Escuela Politécnica Superior de Linares

M.Sc. in Telecommunication Engineering

Universidad de Jaén

Escuela Politécnica Superior de Linares

UNIFIED NAMESPACES IN INDUSTRIAL IOT ARCHITECTURES

Student: **Rubén López Muñoz**

Supervisor: Prof. Dr. José Ángel Fernández Prieto
Supervisor: Prof. Dr. Ulrich Birkel

Department: Ingeniería de Telecomunicación

September, 2024

Universidad de Jaén

Escuela Politécnica Superior
de Linares

UNIFIED NAMESPACES IN INDUSTRIAL IOT ARCHITECTURES

Student: Rubén López Muñoz

Supervisor: Prof. Dr. José Ángel Fernández Prieto
Supervisor: Prof. Dr. Ulrich Birkel

Department: Ingeniería de Telecomunicación

Student Signature:

LOPEZ MUÑOZ RUBEN - 26506985Z

Digitally signed by LOPEZ MUÑOZ RUBEN - 26506985Z
Date: 2024.09.08 20:30:34 +02'00'

Supervisor Signature:

FERNANDEZ PRIETO JOSE ANGEL - 26222606F

Firmado digitalmente por FERNANDEZ PRIETO JOSE ANGEL - 26222606F
Fecha: 2024.09.08 12:03:52 +02'00'

Supervisor Signature:

Ulrich Birkel

Digital unterschrieben von Ulrich Birkel
Datum: 2024.09.08 14:32:40 +02'00'

September, 2024

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

This thesis explores the concept of Unified Namespace (UNS) as a key architectural framework for Industrial Internet of Things (IIoT) data management in Industry 4.0 and Smart Manufacturing projects. Traditional point-to-point communication between devices and applications becomes a bottleneck with the growing number of IIoT devices. UNS offers a central and consistent data format, simplifying integration and data exchange between Operational Technology (OT) and Information Technology (IT) systems.

The motivations for adopting UNS include improved data visibility, accessibility, and decision-making capabilities. The research investigates the Sparkplug specification for MQTT as a promising UNS data model due to its growing popularity and standardized structure. Additionally, the thesis proposes an end-to-end architecture that utilizes MQTT and Apache Kafka to address real-time data processing, scalability, and data format transformation within an IIoT environment. This architecture serves as a reference for future projects and demonstrates the potential of UNS for effective IIoT data management.

Keywords: Industrial Internet of Things (IIoT), Unified Namespaces (UNS), event-driven architecture, OPC UA, OPC Router, Python, MQTT, Kafka, Sparkplug, ThingsBoard.

ABBREVIATIONS

CSV: Comma-Separated Value.

DDoS: Distributed Denial of Service.

ERP: Enterprise Resource Planning.

HMI: Human-Machine Interface.

IDPS: Intrusion Detection and Prevention Systems.

IIoT: Industrial Internet of Things.

IT: Information Technology.

IoT: Internet of Things.

IP: Internet Protocol.

ISA: International Society of Automation.

JSON: JavaScript Object Notation.

KPIs: Key Performance Indicators.

MES: Manufacturing Execution System.

M2M: Machine to Machine.

MQTT: Message Queue-Telemetry Transport.

MOM: Message-Oriented Middleware.

OPC UA: Open Platform Communications Unified Architecture.

OT: Operational Technology.

PLC: Programmable Logic Controller.

SCADA: Supervisory Control and Data Acquisition.

SBOM: Software Bill of Materials.

SQL: Structured Query Language.

SSL: Secure Socket Layer.

TCP: Transport Control Protocol.

TLS: Transport Layer Security.

UDP: User Datagram Protocol.

UNS: Unified Namespace.

URI: Uniform Resource Identifier.

WLAN: Wireless Local Area Network.

1 INTRODUCTION

Industrial communication is one of the key points for modern automation and plays an important role in the implementation of the Industry 4.0 and Industrial Internet of Things (IIoT). It is employed to control and monitor machines and systems in production and manufacturing processes, to assess the quality of services, to perform maintenance, and to integrate management systems [1].

Companies are constantly generating data information from applications, systems, or user interactions. To use the full potential of this data, it must be efficiently moved from its source to the analysis platform. Improving this data flow enhances the company agility and responsiveness, allowing companies to focus on core business objectives with real-time information while minimizing data management [2].

Traditional manufacturing systems were built based on the ISA 95 standard, which defines a five-level hierarchical framework for integrating diverse systems and devices within an industrial architecture. It defines five levels, from the enterprise to the industrial devices. This standard is usually visualized as a pyramid model, as shown in following Figure 1.

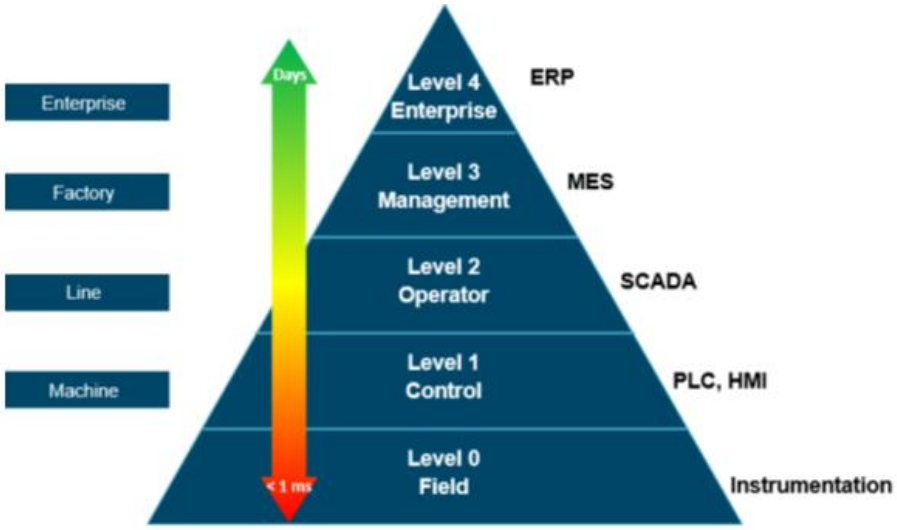


Figure 1. ISA-95's Automation Pyramid.

Adapted from: https://www.linkedin.com/pulse/from-industry-30-40-practice-oriented-executive-summary-hammari

Businesses implementing this hierarchy usually are divided into two main areas. The Operational Technology side (OT), with devices and industrial machines that are connected to control applications to manage them, like Programable Logic Controllers (PLCs). On top there must be an application to gather all the data from the devices, usually the Supervisory Control and Data Acquisition (SCADA). The remaining is called the Information

Technology side (IT), with higher-level applications that analyse the data, like Manufacturing Execution System (MES), or Enterprise Resource Planning (ERP).

These traditional systems were typically implemented with a direct point-to-point connection using client-server pattern between the devices and each application, as represented in the following Figure 2.

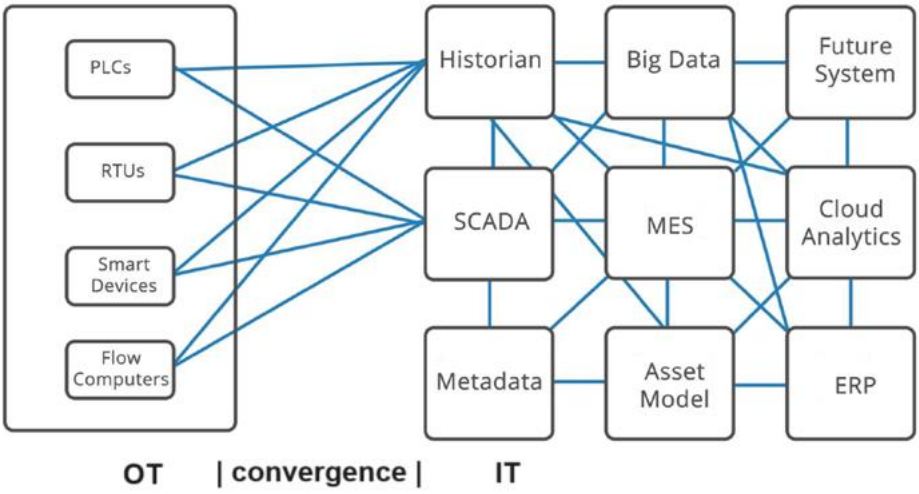


Figure 2. Traditional IT/OT convergence diagram.

Adapted from: https://sparkplug.eclipse.org/

Additionally, it was common to uses proprietary application interfaces only known and provided by device vendors. This means that applications become dependent on a particular device, making it impossible to remove and replace it with a different vendor without rebuilding each application interface. Another significants challenges are the high complexity of network, tight coupling between components, lack of interoperability, difficulties in managing data from various sources, and most notably, inability to scale effectively

While these approaches were effective for smaller-scale systems, they have limitations when dealing with the incremental use of Industrial IoT devices and new high-level applications as Big Data or AI analytics. To understand the new scale, the number of IoT devices across all industry verticals is forecast to grow to more than eight billion by 2033 [3].

In other to solve this problem, a more suitable communication than client-server should be used, as a hub model where devices and applications communicate indirectly through a central point of information. This hub, known as Message-Oriented Middleware (MOM), is one of the core principles for implementing a Unified Namespace (UNS).

By eliminating direct communication between devices and applications, UNS simplifies integration and makes it easier to merge OT and IT systems without requiring knowledge of each other's implementation details. This decoupling also enhances scalability and flexibility, as can be drawn from the following Figure 1Figure 3.

