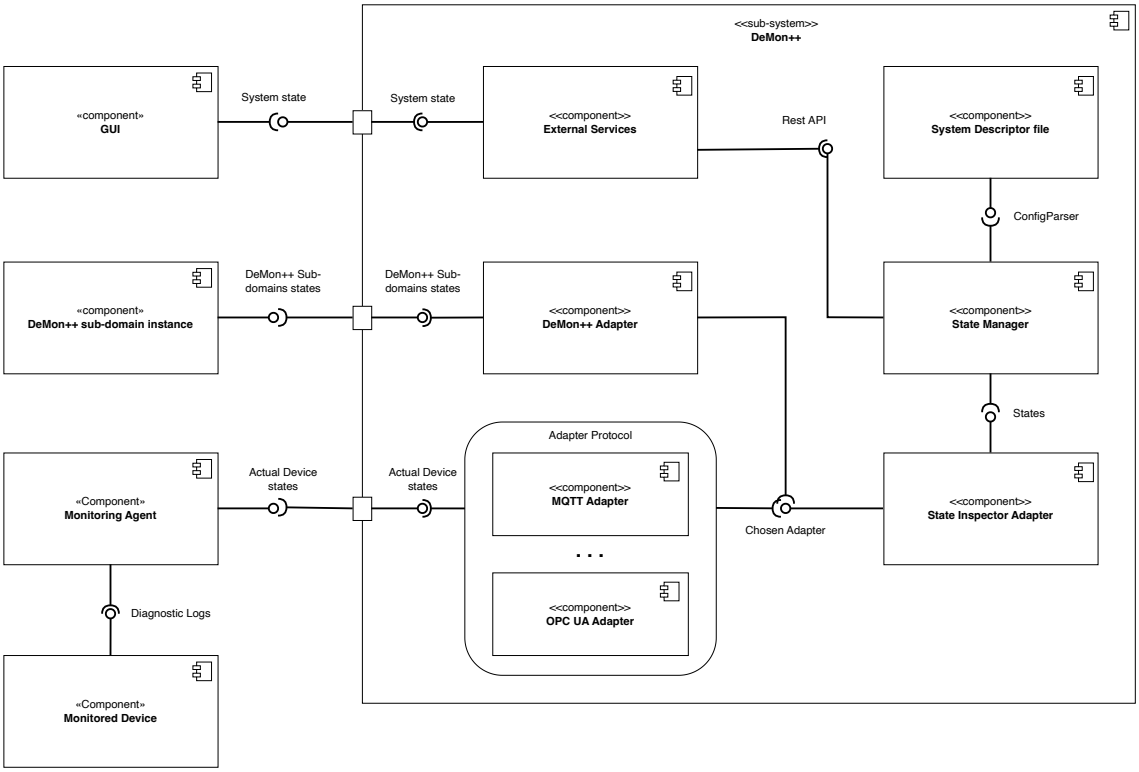

Figure 4.4: DeMon++ - Component Diagram

- State Inspector Adapter.

Each of the aforementioned components provides one specific functionality, and combined, they fulfil the System Requirements and accomplish the overall set of contributions made by this thesis.

4.4.1 System Descriptor File

Regarding the "System Description File", its functionality is to instruct the State Manager component, on how the system is organized and which rules to apply in order to automatically aggregate device states. This component enables the description and organization of different system devices hierarchically. Hence, it allows for the definition of the hierarchical structure of system devices following a tree-like model, with each device assigned to a specific node in the tree. Consequently, the

```
{
    "domain": ...,
    "structure": {
        ...
    },
    "adapters": {
        ...
    },
    "templates": {
        ...
    }
}
```

Figure 4.5: System Descriptor File Snippet: Sample

states of the various devices need to be automatically converted into a summarized state, which represents the node's own state. To achieve this, it is essential to describe a specific behaviour for choosing the group state. The purpose of this component is to provide the user with the ability to configure specific behavioural rules that dictate how a node must choose its own state based on the states of the belonging devices. To define static data for configuring DeMon++, we have opted for JSON [18], a lightweight data-interchange language. We chose JSON primarily for its human readability and parsing compatibility in almost all programming languages.

Regarding the System Descriptor File, we have provided a snippet sample in Figure 4.5. As reported in the figure, this file has been designed as a JSON object composed of four main properties:

- `domain`
- `structure`
- `adapters`

- `templates`

structure

```
"structure": {
    "node_1": {
        "template": "logic_node",
        "inputs": [
            "plc_1",
            "plc_2"
        ]
    },
    "plc_1": {
        "template": "siemens_s7_1200",
        "adapter": "mqtt",
        "id": "192.168.1.1"
    },
    "plc_2": {
        "template": "schneider_electric_1",
        "adapter": "http",
        "id": "192.168.1.2"
    }
}
```

Figure 4.6: System Descriptor File Snippet: structure property

One of the most important uses of the System Descriptor File component is the ability to digitally represent the model of an actual system. To achieve this, the system structure and its inherent relationships must be defined. This aspect is covered by the aforementioned JSON property `structure`, also shown in Figure 4.6. In this JSON object, the values are objects that differ based on the type of the device while the object name represents the device name. These objects can be grouped into two different kinds of devices:

- Actual devices.
- Digital devices.