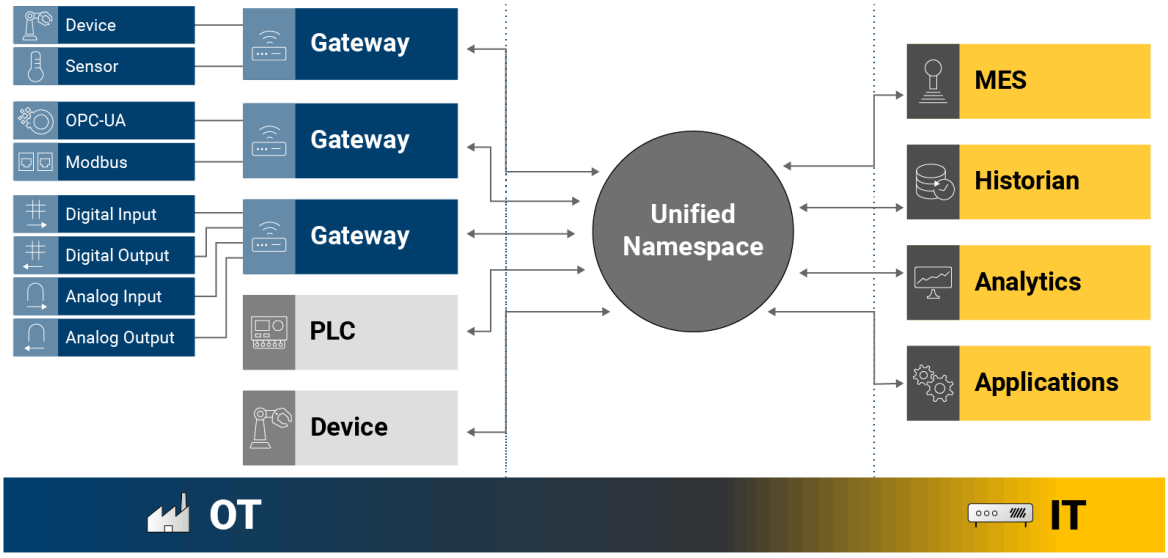


Figure 3. Unified Namespace IT/OT convergence.

Source: https://www.hivemq.com/resources/smart-manufacturing-using-isa95-mqtt-sparkplug-and-uns/

Furthermore, this architecture allows for easy device replacement. Even if a device is replaced with one from a different vendor, other network participants will not be affected as long as the new device publishes to the same topic namespace.

Essentially, a Unified Namespace is a central and consistent data format for all the data across different systems. Ideally with a data model specified to organize data in a hierarchical structure of the business. Usually implemented using an event-driven architecture to improve the interoperability and seamless exchange of information. Unified Namespaces is often shown as the core architectural framework for future digital transformation initiatives within Industry 4.0 and smart manufacturing projects.

1.1 Industrial Motivation

The concept of UNS has become a trend in recent years, as organizations have recognized the need for a more cohesive and scalable approach to IIoT data management. Industry leaders and digital transformation providers have been actively developing and promoting UNS solutions.

One of the primary motivations for adopting UNS is to gain better visibility and accessibility into their operational data. By establishing a centralized data hub, organizations can consolidate information from diverse sources, ensuring that it is easily accessible and

understandable. This improved data visibility enables fast decision-making, identifies areas for optimization, and supports active maintenance and troubleshooting.

This thesis has been done with the collaboration of it-novum, a german company located in Fulda, focused on providing digital data solutions for private customers.

it-novum's participation is driven by the motivation to gain a deeper understanding of Unified Namespaces and their fundamental principles. This knowledge will be used to develope best practices for UNS implementation and management, as well as exploring emerging solutions for current and future clients.

A particular area of interest is the Sparkplug specification for MQTT. As its adoption within the industry grows, it-novum has observed increasing interest from clients in leveraging this technology.

Furthermore, the implementation of the proposed architecture in the following chapters is of significant interest. The end-to-end solution, from OT devices to IT platforms, represents an approach that has been previously explored by the company, but designed for specific customers. The architecture can serve as a valuable demonstration and reference for future projects and customers.

1.2 Academical Motivation

This research can serve as an entry point for those new to the industrial world into understanding the unique challenges and opportunities presented by IIoT environments. Also, by providing a comprehensive overview of UNS, this thesis can help researchers to understand the potential benefits and challenges associated with implementing IoT projects.

Despite the extensive academical research on underlying technologies like OPC UA, MQTT, and Apache Kafka, the research about UNS is limited. This can be attributed to its industry-centric nature, which have primarily been develop in practical industrial implementations rather than a topic in academic research.

1.3 Architecture Implementation

This thesis implements an end-to-end architecture designed to address the specific challenges faced in Industrial IoT environments. By combining the strengths of MQTT and Kafka, the architecture effectively handles real-time data processing, ensures system scalability, and transform multiple data formats into one unified data model namespace.

MQTT was selected for its ease of use and widespread adoption among OT systems. Many PLCs and sensors natively support MQTT, simplifying integration. Apache Kafka has been chosen for its scalability, fault tolerance, and existing enterprise support, making it well-suited for the data-intensive requirements of the architecture. This hybrid approach that combines MQTT and Kafka addresses both the easy of use and IIoT features of the solution, providing a robust and versatile approach.

As a proposal by it-novum, the Sparkplug specification for MQTT have been studied and implemented to be the UNS data model. Sparkplug provides a standardized structure for topic and payload definitions, improving interoperability. Sparkplug is gaining popularity among IIoT devices, making it a desirable feature for customers.

The proposed architecture, as illustrated in the following Figure 4, provides a solid architecture for IIoT implementations. To show the full capabilities of the solution, the dotted lines components indicate potential extensions to the architecture that could be easily integrated.

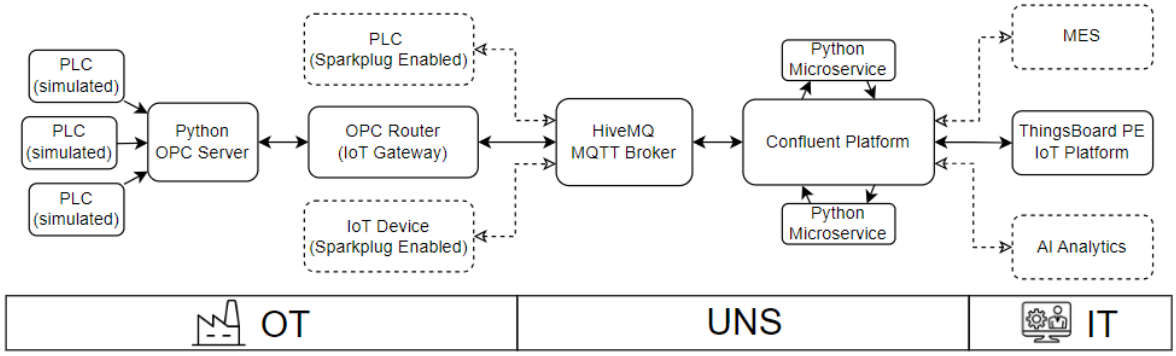


Figure 4. High-level implemented architecture diagram

On the OT side, a Python OPC Server has been deployed to simulate underlying devices and generate a continuous stream of data. An OPC UA client, the OPC Router, has been configured to monitor these simulated devices and act as an IoT Gateway, transforming OPC UA messages into MQTT messages that follows the Sparkplug specification.

On the IT side, ThingsBoard IoT Platform has been deployed to enable real-time data visualization through dashboards. The platform also served as an interface for sending commands to OT devices, allowing for remote control of internal functions like reboot.

1.4 Document Structure

The master thesis document is structured into eight chapters, as follows:

Chapter 1. Introduction: Presents a comprehensive overview of the master thesis, including its motivation and a summarized view of the implemented architecture detailed in Chapter 4.

Chapter 2. Objectives: Clearly outlines the specific goals and objectives to be achieved during the master thesis.

Chapter 3. Fundamentals of Industrial IoT and Unified Namespaces: Offers an introduction to key concepts of Industrial IoT and the technologies employed in the thesis methodology. Additionally, it includes a literature review of the current state of Unified Namespaces.

Chapter 4. Implementation of UNS for Industrial IoT Architecture: Describes the methodology used to develop an end-to-end architecture, from its design to implementation, applying the concepts explored in Chapter 3.

Chapter 5. Evaluation of the Implementation: Reviews the overall end-to-end architecture implemented in Chapter 4, summarizes the challenges encountered, and validates its benefits and pitfalls.

Chapter 6. Conclusions: Discusses the list of the objectives presented in Chapter 2 and presents the conclusions drawn from the master thesis realization.

Chapter 7. Future Lines: Proposes potential avenues for expanding the master thesis in future projects or related documents.

Chapter 8. Appendices: Includes supplementary resources, such as the complete code for certain scripts and extensive configuration files used in the thesis.

2 OBJECTIVES

The main objective of this master's thesis is the evaluation and review of Unified Namespaces in Industrial IoT architectures. This objective will be supported through the implementation of a demostrational system applying UNS principles.

To achieve this goal, a series of specific secondary objectives have been proposed by it-novum. These are outlined as follows:

- Motivation and justification for the use of Unified Namespaces.
- Comparative analysis of Unified Namespaces implementations and alternative approaches.
- Industry and research landscape of Unified Namespaces.
- Design and implementation of a Unified Namespaces End-to-End Architecture.
- Evaluation of benefits and pitfalls for Unified Namespaces architectures.

3 FUNDAMENTALS OF INDUSTRIAL IOT & UNIFIED NAMESPACES

This chapter serves as the foundation for the subsequent chapters within this thesis. It presumes the reader does not possess a basic knowledge of IIoT. To establish a common understanding, this chapter will introduce several key concepts and technologies and literature review of Unified Namespaces.

3.1 Industrial IoT

3.1.1 *ISA 95 Standard*

ISA95 is a large specification with up to eight parts, different extensive documents, developed over many years. Therefore, the scope of this introduction is to only focus on providing the general view of the world-wide structure of businesses.

In the 1990s, as businesses increasingly adopted computers and networks, the need for standardized communication between administrative and production processes became evident. To address this, the International Society of Automation (ISA) developed ISA-95, a model that defines the structure and interactions of various software systems within an organization. Nowadays, ISA-95 is the world-wide standard for the integration of enterprise and control systems.

The main goal of ISA-95 standard is to establish a common language for business and production processes, to define a hierarchical structure for organizational software systems, and to specify how data should be exchanged between administrative and production functions.

The ISA-95 Standard provides a hierarchical model that outlines the different levels of manufacturing operations. It provides a framework for understanding the relationships between various systems and processes within a manufacturing facility. This hierarchy concept is also known as the Automation Pyramid [4], illustrated in the following Figure 5.

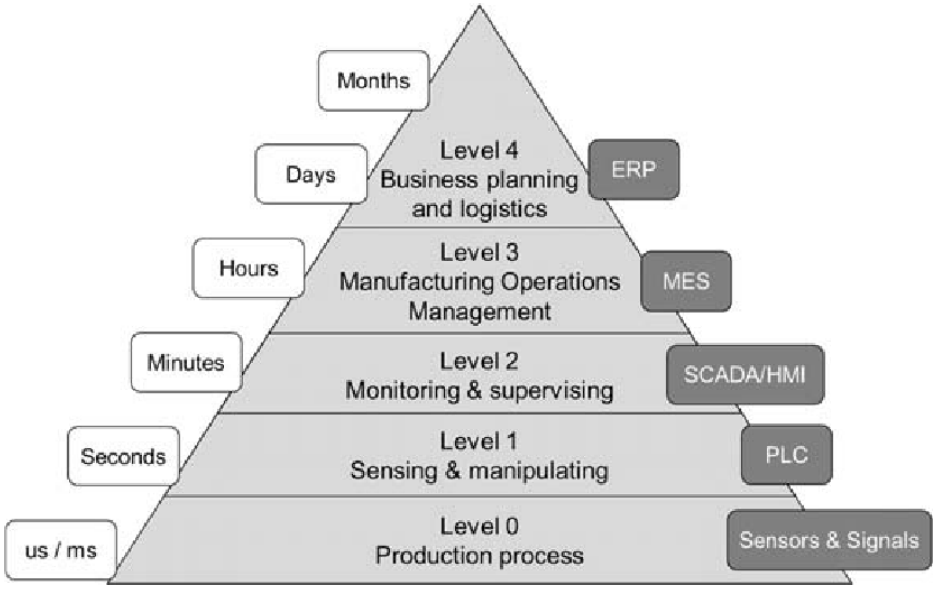


Figure 5. Automation Pyramid

Source: https://www.researchgate.net/publication/326224890_Implementing_Shop_Floor_IT_for_Industry_40

- **Level 0: Production Process (Sensors & Signals)**

It represents the physical production processes and the sensors and actuators that collect and control data. It includes devices like temperature sensors, pressure gauges, and valves.

- **Level 1: Field Devices and Functions (PLC)**

This level consists of programmable logic controllers (PLCs) and other field devices that directly interact with the production process. PLCs are responsible for controlling and monitoring the operation of machines and equipment based on real-time data from sensors.

- **Level 2: Automation (SCADA/HMI)**

SCADA (Supervisory Control and Data Acquisition) systems and HMI (Human-Machine Interface) software are located at this level. SCADA systems collect data from PLCs and other field devices, process it, and provide operators with a centralized view of the entire production process. HMI interfaces allow operators to interact with the SCADA system and control equipment.

- **Level 3: Manufacturing Operations Management (MES)**

MES systems integrate data from various sources, including SCADA systems, ERP systems, and quality control systems. They provide a comprehensive view of manufacturing operations, enabling managers to track production performance, manage inventory, and optimize processes.

- **Level 4: Business Planning and Logistics (ERP)**

ERP (Enterprise Resource Planning) systems handle high-level business functions such as finance, sales, and supply chain management. ERP systems integrate with MES systems to provide a holistic view of the entire manufacturing enterprise.

At the time of writing, ISA-95 consists of eight components and an additional data template that comprehends several hundred of pages. It is possible to brief summaries of these components on the official ISA Standards [5]. For the scope of this thesis, there is not needed to understand on detail each part:

- **Part 1: Models and Terminology:** It defines domain levels (control and enterprise systems), functions, information flows, categories, and definitions. It also describes the plant's production model (Enterprise->Site->Area->Line->Cell).
- **Part 2: Object Model Attributes:** It provides standardized definitions and terminology for clear communication across all elements of the business. It describes each model's potential attributes and how to link them together.
- **Part 3: Activity Models of Manufacturing Operations Management:** Standarizes business processes like production, quality control, and maintenance.
- **Part 4: Objects and Attributes for Manufacturing Operations Management Integration:** Defines data exchange processes between systems.
- **Part 5: Business-to-Manufacturing Transactions:** Standardizes the methods for integrating systems, using data models to define how information is exchanged.
- **Part 6: Messaging Service Model: F**ocuses on data transformation between systems.
- **Part 7: Alias Service Model:** Focuses on data transformation between local names to global names.
- **Part 8: Information Exchange Profiles** Defines a framework aimed at software vendors developing ISA-95 software.

Some of the most relevant challenges of ISA-95 standard [6] are the following:

- Older software systems may not be compatible with newer systems, requiring manual processes or additional middleware.
- Information may be isolated within departments, limiting its accessibility to others.
- Key performance indicators (KPIs) can be calculated multiple times, leading to inefficiencies.
- Management requires information more quickly than it is currently available.

Recent trends like Industry 4.0 and the Internet of Things (IoT) have introduced new complexities into the automation landscape. These technologies require a more interconnected and data-driven approach, which can challenge the traditional hierarchical model defined by ISA-95.

3.1.2 *OPC UA*

OPC UA, or OPC Unified Architecture, is the latest standard of OPC (Open Platform Communications), developed by the OPC Foundation. It sustitutes the older OPC Classic standard, which is still used in many existing automation installations.

While OPC Classic successfully established manufacturer-independent data exchange in automation, it had the major limitation of being platform dependent. Its reliance on Microsoft technologies (COM and DCOM) restricted OPC Classic servers and clients to Windows OS. This became a priority, as the industry was moving forward to use other platforms, like Linux, web architectures, cloud computing, and IoT devices [7].

3.1.2.1 Address Space

The Address Space serves as the foundation for data exchange in the OPC UA architecture. It provides a standardized way for servers to expose their data and functionalities to clients. It is defined as the *"collection of information that a Server makes visible to its Clients"* [8].

The address space is modeled as a collection of Nodes. Each node represents a specific piece of information, such as a data value that can be readed or written (Variable Node), a functionality in the server to trigger actions (Method Node), or the group of related nodes that represents an entity like a machine (Object Node)

To provide a unique way of identifying nodes across different vendors and applications, the Address Space is organized in Namespaces. The namespace is a URI that identifies the naming authority. Naming authorities include the underlying system, the local Server and standards bodies. Namespaces URIs are identified as integer indexes to permit a more efficient processing.

The index "0" is always reserved for the underlying base nodeset, defining core elements and methods defined in the OPC UA standard. The index "1" is also reserved for the local server to expose vendor-specific data or functionality that are not included in the standard.

Multiple standardized nodesets exist within the OPC UA ecosystem that has been created to improve the interoperability in specific automation sectors from standardization bodies [9].

3.1.2.2 Server/Client Architecture

The OPC UA protocol relies on the TCP/IP protocol stack for network communication, following a client-server architecture, establishing a two-way communication channel between devices. This architecture can be shown in the following Figure 6.

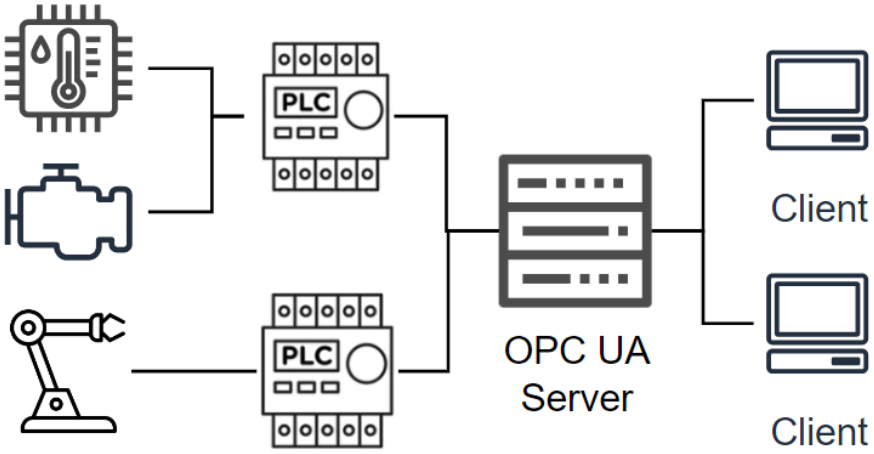


Figure 6. OPC UA Client/Server Architecture.

The OPC UA Server acts as the central data provider within the system. There can be multiple servers in the same infraestructure. First, it serves as a data source, retrieving data from multiple industrial automation equipment like programmable logic controllers (PLCs), sensors, and other devices. This data can encompass real-time process measurements, control settings, or device status information. When a client sends a request using either the request/response or publish/subscribe method, the server retrieves the relevant data from its internal storage or interacts with connected devices to fulfill the request. Finally, the server enforces security measures to ensure only authorized clients can access and modify data. This includes features like user authentication, authorization levels, and data encryption to protect sensitive information on the server.