When represented as a tree, the actual devices are the leaves of the tree, while the digital devices are the tree nodes.

```
"plc_1": {
    "template": "siemens_s7_1200",
    "adapter": "mqtt",
    "id": "192.168.1.1"
}
```

Figure 4.7: System Descriptor File Snippet: actual device properties

A snippet of the actual device properties is depicted in Figure 4.7. Focusing on this object, three properties are supposed to be defined:

- `template`
- `adapter`
- `id`

The `template` property is used to specify which template to apply to this particular device. More details about the templates are explained later in the specific `templates` section (4.4.1). The `adapter` defines which adapter protocol a device should use to establish a connection with a specific monitoring agent. The various available protocols are defined in the `adapters` specific section (4.4.1). Lastly, the `id` property defines a unique identifier that represents the actual device. This identifier can be an IP address, UUID, or any other unique identifier. Its purpose is to synchronize the digitally represented devices with the actual devices connected to monitoring agents.

Similar to the actual devices, a snippet of a digital device is provided in Figure 4.8. As a node, a digital device represents an abstraction of a group of devices. Its

```
"node_1": {
    "template": "logic_node",
    "inputs": [
        "plc_1",
        "plc_2"
    ]
}
```

Figure 4.8: System Descriptor File Snippet: digital device properties

behaviour is the same as an actual device, except for the fact that its state is not provided by a monitoring agent. Instead, it is determined by its own sub-devices. These sub-devices can be both actual and digital devices or a mix of them. The digital device object properties are:

- `template`
- `inputs`

The `template` property works the same way as it does for actual devices, while the `inputs` one consists of a list of values that identify the names of the sub-devices belonging to the node. Through this property, the system structure and relationships among its devices are defined.

templates

The system descriptor file includes a feature called "templates", which allows for defining a set of properties that are common to multiple devices. This feature aims to improve system scalability and facilitate the system configuration process by the administrator. Here is an explanation of the template properties:

```
"templates": {
    "siemens_s7_1200": {
        "starting_state": "Faulty",
        "states": [
            "Success",
            "Faulty"
        ],
        "rules": [...]
    },
    ...
}
```

Figure 4.9: System Descriptor File Snippet: template properties

The value of the template property is an object in which, each property defines the template name. Within each template, a three properties object is defined. The properties are:

- `starting_state`: Specifies the state to be used at system startup.
- `states`: Indicates the states supported by the device.
- `rules`: Defines the aggregation rules using a syntax called JsonLogic [19]: a small, safe way to delegate decision-making. It is important to underline it is not a full programming language but it leverages JSON syntax to offer deterministic computation. The "rules" property consists of a list of objects that define the following two properties:
  - `rule_logic`: Represents the logical rules written in JsonLogic that the State Manager must follow.
  - `out_state`: Specifies the output state that satisfies the defined rules.

Here is a snippet representing the template rule properties:

```
"rules": [
    {
        "rule_logic": {
            "some" : ["@input_states", { "===":[ {"var":""}, "@undefined_state" ] }]
        },
        "out_state": "@undefined_state"
    },
    {
        "rule_logic": {
            "===":[ "@this_state", "Success" ]
        },
        "out_state": "Success"
    },
    {
        "rule_logic": {
            "===":[ "@this_state", "Faulty" ]
        },
        "out_state": "Faulty"
    }
]
```

Figure 4.10: System Descriptor File Snippet: template rule properties

Additionally, the system descriptor file includes several keywords that are denoted by the "@" symbol. These keywords include:

- `@undefined_state`: Refers to a non-defined state, which can be useful for identifying and handling situations where no state has been provided by the monitoring agent yet.
- `@this_state`: Represents the current device state.
- `@input_states`: Defines the current states of the sub-devices.

These keywords provide additional context and information for defining the behaviour and relationships within the system. More details regarding how these rules are applied for determining states aggregation are explained in System Device States Aggregator.

```
"adapters": {
    "mqtt": {
        "plugin": "mqtt_plugin",
        "domain": "192.168.4.1"
    },
    "http": {
        "plugin": "http_plugin",
        "domain": "192.168.4.2"
    }
}
```

Figure 4.11: System Descriptor File Snippet: adapter properties

adapters

As mentioned in the section System Requirements, DeMon++ aims to achieve maximum compatibility and extendibility. To establish connections with actual monitored devices, it leverages several services called Monitoring Agents. To configure the support with monitoring agents, a property called `adapters` needs to be defined. This property contains a set of objects, where each object specifies a different adapter. Figure 4.11 illustrates this structure. In particular, the `adapters` properties represent the adapters' names, while the value defines an object that specifies the adapter settings. The properties of the adapter object include:

- `plug-in`
- `domain`

The `plug-in` property specifies which DeMon++ plug-in to use for supporting a specific connection protocol. More details regarding plug-ins will be explained in the specific section State Inspector Adapter. The `domain` property defines the specific IP address of the monitoring agent to use for retrieving device states. This property

is referenced as "domain" and not "ip" because it could also refer to DeMon++ sub-domains.

domain

```
"domain": "0.0.0.0:8000"
```

Figure 4.12: System Descriptor File Snippet: domain

Beyond monitoring agent adapters, DeMon++ also allows the specification of sub-domains. A sub-domain in DeMon++ can be configured similarly to a monitoring agent, with the key difference being that the state it shares is not that of an actual device, but rather the state of the root node of the sub-domain. The domain IP of a DeMon++ instance can be specified using the `domain` property, as shown in Figure 4.12. Consequently, to establish a connection between a DeMon++ domain and its sub-domain, the `domain` property value of the sub-domain is used in the adapters domain property of the root domain instance. More information regarding DeMon++ sub-domains can be found in the specific section DeMon++ sub-domains.

Finally, a complete sample snippet of a system descriptor file is outlined in Figure 4.13. As its name suggests, the "System Descriptor File" is just a file. To make it operational, this file needs to be read and interpreted. This is where the role of the "State Manager" component comes into play State Manager.

```
{
    "domain": "0.0.0.0:8000",
    "structure": {
        "node_1": {
            "template": "logic_node",
            "inputs": [
                "plc_1",
                "plc_2"
            ]
        },
        "plc_1": {
            "template": "siemens_s7_1200",
            "adapter": "mqtt",
            "id": "192.168.1.1"
        },
        "plc_2": {
            "template": "schneider_electric_1",
            "adapter": "http",
            "id": "192.168.1.2"
        }
    },
    "adapters": {
        "mqtt": {
            "plugin": "mqtt_plugin",
            "domain": "192.168.4.1"
        },
        "http": {
            "plugin": "http_plugin",
            "domain": "192.168.4.2"
        }
    },
    "templates": {
        "siemens_s7_1200": {
            "starting_state": "Faulty",
            "states": [
                "Success",
                "Faulty"
            ],
            "rules": [ ... ]
        },
        "schneider_electric_1": {
            ...
        },
        "logic_node": {
            ...
        }
    }
}
```

Figure 4.13: System Descriptor File Snippet: full

4.4.2 State Manager

The State Manager (SM) component is a fundamental part of DeMon++. Its role is to digitally manage the system and the states of its devices based on the system description provided by the System Descriptor File. The functionalities of the State Manager can be summarized into two main phases:

- Parsing the System Descriptor File.
- Aggregating the system device states and determining the current state of the ICSs.

System Descriptor File Parser

As a parser, the State Manager component's first functionality is to read the System Descriptor File System Descriptor File and parse the configurations in order to model its behaviour accordingly. The parsing process involves two main steps.

- `Build the system model:` The first step consists of creating a structured digital model of the system by following the structure defined in the System Descriptor File System Descriptor File. This is achieved by using a tree data structure in which, each node in the tree represents either an actual system device or an abstraction of a group of actual devices called digital devices.
- `Apply the device-specific configurations:` The second step of the parsing process consists of merging the device-specific configurations with the common configurations defined in templates. In this process, the specific device configurations take priority over the common ones. This allows for the reuse of the same template for multiple types of devices. The same concept applies to rules, where device-specific rules overwrite the common ones. During this process, keywords defined in the System Descriptor File System Descriptor File are replaced with their corresponding references. Additionally, the adapter

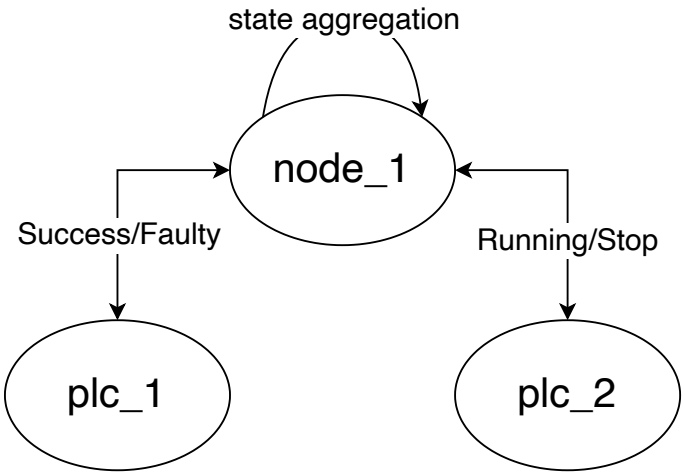

Figure 4.14: State Manager: Finite State Machine sample

settings specified in the System Descriptor File System Descriptor File are applied to each node in the tree. This instructs the leaf nodes to use a specific plug-in for establishing a connection with the monitoring agent or DeMon++ sub-instance at the designated domain.

In summary, the parsing process involves building a tree data structure and instantiating objects with user-defined properties assigned to each specific node in the tree.

System Device States Aggregator

Once the parsing process is ended, a digitally structured model of the system is finally defined. In order to make each node of the tree accomplish its own tasks (aggregation of input node states), SM defines the digital model as a [20] in which, each node state is defined by the aggregation of its input node states. Consequently, the ICS state is represented in the root node of the tree and it is defined by the aggregation of its input node states following the user-defined first-order logical rules. According to the structure sample provided by the snippet represented from Figure 4.13, the consequent finite state machine is outlined in Figure 4.14.

Current State / Inputs	*node_1: Good*	*node_1: Bad*
plc_1: Success *plc_2: Running*	**Good**	**Good**
plc_1: Success *plc_2: Stop*	**Good**	**Bad**
plc_1: Faulty *plc_2: Running*	**Bad**	**Bad**
plc_1: Faulty *plc_2: Stop*	**Bad**	**Bad**

Figure 4.15: State Manager: State Transition Table sample

Therefore, this approach consists of leveraging the finite state machine transition tables [21] to automate each node aggregation process. An example of a state transition table is represented in Figure 4.15. During the parsing phase, the finite state machine transition tables are built following the rules defined in the `rules` property defined in the System Descriptor File. In particular, the rules are applied top-down. As mentioned above, the JsonLogic [19] rules are defined in the `rule_logic` property, while the state of the related rule is defined by the property `out_state`. Practically, for each node, the building process consists of 2 steps.