The OPC UA Client initiate communication with the server to access data. First, clients retrieve data from the server using either the request/response or publish/subscribe method. This data can be information, confirmation/error messages indicating success or failure of actions, or ongoing data updates depending on the chosen communication method.

Each client application interprets the received data and utilize it for their specific purposes, this could involve displaying data on a user interface for monitoring, performing calculations based on sensor readings, or triggering control actions based on received information.

3.1.2.3 Communication Pattern

OPC UA offers two communication patterns for data exchange between devices, each one dedicated for different.

The Request/Response method adheres to a traditional client-server model. The client initiates communication by sending a specific request message to the server. This message could be a data read request specifying the desired data point (e.g., sensor temperature) or a server method request to perform an action (e.g., turn on a machine). The server receives and processes the request message. This will retrieve data from internal storage, interact with connected devices, or performe the specified method. Once processed, the server constructs a response message containing the requested data and a confirmation/error code. Finally, the server transmits the response message back to the client. Upon receiving the response, the client interprets the data or confirmation and typically disconnects from the server. This approach is well-suited for scenarios where the client desires a communication for one-time or infrequent period.

The optional Publish/Subscribe method leverages an architecture for efficient real-time data updates. Publisher, acting as either OPC UA servers or clients, can send updates on specific data variables. These updates are triggered by changes in sensor readings, control settings, or any other relevant data points. The publish message contain the data variable identifier, the updated data value, and a timestamp to the OPC UA Server. The servers create a subscription service, where clients can register via a subscription request to receive updates for specific data variables published by providers. When a data provider publishes an update on a subscribed variable, the server efficiently transmits a notification message to all subscribed clients. This notification message includes the updated data value, the corresponding data variable identifier, and a timestamp. Clients periodically receive these notifications, ensuring they possess the latest information on subscribed data variables. This method is ideal for real-time data monitoring applications where clients require constant awareness of changing data.

3.1.3 *MQTT*

MQTT (Message Queue Telemetry Transport) is a lightweight messaging protocol designed for resource-constrained environments, commonly found in the context of M2M and IoT communication. It utilizes a publish/subscribe architecture for efficient data exchange.

Currently there are two main versions of the protocol standardized. The MQTT 3.1.1 is the most widely adopted version and offers a robust foundation for M2M communication, It supports features like Quality of Service (QoS) levels to ensure message delivery reliability, message retention for offline subscribers, and wildcards for subscribing to broader categories of topics.

MQTT 5.0 is the newer version that builds upon the core functionalities of MQTT 3.1.1 and introduces several enhancements. These include improved message delivery performance, enhanced security mechanisms, and support for additional data types. However, adoption of MQTT 5 is still evolving as devices and applications need to be updated to leverage its new features.

3.1.3.1 MQTT Clients

An MQTT publisher acts as a data producer. The publisher creates messages containing the data. This data can be sensor readings, device status updates, or any relevant information. Crucially, the publisher attaches a "topic" to each message. This topic acts as a label, categorizing the data and signifying its meaning. For instance, a temperature sensor reading would be using the topic "room/temperature.". The publisher sends its message, complete with the chosen topic, to a central hub called broker.

An MQTT subscriber acts as the information receiver, they are selective listeners withing the network. The subscriber expresses its interest in specific data by subscribing to their topic on the broker, sending a registration message.

The MQTT broker acts as the central system on the communication network. As the central hub, the broker will receive all the data sent by all the publishers. The broker's critical function is to efficiently route the data, filtering by their topics label, to the corresponding registered subscriber. This targeted delivery eliminates the need for direct communication between publishers and subscribers, and reducing network traffic as subscribers only get the data they truly require for their operations

3.1.3.2 MQTT Topic & Payload

The labelling system for all data messages, the MQTT Topic, is a string that categorizes the data and specifies its meaning within the MQTT network. Similar to how a mail address specify where to deliver a letter, the topic specify the MQTT broker which subscribers are interested in receiving the data. Topics can be hierarchical, using forward slashes (/) to separate levels. For instance, "factory1/sensor3/humidity".

The MQTT payload carries the useful data being transmitted. This data can be diverse, ranging from sensor readings (temperature, pressure) to device status updates (on/off) or control commands. The MQTT protocol itself doesn't dictate the format of the data within the payload. This allows for flexibility in how devices and applications represent their information. Common payload formats are plain text, JSON, or binary data, depending on the specific use case.

3.1.3.3 QoS Levels

MQTT does not provide a unique Quality of Service (QoS) as the TCP/IP guaranteed delivery. It provides more options to resource-constrained environments where balance between reliability and efficiency might be necessary:

- **At Most Once (QoS Level 0):** It prioritizes efficiency over guaranteed delivery. Publishers send messages to the broker, and the broker attempts to deliver them to subscribed clients. However, MQTT doesn't confirm successful delivery. This approach is suitable for scenarios where occasional data loss is acceptable, and minimizing network traffic is crucial.
- **At Least Once (QoS Level 1):** It offers a balance between efficiency and reliability. Publishers send messages to the broker, and the broker acknowledges receipt. However, the message might be delivered more than once due to retransmissions if the initial delivery attempt fails. This approach is suitable for scenarios where some data duplication is tolerable, but ensuring messages are eventually received is important.
- **Exactly Once (QoS Level 2):** It provides the highest level of delivery guarantee. Publishers send messages to the broker, and a complex handshake process ensures the message is delivered exactly once to each subscribed client. This approach is ideal for scenarios where data loss is critical, but it comes at the cost of increased network traffic and processing overhead.

3.1.4 *Sparkplug*

Sparkplug is an open-source specification hosted by the Eclipse Foundation, aiming to seamlessly integrate data from various sources like sensors, devices, and applications within the MQTT protocol for Industrial IoT (IIoT) applications. The specification is freely available for anyone to copy and distribute.

This section presents a summary of the Sparkplug Specification 3.0.0 [10], the latest publication at the time of writing. The document outlines the structure and implementation requirements for MQTT clients and applications adhering to the Sparkplug specification, both for Edge Nodes and Host Applications.

3.1.4.1 Fundamental concepts

While MQTT has become a dominant messaging protocol across both IT and OT domains, it does not standarize mechanisms for data topic naming and payload structure. Sparkplug addresses this gap by providing a well-defined approach, promoting interoperability and simplifying development for IIoT solutions. This translates to a better user experience and less time-consuming deployment for managing industrial automation systems.

Sparkplug is not intended to replace OPC UA as the unique industrial communication protocol. In existing industrial facilities, OPC UA polling engines remain valuable for accessing raw process data. However, Sparkplug offers several advantages in certain scenarios. The followings are mentioned by their authors [10]:

- Sparkplug provides a pure device-to-OT publish/subscribe messaging paradigm, offering more flexibility and scalability compared to polling-based approaches used in OPC UA.
- MQTT brokers and Sparkplug implementations are lightweight, making them suitable for communication with resource-constrained sensor and device deployments at the network edge.
- Sparkplug is light enough for easy implementation on all types of edge devices.

To ensure compatibility and adherence to the Sparkplug specification, a list of validated Hardware and Software products is maintained by the working group [11]. These products have successfully passed the Sparkplug Technology Compatibility Kit (TCK) testing. Many ther products support Sparkplug but they are not included in the official list.

The Sparkplug MQTT Architecture introduces the concept of a Primary Host Application, often referred to a Supervisory Control and Data Acquisition (SCADA)

application in legacy architectures. This is an MQTT client responsible for subscribing to messages from Sparkplug Edge Nodes. Edge Nodes can be configured to modify their behavior based on the online or offline status of a specific Primary Host. For more details on the Sparkplug MQTT architecture, please refer to Section 3.3.2.2.

The Sparkplug specification focuses on the definition of the following three key aspects to enable the IIoT interoperability of MQTT. These three aspects are detailed in the following sub-sections.

- OT-centric Topic Namespace.
- OT-centric Payload definition optimized for industrial process variables.
- MQTT Session State management required by OT SCADA systems.

3.1.4.2 Topic Namespace

MQTT's flexibility in the topic name has contributed significantly to its widespread adoption in IoT solutions. However, this freedom can lead to inconsistencies and challenges in complex industrial environments. Sparkplug addresses this by defining a standardized topic namespace focused for SCADA and IIoT applications. By following the standardized topic, applications can easily identify and subscribe to the specific data topics they require, eliminating the need to search through topic names.

Topics are built hierarchically, with each level providing more specific information about the data. All MQTT clients adhering to the Sparkplug specification must employ the following topic namespace structure shown in the following Figure 7:

Figure 7. Topic definition from Sparkplug.

- **namespace:** The Namespace element serves as the root element, defining the structure of subsequent elements and the payload encoding. The Sparkplug specification currently supports two namespaces: “spAv1.0” for payload definition A (deprecated) and “spBv1.0” for payload definition B (active).
- **group_id:** The group ID element indicates a logical categorization of Sparkplug Edge Nodes within the MQTT server. To optimize bandwidth, group IDs should be concise and descriptive.
- **message_type:** The message type element indicates the appropriate payload data. Payload encoding varies based on the Sparkplug version specified in the

namespace element. Detailed specifications for message types can be found in Table 1.

- **edge_node_id:** The Edge Node ID identifies a Sparkplug Edge Node within the infrastructure. This element is included in every published message and should be kept minimal in length.
- **device_id:** The optional Device ID element identifies a device connected to the Sparkplug Edge Node. It is excluded for messages originating from or destined to the Edge Node itself.

3.1.4.3 Payload definition

Similar to its approach for topic namespaces, MQTT leaves undefined the payload format and encoding. Sparkplug introduces specific encoding mechanisms that align with MQTT's core principles of lightweight, low-latency communication while incorporating features essential for industrial applications. Sparkplug topic "namespace" element indicate the payload encoding scheme used by an Edge Node.

Historically there have been two Sparkplug defined payload schemes. The first one was Sparkplug A, and the second Sparkplug B. Sparkplug B addressed a few issues that were present in the A version. Due to lack of adoption and the fact that Sparkplug B was available shortly after the first version, the Sparkplug A definition is no longer supported and omitted from official documentation. For consistency, all subsequent mentions of Sparkplug will refer to the Sparkplug B payload schema.

The following Table 1 shows the group of Message Types defined by Sparkplug B, who originates the message and a brief description of itself.

Table 1. Sparkplug B Message Types.

Message Type	Producer	Description
NBIRTH **D**BIRTH	Edge **N**ode or **D**evice	Indicates the online status of an Edge Node or Device
NDEATH **D**DEATH	Edge **N**ode or **D**evice	Indicates the online status of Edge Node or Device
NDATA **D**DATA	Edge **N**ode or **D**evice	Contains time-based metric values of Edge Node or Device that have changed
NCMD **D**CMD	Application	Delivers updated metric values to Edge **N**ode or **D**evice

STATE	Primary Host Application	Reports the online/offline status of the Primary Host Application

3.1.4.4 Session State Management

While MQTT supports session state management, its full potential often remains unused in many implementations. Sparkplug aims to leverage this feature to its maximum advantage in real-time SCADA and IIoT scenarios. Traditional SCADA systems often rely on legacy poll-response protocols to maintain device connectivity awareness. This approach can be inefficient and resource intensive.

Sparkplug makes use of MQTT's built-in "Will Message" feature to provide robust session state management**.** By defining specific BIRTH and DEATH topics and payloads, in conjunction with MQTT's keep-alive mechanism, Sparkplug ensures that other MQTT clients, such as SCADA hosts, can accurately track the status of connected devices.

As an example of session state management, the following outlines the process for establishing a Sparkplug Edge Node's state:

- The Edge Node client initiates a connection to available MQTT servers using the MQTT CONNECT control packet. The NDEATH certificate is concurrently set as the Last Will message, to be published upon unexpected disconnection. *
- Subsequently, the client subscribes to NCMD, DCMD, and STATE topics. The initial online status is declared through the publication of an NBIRTH message. *
- In case of an abrupt connection loss, the MQTT server broadcasts the pre-defined NDEATH message, signaling the Edge Node's offline state. Consequently, all devices associated with the Edge Node are implicitly considered offline by the application.

While any connected application can subscribe and receive the state information, the Primary Host Application is mainly responsible for tracking online/offline status. This application subscribes to all BIRTH and DEATH topics by default.

For in-depth details on other session establishment, termination, and complex scenarios such as out-of-order messages, please refer to the latest Sparkplug specification.

3.1.5 *Apache Kafka*

Apache Kafka, ofter referred as just Kafka, is a widely adopted, open-source platform designed for processing and managing high-throughput streams of data. It is employed for

critical applications such as data pipelines, real-time transformations, and data integration [12].

Kafka has gained widespread adoption across multiple industries, with a user base of thousands of companies, including over 80% of the Fortune 100 group. Their user's sectors include technology (Cisco), finance (Goldman Sachs), retail (Target) and many others, demonstrating its versatility and applicability to a broad range of organizational needs [13].

3.1.5.1 Event-Driven Architecture

Kafka did not create the term "event-driven architecture," but it played a pivotal role in popularizing and implementing it at scale.

Event-driven architecture is a software architectural paradigm where systems communicate and react to events. Unlike traditional request-response models, where systems directly request information from each other, an event-driven architecture involves systems publishing events and other systems subscribing to those events. This asynchronous approach promotes scalability, resilience, and decoupling between system components [14].

The origin of Kafka lies in the challenges faced by LinkedIn during its rapid growth over a few years [15].

Traditional request-response architectures struggled to efficiently handle some applications, particularly those triggered asynchronously by other system activities. At the same time, managing data flow between multiple systems (databases, monitoring tools, etc.) relied on custom pipelines created as they were needed. This approach was time-consuming, error-prone, and not scalable. Each pipeline presented unique challenges, sacrificing scalability, real-time processing, or throughput. This data flow of pipelines is represented in the following Figure 8.

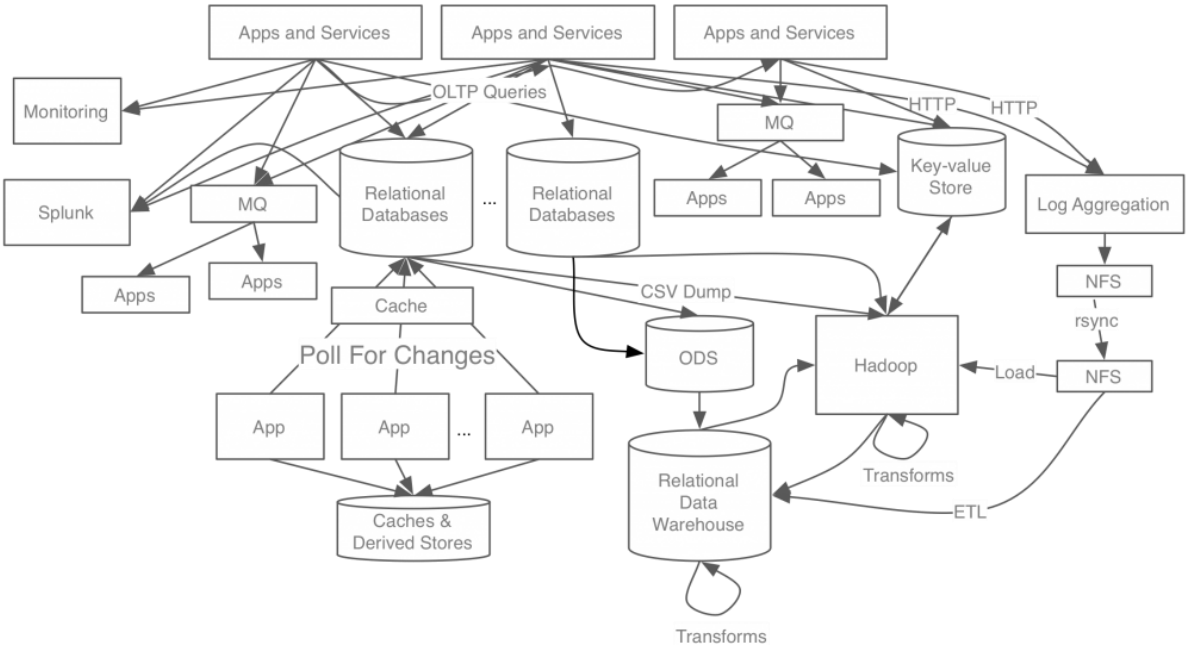


Figure 8. Traditional data flow with direct pipelines.

Source: https://www.confluent.io/blog/event-streaming-platform-1/

Recognizing these limitations, in 2010 LinkedIn started to develop a platform capable of handling massive data volumes between applications while providing real-time storage and processing. This led to the concept of "stream data" and the creation of a central "event-streaming platform," as now is defined Kafka. A visual representation of this system is shown in the following Figure 9.

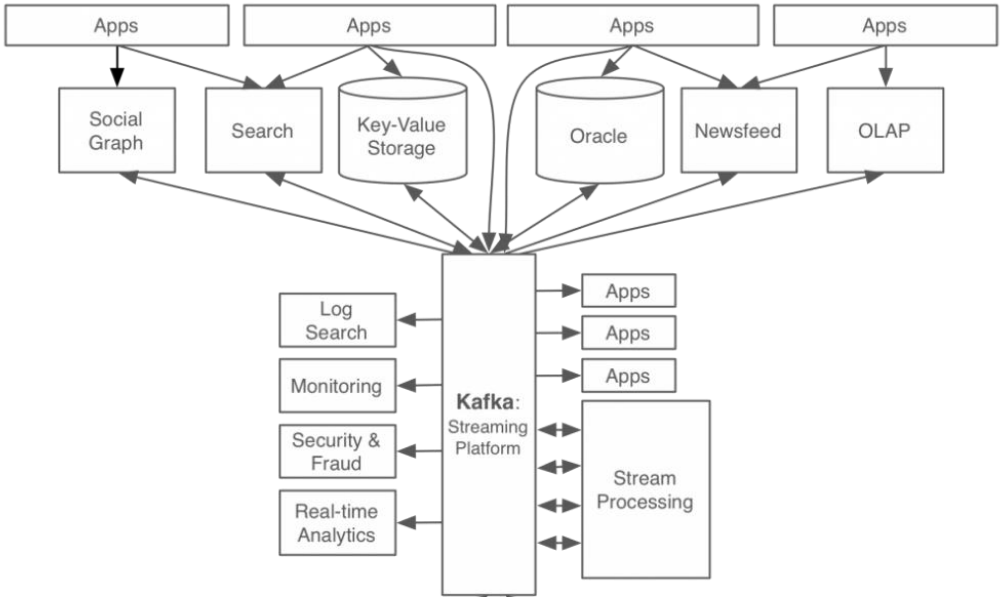


Figure 9. Data flow with stream-centric system.

Source: https://www.confluent.io/blog/event-streaming-platform-1/

Today, Kafka handles trillions of events daily at LinkedIn and has become the backbone for data flow between systems. Its open-source nature has led to widespread adoption across different industries [16]

Kafka typically operates as a cluster of multiple brokers to enhance redundancy and scalability. Each broker serves as a core component, responsible for storing and managing messages. To efficiently manage the cluster, one broker is designated as the controller. The controller handles critical tasks such as partition leader assignment and failure recovery.

3.1.5.2 Topics, partitions and schemas

The fundamental unit of Kafka are messages. They are categorized into Topics, analogous to database tables. To distribute data efficiently and enable scalability, topics are divided into Partitions. Each partition is an ordered, append-only, inmutable sequence of messages.