The first step consists of initializing a matrix data structure, which will serve as our state transition table. The matrix rows represent all possible combinations of each input state, while the columns represent the possible states supported by the node. Since the number of supported states for the input device can vary, the combination order doesn't matter and we don't want repetition we can determine the table rows size by leveraging the "Cartesian product" [22]. The "Cartesian product"

(or "Cross product"), is a concept in set theory that allows us to combine multiple sets to create a new set containing all possible combinations of their elements. In our case, each set is represented by the group of supported states for each input device. Hence, the matrix row size can be summarized as:

$$\text{number of rows} = \prod_{i=1}^{n} \text{number of supported states for device } i \tag{4.1}$$

The number of columns corresponds to the number of states supported by the node. In each cell of the matrix, the transition state is placed. This state indicates the state the node must assume when the node is in a certain current state and there is a certain combination of input states. Let's consider an example:

Suppose a node has 3 input devices, with the first device supporting 2 different states, the second device supporting 3 states and the third one supporting 2 states. Finally, let's assume that the node supports 4 states. Then, the building process will create a matrix with shape:

$$\text{rows} = 2 \times 3 \times 2 = 12 \quad \text{and} \quad \text{cols} = 4 \tag{4.2}$$

The second step involves filling each cell of the built matrix with the correct state according to the defined rules. This is achieved using JsonLogic user-defined rules. For each row and column, all rules are executed top-down. When a rule is satisfied, the state to be filled in the corresponding matrix cell is determined by the `out_state` property of the respective rule. This process is repeated for each system node during the parsing phase. Hence, at run-time, the aggregation process and the subsequent propagation through the tree are achieved by performing a table lookup.

Other functionalities

This component also provides some additional functionalities, which are as follows:

- Disabling specific monitoring of actual or digital devices: This functionality allows for the selective deactivation of monitoring for particular physical or virtual devices. When applied to a digital device, it also disables the monitoring of all its sub-devices.

- Forcing a specific state for actual or digital devices: This functionality enables the enforcement of a predetermined state on a particular physical or virtual device. It serves a similar purpose to disabling monitoring but differs in that it provides the ability to simulate device monitoring. Essentially, forcing a device into a specific state serves as a workaround for disregarding the actual device status.

These features have been designed to enhance maintainability. They are primarily intended for use cases involving upgrades, repairs, or modifications to the ICS. By completely ignoring the state of one or more devices, the first functionality ensures a smooth transition during system changes. The second functionality, on the other hand, allows the system to function as if the device is being monitored, offering flexibility in managing device states. Overall, these functionalities contribute to the overall maintainability and adaptability of the ICS.

4.4.3 External Services

In order to work, DeMon++ requires some external support. Specifically, we can group external services into three different types:

- Monitoring Agents.

- DeMon++ sub-domain instances.

- Graphical User Interface.

Monitoring Agents

By "Monitoring Agents," we refer to all the existing tools used to extract diagnostic logs from devices. They act as middleware between devices and DeMon++. Possible solutions and research topics related to monitoring agents can be found in this thesis under the section Related Work. In a heterogeneous environment, finding a single monitoring agent compatible with all kinds of devices can be challenging. To ensure maximum compatibility, DeMon++ has been designed to support multiple monitoring agents. Thus, DeMon++ can be seen as an abstraction layer that leverages different monitoring agents, allowing for wide device monitoring support. Communication between monitoring agents and DeMon++ is facilitated through an adapter protocol. More details about the adapter protocol can be found in the State Inspector Adapter section. Unfortunately, achieving compatibility with all monitoring agents is not sufficient. Heterogeneous devices and their related monitoring agents generate different diagnostic logs. To interpret these logs uniformly, a standard language is required. In the monitoring context, the concept of "status" aligns with our objective. As discussed in literated work (Related Work), there is already a standard specifically designed for this purpose: the Namur NE 107 standard [7]. This standard promotes interoperability among monitoring agents. However, in order to achieve adaptability and compatibility, DeMon++ also allows users to define their own state-based standards. Using the System Descriptor File System Descriptor File, users can specify the states supported by each device. This flexibility enables users either to apply existing standards such as Namur NE 107, to define their own standards or mix them. Therefore, DeMon++ requires monitoring agents not only to extract diagnostic logs but also to convert them into states respecting the System Descriptor File.

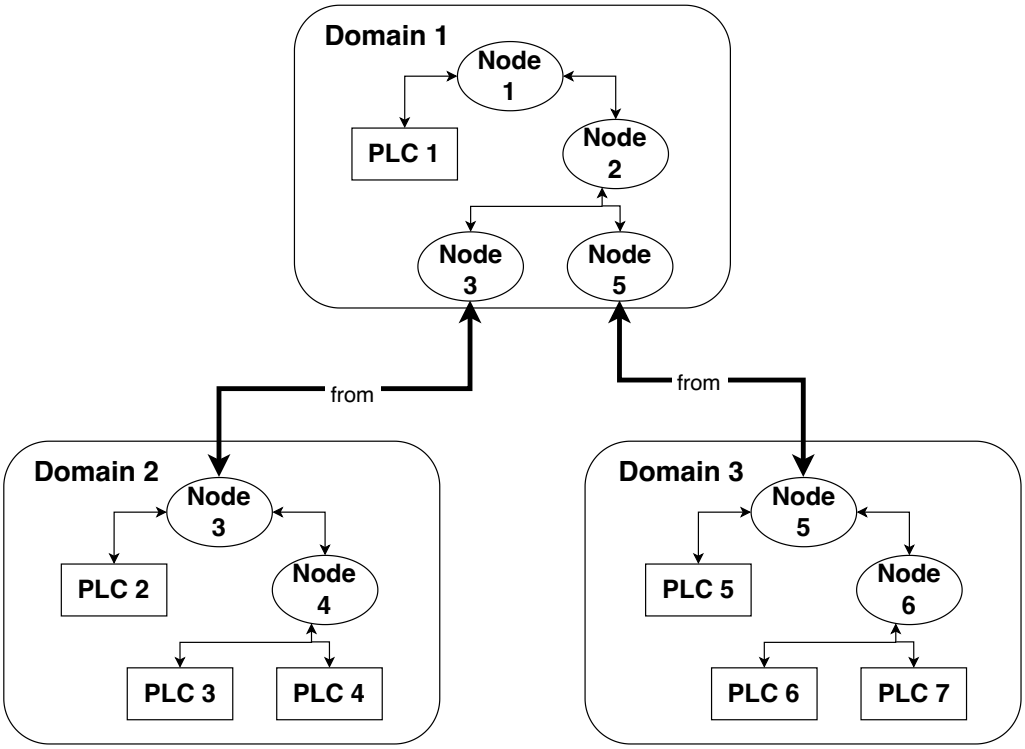

Figure 4.16: DeMon++ hierarchical finite state machine sample

DeMon++ sub-domains

As mentioned in the Distributed ICS use-case section, from the perspective of DeMon++, an ICS is referred to as a monitored domain. Therefore, as shown in Figure 4.16, we can handle distributed ICS domains by instantiating multiple DeMon++ instances. Each instance is responsible for monitoring a specific ICS domain and can identify its own domain using an IP defined by the `domain` property in the System Descriptor File.

Therefore, since each DeMon++ instance operates as a finite state machine, as explained in the State Manager component section, and each instance is hierarchically linked to other DeMon++ instances according to the distributed ICS, DeMon++ root domain and its sub-domains can be represented as a hierarchical finite state machine [23].

At the implementation level, this concept has been consistently applied: each

DeMon++ instance provides a set of APIs for sharing the "domain state" with upper-layer domains. The "domain state" refers to the result of state aggregation from all the devices monitored within the domain. From the perspective of upper-layer domains, a DeMon++ sub-domain is essentially a monitoring agent. In fact, the only difference between a DeMon++ sub-domain instance and Monitoring Agents is the plug-in used to establish the connection. More details about the DeMon++ plug-ins can be found in State Inspector Adapter.

Graphical User Interface

The last external service involved is the graphical user interface (GUI). As mentioned earlier, a distributed industrial system can be fragmented and distributed into multiple sub-systems. Therefore, the GUI must also follow this concept. Specifically, each DeMon++ instance acts as a back-end and provides GUI APIs. These APIs offer the following functionalities:

- Real-time visualization of the ICS and the current status of its devices.
- The possibility to force a node to remain in a specific state.
- Disabling the monitoring of specific devices.

More details regarding these functionalities can be found in the State Manager component section.

All of the above functionalities are achieved by leveraging REST APIs. An example sequence diagram for GUI states retrieving over HTTP is presented in Figure 4.17.

4.4.4 State Inspector Adapter

The heterogeneous external service components (see section 4.4.3) can vary significantly in the protocols they require. To achieve interoperability, an adapter protocol