Kafka distributes partitions across multiple brokers for redundancy and performance. This partitioning allows horizontal scaling to handle massive data volumes. Kafka's data retention policies enable time-based (e.g. 7 days) or size-based (e.g. 5 GB) management for data expiration. The following Figure 10 shows a representation of a Kafka Topic with Partitions in multiple Brokers.

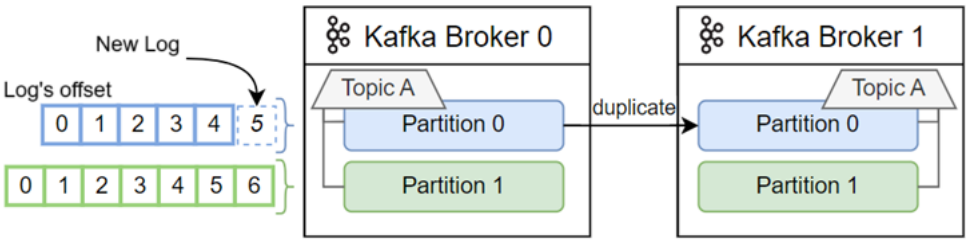


Figure 10. Diagram of a Kafka Topic with Partitions in multiple Brokers.

While Kafka stores messages as simple byte arrays, defining a Schema for message content is highly recommended. Schemas provide structure, ensuring data integrity and facilitating understanding. For example, if a message received does not follow the Schema, the message is discarded. Supported Schema formats include JSON, Apache Avro, and Protobuf.

3.1.5.3 Kafka Clients

Kafka clients used by applications or microservices to interact with the Kafka cluster to write or read data from Topics. There are two main client definition by functionality.

Producers are clients that create and send messages to Kafka topics, often referred to as "publishers" in other architectures. They can distribute messages evenly across partitions or specify target partitions.

Consumers are clients that read messages from topics, often referred to as "subscribers" in other architectures. They read the data in the order they were produced within each partition. Kafka maintains the Consumer's offsets to track their last message readed, enabling consumers to stop and restart, fault-tolentant, without data lost.

The main way to create producers and consumers is by using client libraries available for various programming languages like Java, C/C++, Python or Go, with a bigger variety of supported languages by the open-source community such as Node.js, Rust, or PHP, among many others.

3.1.5.4 Kafka Connect

A Kafka broker is usually supported by other Kafka components that typically runs in different services, the most important components are Kafka Streams for building complex processing applications, Kafka Schema registry to manage the schema of data inside of topics, and Kafka Connect.

Kafka Connect is a crucial component within the Kafka ecosystem that facilitates data pipeline integration between the Kafka platform and external systems. It serves as the tool to enable the transfer of data into and out of Kafka from various sources and destinations.

Unlike traditional producer and consumer clients that operate in applications or microservices, Kafka Connect handles the complexities of integrating with external platforms. It pulls data from external sources into Kafka or pushes data from Kafka to external endpoints, providing a flexible and scalable solution.

The main key of Kafka Connect are connectors, specialized plugins designed to interact with specific data systems. A robust set of connectors already exists, giving support to a wide range of databases, systems, cloud services, and other data sources and sinks. These connectors already developed makes easier the integration process, often only requiring minimal configuration by the user.

Kafka Connect operates through a distributed architecture involving workers and tasks. Workers are processes that manage the execution of connectors, while tasks represent individual units of work within a connector. This distributed approach enhances scalability and fault tolerance.

Kafka Connect includes the Single Message Transformation feature, which transforms records while they are being copied from a source to Kafka, or from Kafka to a target. This includes routing messages to different topics, filtering messages, chang- ing data types, redacting specific fields, and more. More complex transformations that involve joins and aggregations are typically done using Kafka Streams.

3.2 Technologies for Architecture

3.2.1 *Python*

Python is a high-level, interpreted programming language known for its readability and versatility. Its simplicity and intuitive syntax have contributed to its widespread adoption across various domains, including data science, web development, automation, and scientific computing.

Python's dynamic typing and object-oriented nature offer flexibility and efficiency, making it a preferred choice for both beginners and experienced programmers. Its extensive standard library, coupled with a vast ecosystem of third-party packages, provides a rich set of tools for solving complex tasks.

Visual Studio Code, sometimes refers to as VSCode, is a free and popular code editor from Microsoft. It is a great choice for Python development. One of its most valuable features is the ability to create and manage virtual environments. Virtual environments provide a self-contained space where it can be installed and managed Python packages specific to each project. This prevents conflicts between projects that might require different dependencies. It is possible to easily install necessary packages using tools like pip and share these environments with others to ensure consistent setups across machines.

3.2.2 *Docker*

Docker is a platform designed to create, deploy, and run applications within standardized units called containers. These containers package up code and all its dependencies, such as libraries, system tools, system libraries, and settings, ensuring that applications run quickly and reliably from one computing environment to another.

Essentially, Docker provides a consistent and isolated environment for applications, making it easier to develop, test, and deploy them across different systems. By using Docker, developers can focus on writing code without worrying about underlying infrastructure complexities.

Docker has revolutionized the way applications are developed, deployed, and managed. By encapsulating applications and their dependencies into isolated environments, Docker offers several advantages.

Containers provide a consistent and reproducible execution environment, ensuring applications behave identically across different systems. This isolation also enhances security by preventing conflicts between applications. Furthermore, Docker's lightweight nature, compared to traditional virtual machines, optimizes resource utilization. In the following Figure 11 it is shown the how Docker containers are isolated from each other and from the host operating system, providing a consistent and portable environment for applications.

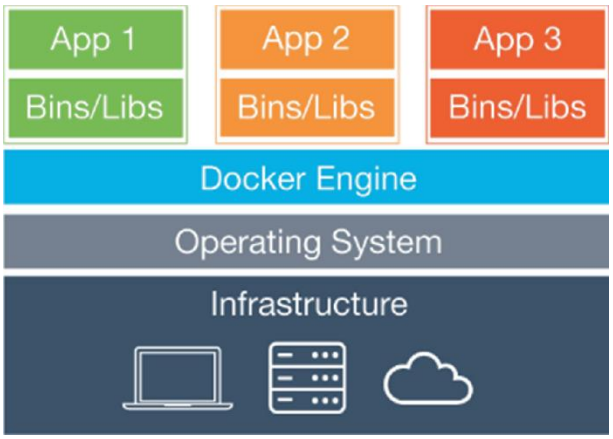


Figure 11. Docker containers architecture.

Source https://link.springer.com/chapter/10.1007/978-3-031-07012-9_54

However, adopting containerization comes with its own set of challenges. Managing complex containerized applications requires expertise knowledge.

3.2.2.1 Fundamental concepts of Docker

In the Docker context there are three fundamental concepts: containers, images, and volumes. An image serves as a read-only template, encapsulating the application code, libraries, tools, and configurations required to execute an application. It functions as a blueprint from which containers are instantiated.

A container represents a running instance of an image. It provides an isolated environment for the application to operate. Multiple containers can be generated from a single image, each one working independently. Containers are inherently transient; their lifespan is contingent upon their execution.

To manage persistent data, Docker introduces volumes. Unlike containers, which are ephemeral, volumes persist data beyond the container's lifecycle. This mechanism is crucial

for storing application data, databases, and other critical information. By decoupling data from the container, volumes enhance data management and flexibility.

3.2.2.2 Docker Hub

Docker Hub is a central repository for discovering, sharing, and managing container images [17]. It functions as a platform where users can locate pre-built images for various applications, ranging from operating systems to databases and programming languages.

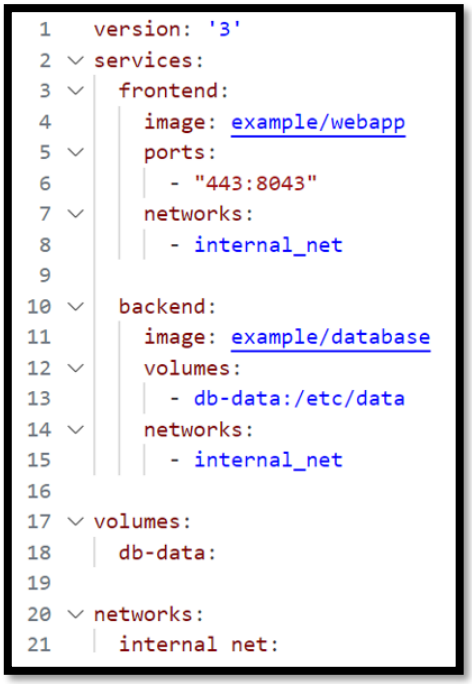
Docker Hub offers a rich collection of container images categorized for user convenience:

- **Official Images**, maintained by Docker, represent the most trustworthy source. They often serve as the foundation upon which custom images are built.
- **Verified Publisher Images**, published by Docker partners, also provide a high level of quality and security.
- **Community Images**, created and maintained by the community, offer a vast array of applications and tools. The level of quality and support may vary.
- **Docker-Sponsored Open-Source Images**, created by the community but they benefit from official Docker's support and promotion.

3.2.2.3 Docker Compose

Docker Compose is a tool designed to define and run multi-container Docker applications. Users can create and start all containers defined with a single command. Moreover, Docker Compose facilitates the orchestration of multiple containers, enabling them to communicate and interact effectively.

It simplifies the management of complex applications by utilizing a YAML configuration file known as the ***docker-compose.yml*** file. This file serves as the blueprint for the application's architecture. It defines services, which represent individual containers, and their configurations. In the following Figure 12 it is shown a basic configuration file as example.



```yaml
version: '3'
services:
  frontend:
    image: example/webapp
    ports:
      - "443:8043"
    networks:
      - internal_net

  backend:
    image: example/database
    volumes:
      - db-data:/etc/data
    networks:
      - internal_net

volumes:
  db-data:

networks:
  internal net:
```

Figure 12. Example Docker Compose File.