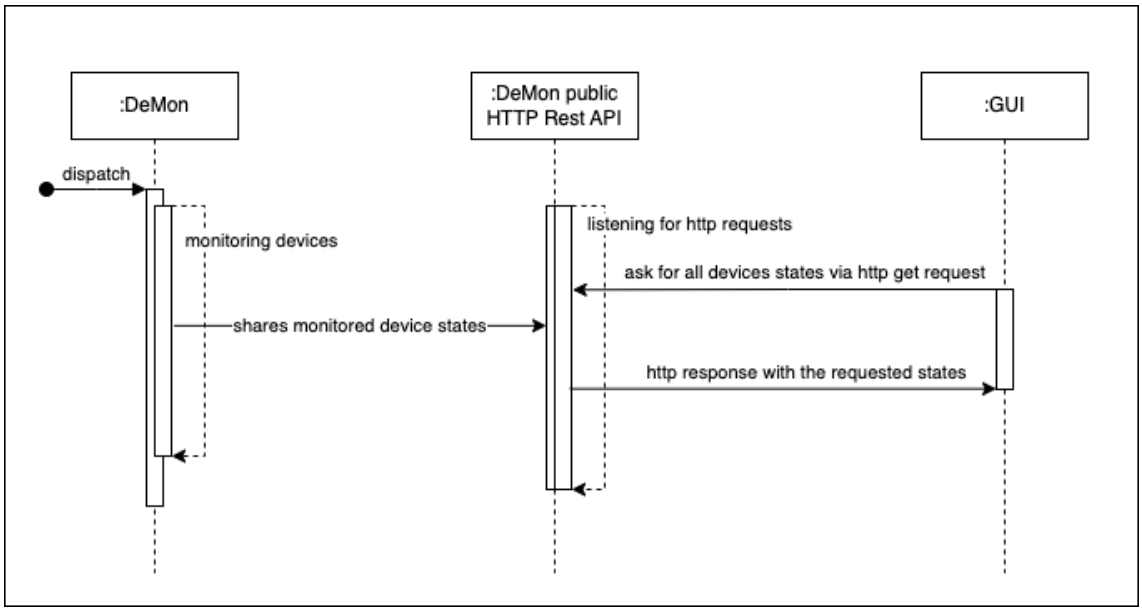

Figure 4.17: DeMon++ GUI API - Sequence Diagram

is employed. The State Inspector Adapter component is responsible for facilitating the connection with these external sources, ensuring maximum flexibility, adaptability, and extensibility. In order to work with different protocols, additional software is required to support various connection protocols. Therefore, DeMon++ has been designed to be extended with plug-ins. As shown in the Python snippet code in Figure 4.18, a DeMon++ plug-in is implemented as a class that follows a common interface called `Adapter`. Each plug-in applies the common `Adapter` interface by implementing a specific communication protocol. This design allows DeMon++ to be open and adaptable to various Monitoring Agents. DeMon++ determines which plug-in to use for a particular monitoring agent through the `plugin` property defined in the System Descriptor File.

As mentioned in DeMon++ sub-domains, the State Inspector Adapter component in DeMon++ is used not only for retrieving states from Monitoring Agents but also for retrieving states from DeMon++ sub-domains. Therefore, a plug-in is

```
class Adapter:

    def __init__(domain: str, device_ip: str = None, device_name: str = None) -> None:
        pass

    def get_current_state(self) -> str:
        """
        Get the current state of the actual device with the given ip.

        Returns:
            str: the current state of the device with the given ip.
        """
        pass
```

Figure 4.18: State Inspector Adapter Snippet: Python adapter interface

required to establish connections among DeMon++ instances. Additionally, each DeMon++ instance must support the outsourcing of its own state. One possible approach could be to leverage the REST API built for the GUI (see Graphical User Interface) external service. An example of this approach is reported and explained by the sequence diagram in Figure 4.19. This diagram provides an example of the workflow between two DeMon++ domains, where a custom DeMon++ plug-in and GUI REST API are involved. As an alternative, we can build proper APIs for this purpose. In particular, in this case, their function would just be limited to share with a specific protocol a real-time state of this instance root node.

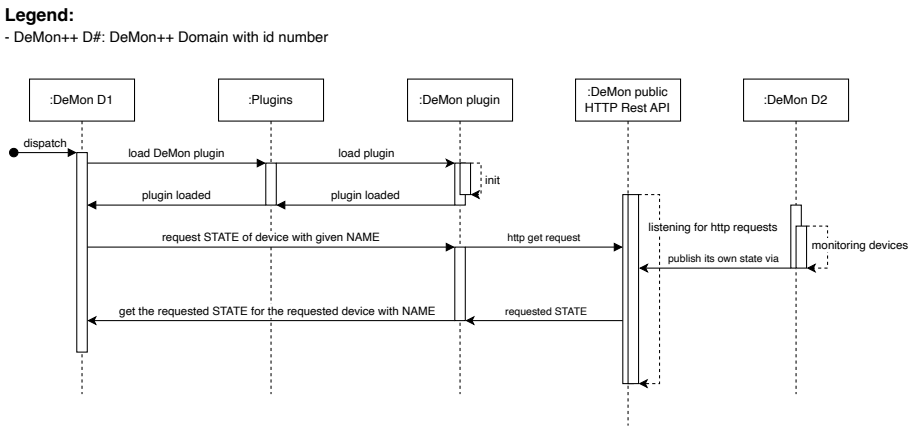

Figure 4.19: State Inspector Adapter: DeMon++ intra-domain communication via HTTP REST APIs

5 Validation

In this chapter, we start with a presentation of our proof of concept implementation. We are also going to discuss and tentatively identify possible directions for further development and implementation of the research results.

5.1 Proof Of Concept Overview

This work is the result of research and collaboration between Siemens AG and CERN. Throughout the development process, it was essential to outsource the work through prototypes and obtain evaluation feedback. As a result, a proof-of-concept implementation was developed. Several people from both Siemens and CERN were involved in the thesis development process. To ensure that the prototype could be shared and executed by the team, we decided to develop it in a platform-independent manner. In particular, Docker was chosen as a common platform for testing the prototype and eliminating dependency issues. Python 3.11 [24] was chosen as the programming language for its ease of use, power, and wealth of useful libraries such as NumPy [25], pandas [26] and panzi-json-logic [27].

The implemented prototype serves both as a means of improving communication and understanding between team members and as a demonstration that the concepts described in this thesis are feasible.

During the development of this project, we recognised the paramount importance of validating our work through the developed prototype. As a result, we made the

strategic decision to create a virtual simulated environment that would not only significantly streamline software testing, but also facilitate seamless collaboration between remote team members. To address this critical requirement, we identified a suitable Siemens tool that was capable of virtualising real devices through the use of software threads.

This approach allowed us to obtain invaluable feedback from CERN and to validate our work comprehensively. It's also worth noting that this project was a close collaboration with CERN Openlab, characterised by a strong spirit of partnership and knowledge sharing. Through this fruitful collaboration, the team members on the CERN side not only provided critical insights, but also had the unique opportunity to conduct extensive testing of our solution in a real-world environment that mirrored the demanding conditions of their scientific experiments.

This hands-on testing confirmed that DeMon++ met CERN's requirements in several key areas. The CERN team praised its exceptional flexibility, allowing seamless adaptation to evolving experimental setups. The maintainability and scalability of the solution was evident, as it easily adapted to their complex research infrastructure. Compatibility was assured as it seamlessly integrated with their existing systems, while extensibility provided a solid foundation for future development and expansion of their experimental setups.

5.2 Future Development

With this thesis, our objective was to find a lightweight approach that enables the creation and modelling of a digital twin for an ICS, comprising thousands of devices. The primary goal was to achieve intelligent system observability. Additionally, we aimed to establish a fully configurable automation process, allowing users to define logical sets of conditional rules. These rules would facilitate the automatic aggregation of each device's state and enable the derivation of the overall system health.

At the same time as the thesis was being written, front-end software was being developed at CERN to interact seamlessly with this tool.

5.2.1 Application as Industrial Edge App

Because of its nature to suit distributed environments, we think that DeMon++ has the potential to be suitable as an edge application within ICSs. An edge application is an application designed to run on computing devices deployed at the edge of the network. More details on edge computing are described in the Edge Computing section. This subsection aims to present the benefits of implementing our monitoring tool as an edge application.

Advantages as an Edge Application

By deploying our monitoring tool at the edge, we can take advantage of several benefits that improve the efficiency, responsiveness and overall performance of the ICS. Some of the key benefits are:

- **Reduced Latency:** By placing DeMon++ closer to the monitored devices, the latency associated with data transmission and processing is significantly reduced. This enables real-time monitoring, resulting in more accurate device observability.

- **Bandwidth Optimization:** By performing monitoring and analysis at the edge, only relevant and essential data is sent to the upper nodes, reducing the overall amount of data transmitted over the network.

- **Improved Resilience:** Deployed as an edge application, DeMon++ would provide increased resilience by allowing the monitoring tool to continue operating in the event of network failure or intermittent connectivity issues.

- **Enhanced Privacy and Security:** By processing and analysing data locally at the edge, DeMon++ can address privacy concerns by minimising the transfer of sensitive information across the network. In addition, edge-based security measures can be implemented to detect and mitigate potential threats before they reach the central monitoring system.

- **Decentralized Decision-making:** Deploying DeMon++ at the edge enables decentralised decision-making, allowing faster response times and reducing computational costs across the distributed system.

By leveraging these benefits, implementing our monitoring tool as an edge application within ICSs can significantly improve system performance, responsiveness, privacy and security. However, it is important to recognise that an edge-centric approach requires the deployment of advanced infrastructure, resulting in high costs associated with managing and supporting edge devices. This is particularly relevant when considering legacy ICSs, as the need to upgrade hardware can result in significant costs that outweigh the potential benefits.

6 Conclusion

In this thesis, we conducted an analysis of the challenges associated with monitoring complex ICSs that consist of a large number of heterogeneous devices. Our objective was to design a novel approach for constructing a flexible, scalable, and maintainable monitoring framework. This framework's aim was to abstract, aggregate, and summarize the health state of such ICSs, adopting the CERN use-case as a relevant case study.

Our approach has been to use an interpreter software architecture to provide a flexible and adaptable method of defining a digital abstraction of a distributed ICS. This approach allows us to effectively manage devices with different characteristics while ensuring compatibility through the use of an adapter protocol. In addition, by representing device health information as states rather than diagnostic logs, we gain valuable insight into the overall state of the system. The system model is structured using a tree format, with each tree acting as a finite state machine. This design allows each tree node to autonomously manage its own state by aggregating the states of its subnodes using user-defined first-order logic rules. To meet the requirements of distributed systems, we have made the strategic choice to implement DeMon++ as a seamlessly distributed hierarchical finite state machine. This organisation promotes efficiency and scalability in handling complex system behaviour.

6.1 Shortcomings and Advantages

Despite the many benefits of the monitoring tool, it is important to recognise and address certain limitations:

- **Configuration Requirement:** The tool requires extensive configuration before it can be used effectively, making it difficult to use straight out of the box.
- **Dependency on Compatible Monitoring Agents:** The tool relies on compatible monitoring agents to collect data, which can impose constraints on the system and require additional setup effort.
- **Insufficient State Descriptions:** The current state representation may not provide sufficient detail to accurately describe the health of certain devices or subsystems within the ICS.
- **Focus on Current ICS Status:** The tool primarily monitors and reports on the current status of ICS components and lacks the ability to predict or identify potential device failures before they occur.

In contrast to the aforementioned limitations, the monitoring tool offers several advantages:

- **High-Level Compatibility:** The tool demonstrates high compatibility as it is designed to potentially support a wide range of devices commonly found in ICS environments.
- **Flexibility and Customisation:** The tool provides a flexible and customisable framework, allowing users to tailor its functionality to specific monitoring requirements or system configurations.

- **Ease of Interpretation:** The monitoring tool simplifies the interpretation of the current status of the ICS by presenting information in a clear and understandable manner.

- **Comprehensive System Summary:** The tool provides a summarised overview of the overall status of the ICS, including its sub-systems, allowing users to quickly assess the overall health and performance of the system.