- **Version**: Indicates the Docker Compose file format version.
- **Services**: Define each container application and their components, specifying the image to use, environment variables, port mappings, volumes, and dependencies.
- **Volumes**: Define data volumes for persistent storage.
- **Networks**: Define networks for the the container internal communication.

3.2.3 *KEPserverEX*

KEPserverEX is a robust industrial connectivity platform that functions as an OPC server software. It serves as a critical bridge, facilitating communication and data exchange between diverse industrial devices and protocols. By acting as a central hub, KEPserverEX enables seamless integration of disparate automation systems [18].

KEPserverEX boasts a vast library of drivers, enabling connectivity to a wide range of industrial devices and protocols, including OPC UA, Modbus, and many more. Also, it can seamlessly integrate with various HMI/SCADA systems, historians, and enterprise applications to optimize overall system performance.

KEPserverEX provides robust tools for data transformation, allowing users to convert data formats, scale values, and perform calculations with given data to create more advanced data. The following Figure 13 shows the user interface of KEPserverEX main screen.

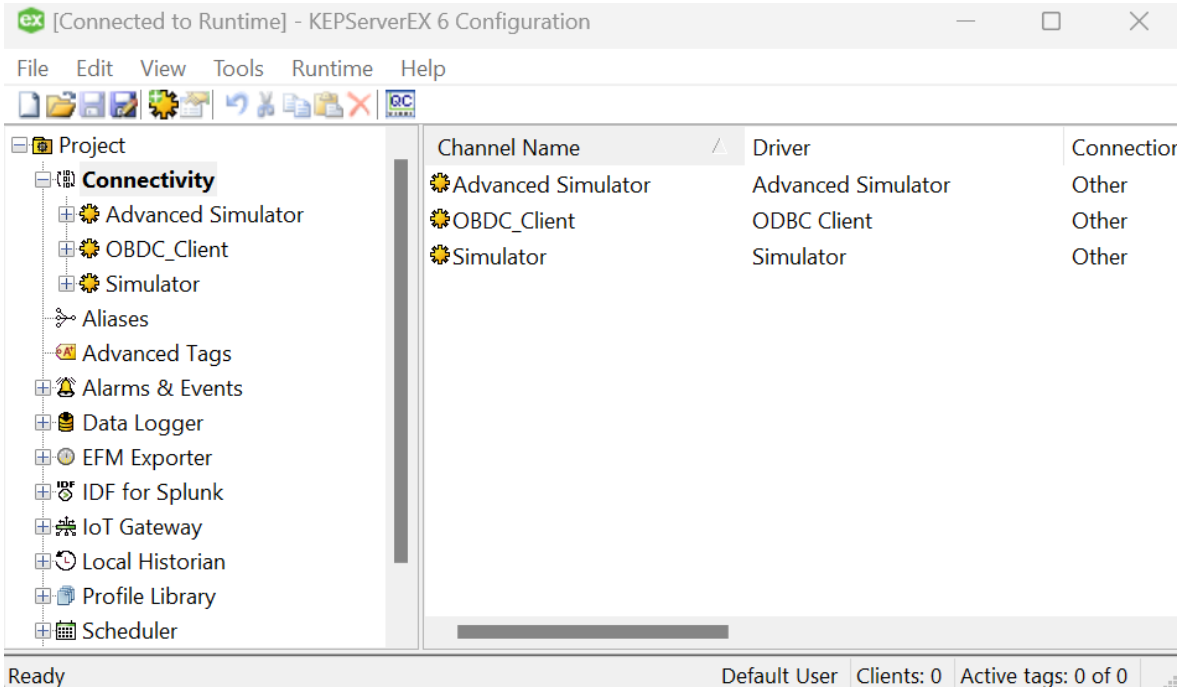


Figure 13. KEPserverEX main interface.

While KEPserverEX is a powerful and versatile platform, it's essential to consider certain factors when implementing it. The performance of KEPserverEX can be influenced by factors such as the number of connected devices, data volume, and network conditions. Implementing and configuring KEPserverEX can involve a complex learning curve, especially for users without extensive IT or automation experience. KEPserverEX operates on a licensing model, with different subscription types offering varying levels of functionality and connectivity options. In the following section it is described the free trial license and its limitations.

3.2.3.1 Free trial license limitations

KEPserverEX operates on a licensing model, with different license types offering varying levels of functionality and connectivity options. The KEPserverEX free trial version is designed to provide users with a comprehensive experience of the platform's capabilities without long-term commitment. However, it comes with certain limitations:

Time Restrictions. The most significative limitation is a restricted runtime. The free trial for only operates for periods of two hours. After this period, the license must be manually reinitialized using the Runtime menu. While this allows for evaluation, it can be inconvenient for extended testing or proof-of-concept projects. Figure 14 shows the license warning for limited usage.

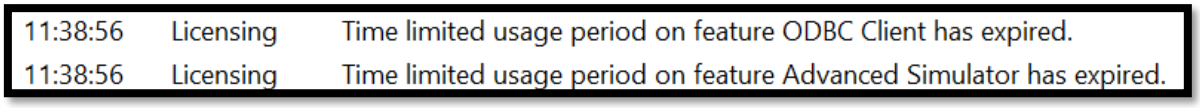


Figure 14. KEPserverEX limited usage license.

There are no limitations regarding functionality, data volume, or tag count within the demo period. Users can fully explore the platform's capabilities without restrictions during the active session [19].

3.2.4 *OPC Router*

OPC Router is an OPC UA client software designed to facilitate seamless data exchange between various industrial systems and devices. It acts as a central hub, connecting disparate components and enabling smooth data flow within an industrial environment.

By serving as a middleware solution, OPC Router bridges the communication gap between different protocols, systems, and platforms. This interoperability is crucial for modern industrial architectures, as it allows for efficient data integration and utilization across the entire enterprise.

As a summary, OPC Router optimizes data management by aggregating, filtering, transforming, and routing data to its intended destinations. This centralized approach enhances data accessibility, improves system performance, and supports data-driven decision-making.

3.2.4.1 Fundamental concepts of OPC Router

In order to fully understand the later working flow of OPC Router, first it has to be introduced the core components of this software: Plugins and Components

OPC Router's connectivity is extended through the use of plugins. They provide connectivity to specific data sources and destinations, enabling the integration of various industrial protocols and systems. Plugins act as adapters, translating data between different formats and communication protocols. This modular approach allows for flexibility and customization, as creating new plugin instances it can be created different data sources or destinations even for the same protocol but different configuration.

The OPC Router main interface presents a visual and intuitive approach to managing data flow and integration. Central to this interface are the concepts of Connections and Transfer Objects. In the following Figure 15 it is shown the interface of an example connection.

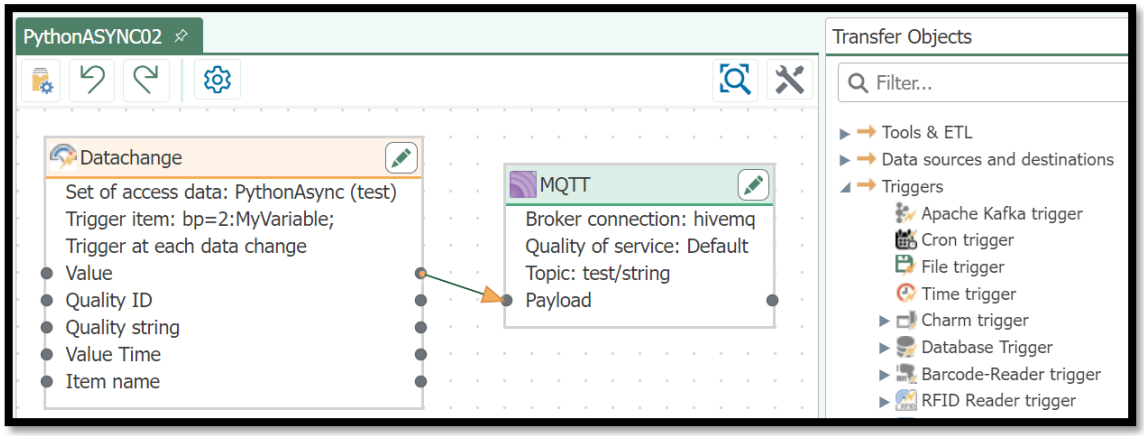


Figure 15. OPC Router "Connection" interface.

Connections represent the links between the OPC Router and external systems or devices. They serve as the entry and exit points for data. Transfer Objects, visualized as blocks or nodes in the interface, are responsible for the data transformation and routing logic. They are the core components responsible for processing and directing data flow within the OPC Router and external systems. Key types of Transfer Objects include:

- **Tools and ETL:** Perform specific data manipulation tasks, like filtering, transformation, or aggregation.
- **Data sources and destinations:** Represent the origin of data, such as PLCs, sensors, or databases, or the target systems for processed data, such as SCADA systems, historians, or cloud platforms.
- **Triggers**: These are the objects that will control the connection execution, setting a trigger alarm depending on the specific object target. Most of them include output data that is used for transformation and payload data for the connection.

Templates in OPC Router provide a mechanism for creating and managing multiple similar connections efficiently. By defining a template with variable parameters, users can create multiple instances of the connection with different configurations. They allow for the use of variables to define dynamic configuration options, such as OPC UA addresses, filter criteria, and destination settings. A single template can be used to create multiple instances of the connection, each with its own unique configuration.

3.2.4.2 Free Trial Version

The OPC Router trial version offers a comprehensive evaluation experience without significant limitations on features or data volume. However, there is a time-based restriction, requiring users to restart the software manually after a period of 2 hours of working.