By understanding both the benefits and limitations of the proposed monitoring tool, we can effectively assess its applicability and make informed decisions regarding its implementation and use in ICS.

References

[1] Inductive Automation. "What is scada?" (2023), [Online]. Available: https://www.inductiveautomation.com/resources/article/what-is-scada (visited on 04/01/2023).

[2] CERN. "Cern openlab". (2023), [Online]. Available: https://home.cern/science/computing/cern-openlab (visited on 05/26/2023).

[3] C. G. B. Franek, "Smi++ object oriented framework for designing and implementing distributed control systems", vol. 45, 1998. (visited on 09/20/2023).

[4] D. Hästbacka, L. Barna, M. Karaila, Y. Liang, P. Tuominen, and S. Kuikka, "Device status information service architecture for condition monitoring using opc ua", in *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–7. DOI: 10.1109/ETFA.2014.7005141.

[5] O. FOUNDATION. "Unified architecture". (2023), [Online]. Available: https://opcfoundation.org/about/opc-technologies/opc-ua/ (visited on 07/06/2023).

[6] V. Kapoor and D. Haller, "Cloud-based standardized device diagnostics for optimized operability of plants in the process industry: Cloud based self-monitoring and diagnosis of the field devices", in *Companion Proceedings of the 10th International Conference on the Internet of Things*, ser. IoT '20 Companion, Malmö, Sweden: Association for Computing Machinery, 2020, ISBN:

9781450388207. DOI: 10.1145/3423423.3423466. [Online]. Available: https://doi.org/10.1145/3423423.3423466.

[7] I. Tools. "Namur ne107 standard". (2023), [Online]. Available: https://instrumentationtools.com/namur-ne107-standard/ (visited on 06/01/2023).

[8] M. Wollschlaeger, S. Theurich, A. Winter, F. Lubnau, and C. Paulitsch, "A reference architecture for condition monitoring", in *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, 2015, pp. 1–8. DOI: 10.1109/WFCS.2015.7160555.

[9] C. Akpolat, D. Sahinel, F. Sivrikaya, G. Lehmann, and S. Albayrak, "Chariot: An iot middleware for the integration of heterogeneous entities in a smart urban factory.", in *FedCSIS (Position Papers)*, 2017, pp. 135–142.

[10] L. Tseng, L. Wong, S. Otoum, M. Aloqaily, and J. B. Othman, "Blockchain for managing heterogeneous internet of things: A perspective architecture", *IEEE network*, vol. 34, no. 1, pp. 16–23, 2020.

[11] G. Valente, T. Fanni, C. Sau, T. D. Mascio, L. Pomante, and F. Palumbo, "A composable monitoring system for heterogeneous embedded platforms", *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 20, no. 5, pp. 1–34, 2021.

[12] NIST. "Industrial control system (ics)". (2023), [Online]. Available: https://csrc.nist.gov/glossary/term/industrial_control_system (visited on 04/01/2023).

[13] B. Z. Shan Gao Junjie Chen and K. Ren, "Privacy-preserving industrial control system anomaly detection platform", vol. 2023, p. 12, 2023. DOI: 10.1155/7010155. [Online]. Available: https://doi.org/10.1155/2023/7010155 (visited on 07/11/2023).

[14] CERN. "Be-ics industrial control systems responsibles". (2023), [Online]. Available: https://readthedocs.web.cern.ch/display/ICKB/BE-ICS+Industrial+Control+Systems+responsibles (visited on 04/01/2023).

[15] A. AWS. "What is mqtt?" (2023), [Online]. Available: https://aws.amazon.com/what-is/mqtt/ (visited on 03/28/2023).

[16] E. J. Bruno. "Introducing the constrained application protocol (coap) for java". (2023), [Online]. Available: https://blogs.oracle.com/javamagazine/post/java-coap-constrained-application-protocol-introduction (visited on 07/06/2023).

[17] S. AG. "Siemens industrial edge ecosystem documentation". (2023), [Online]. Available: https://cache.industry.siemens.com/dl/files/785/109783785/att_1038592/v1/ied_operation_enUS_en-US.pdf (visited on 04/28/2023).

[18] J. organization. "Json main webpage". (2023), [Online]. Available: https://www.json.org/json-en.html (visited on 06/15/2023).

[19] J. Wadhams. "Jsonlogic main webpage". (2023), [Online]. Available: https://jsonlogic.com/ (visited on 05/02/2023).

[20] Wikipedia. "Finite-state machine". (2023), [Online]. Available: https://en.wikipedia.org/wiki/Finite-state_machine (visited on 04/01/2023).

[21] D. H. Sarah L. Harris, "Digital design and computer architecture", pp. 106–169, 2022.

[22] T. Johnston. "Cartesian product". (2023), [Online]. Available: https://www.sciencedirect.com/topics/computer-science/cartesian-product (visited on 06/16/2023).

[23] M. Yannakakis, "Hierarchical state machines", 2000.

[24] Python. "Python". (2023), [Online]. Available: https://www.python.org/ (visited on 09/20/2023).

[25] NumPy. “Numpy”. (2023), [Online]. Available: https://numpy.org/ (visited on 09/20/2023).

[26] pandas. “Pandas”. (2023), [Online]. Available: https://pandas.pydata.org/ (visited on 09/20/2023).

[27] panzi-json-logic. “Panzi-json-logic”. (2023), [Online]. Available: https://pypi.org/project/panzi-json-logic/ (visited on 09/20/2023).