3.2.5 *HiveMQ*

HiveMQ is a robust and scalable MQTT broker designed to handle the demands of modern IoT applications. It serves as a central hub for facilitating communication between IoT devices, applications, and cloud services using the MQTT protocol.

The broker offers robust authentication and authorization mechanisms to control access and protect sensitive data. Encryption is also supported to safeguard data both in transit and at rest. HiveMQ supports message persistence, ensuring that messages are not lost even if the broker restarts or experiences network interruptions. Also, it can seamlessly integrate with popular cloud platforms as Amazon Web Services, Azure, and Google Cloud Platform.

Beyond these core features, HiveMQ offers advanced capabilities like MQTT 5.0 support, quality-of-service levels, and wildcards for topic filtering. It also provides a plugin system for customization and integration with external systems.

3.2.5.1 HiveMQ Community Edition

While the HiveMQ Community Edition offers a generous trial period without limitations on features or data volume, there are some restrictions to consider. The community edition has a limit on the maximum number of concurrent client connections it can handle. This limitation could impact performance in scenarios with a large number of connected devices. Also, the community edition is intended for non-commercial use. For commercial deployments, an enterprise license is required.

By understanding these potential restrictions, users can make decisions about whether the community edition meets their specific requirements or if an enterprise license is necessary.

3.2.6 *Confluent*

Confluent is a leading provider of streaming data platforms, specializing in Apache Kafka. They offer two primary products: Confluent Cloud and Confluent Platform.

Confluent Cloud is a fully managed Kafka service hosted on the cloud. It provides a hassle-free way to deploy and manage Kafka clusters without the need for on-premises infrastructure. Confluent can automatically scale up or down based on your data throughput requirements, ensuring optimal performance and cost-efficiency. It is available in multiple

regions worldwide, providing low-latency access to your data. Confluent Cloud is ideal for organizations that want to leverage Kafka without the operational overhead.

Confluent Platform is an installable software platform that includes Apache Kafka, Confluent Control Center, and other components. It gives more control over the Kafka cluster and offers flexibility for on-premises or hybrid deployments as it is deployed in your own enviroment. Confluent Platform is suitable for organizations that require more customization and control over their Kafka environment.

3.2.6.1 Differences with Apache Kafka

In essence, Confluent offers a managed and simplified experience on top of a Kafka Broker, while Kafka provides greater flexibility and control but requires more self-management. There are some key distinctions between the two that are important to define.

- **Cloud Managed.** Confluent Cloud is a managed service, meaning Confluent handles the infrastructure and management of the Kafka cluster. Apache Kafka, on the other hand, requires users to manage and operate the cluster themselves.
- **Additional Features.** Confluent includes additional features beyond the core Kafka functionality. These features can include a control center, data governance tools, and integration with other systems.
- **Private Support.** Confluent provides comprehensive support and management services for its products, including monitoring, troubleshooting, and security. With Apache Kafka, users are responsible for their own management and support.
- **Cost.** Confluent typically runs with a subscription-based pricing model, while Apache Kafka is open-source and can be deployed on-premises at no cost.

3.2.6.2 Pricing and Deployment Options

Confluent Cloud offers three subscription tiers: Basic, Standard, and Enterprise. The Basic tier is free and suitable for testing and evaluation, but it has limitations in terms of data throughput and features. The Standard and Enterprise tiers provide more robust capabilities and are designed for production environments.

Confluent Platform requires a license for production environments, but it can be downloaded and installed for free. The platform can be deployed using Kubernetes or Docker containers, offering more flexibility in deployment options. Confluent Platform is

not a standalone service, but it includes multiple components to provide an improved Kafka solution.

For detailed information on pricing, licensing, and deployment options, please refer to the official Confluent pricing webpage [20]

3.2.7 *ThingsBoard*

ThingsBoard is an open-source IoT platform that offers a comprehensive set of tools for managing IoT devices, data, and applications. It provides a platform for collecting, processing, and visualizing IoT data, making it a valuable asset for organizations looking to leverage IoT technologies [21].

3.2.7.1 Thingsboard Versions

ThingsBoard offers two main versions: Community and Professional. While both versions share a core foundation, they differ in terms of features, support, and pricing.

ThingsBoard Community Edition is a freely available, open-source platform for managing IoT devices. It provides essential features but offers limited support through community forums and online resources. This makes it suitable for small-scale IoT projects, experimentation, and learning.

ThingsBoard Professional Edition (PE), on the other hand, requires a subscription and offers additional features beyond the Community Edition. It also provides dedicated support channels, making it ideal for large-scale IoT deployments, mission-critical applications, and organizations demanding enterprise-grade features and support.

While both versions have numerous differences, this thesis focuses on one key feature: platform integrations. The Community Edition does not allow for platform integrations, while the Professional Edition offers a wide range of common platform integrations, including Google Pub-Sub, AWS IoT, TheThingStack, and Kafka. This feature was a crucial factor in differentiating between the two versions and selecting the appropriate choice for the specific needs of this thesis.

3.2.7.2 Rule Chain

Rule chains in ThingsBoard are a fundamental component of its event processing engine. They allow you to define complex workflows that can process, transform, and route incoming data from IoT devices. The visual representation of them is as a series of interconnected nodes, each performing a specific task. There are three main components:

1. **Messages:** Any incoming event, such as data from devices, device lifecycle events, REST API events, RPC requests, and more.
2. **Rule Nodes:** Functions that are executed on incoming messages. A variety of node types exist to filter, transform, or perform actions on messages.
3. **Rule Chains:** A sequence of connected rule nodes. Outbound messages from one rule node are sent to the next connected rule node, forming a workflow.

The common functional flow of a Rule Chain can be understood as follows. A message from an IoT device enters the rule chain. The message is processed by the first rule node in the chain. This node might filter, enrich, or transform the message. Depending on the outcome of the processing, the message may be sent to the next node in the chain or routed to a different rule chain. The output of the rule node continues to flow through the rule chain, being processed by each node until it reaches the end or is terminated.

Root Rule Chain in ThingsBoard is a special type of rule chain that serves as the entry point for all incoming messages. Every message received by the ThingsBoard platform will first pass through the root rule chain before being routed to other rule chains based on filtering and routing logic. In the following Figure 16 it is shown the common interface of a Root rule chain.

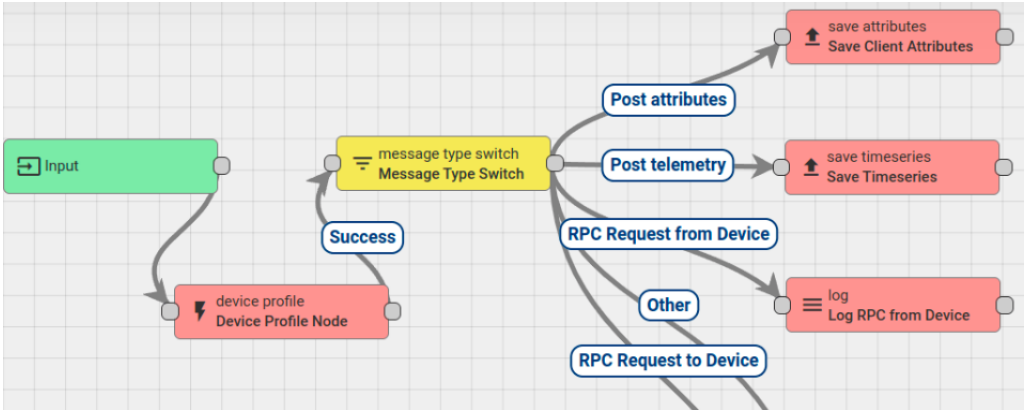


Figure 16. ThingsBoard default Root Rule Chain.

3.3 Unified Namespaces

This sub-chapter begins by clarifying the ambiguity surrounding the definition of UNS, highlighting various interpretations from industry experts and vendors. Following this, it delves into different UNS implementation architectures, focusing on three prominent approaches. For each implementation there is a discussion on their benefits and pitfalls. Finally, it describes general challenges encountered during UNS implementation.

3.3.1 *Definition*

Unified Namespaces has become a concept gaining significant traction within the IIoT landscape. However, despite its association with the "revolution" of Industry 4.0, UNS currently lacks a universally accepted definition or formal standardization by established bodies like IEEE or ISA. This ambiguity leads to confusion and inconsistencies in how UNS is interpreted and implemented across different projects. Furthermore, skepticism exists within the field, with some experts questioning its transformative impact, citing minimal changes observed in Industry 4.0 over the past decade [22].

While Unified Namespaces are still hard to define precisely, there are a variety of interpretations and approaches that have been proposed.

Walker Reynolds, CEO of Intelligic 4.0 solutions and one of the main popularizers of the term UNS, defines it as *"The structure of the business and all the events. A single source of truth for all data and information of the business. The place where the current state of the business lives. The hub through which the smart things in the business communicate with one another. The architectural foundation of the Industry 4.0 and Digital Transformation initiative"* [23].

Propietary UNS software vendors, also called sometimes as IT/OT integration platforms, offer their own interpretations. HighByte, for example, describes it as *"one of the fastest-growing design patterns for integrating industrial data into a scalable and accessible structure that provides a real-time view of operations across the business"* [24]. As another example, the open-source platform, Unified Manufacturing Hub (UMH), defines UNS as *"A single source of truth for all your enterprise and shopfloor data, in an event-driven architecture with a consistent and standardised data model"* [25].

Not only the definition but also the terminology surrounding UNS lacks consistency. From prominent names in the industry like John Rinaldi, CEO of Real Time Automation , refer to it as a *"philosophy"* [26]. Similarly, relevant organizations as Gartner employs the term *"design strategy"* in its research papers [27]. Others within the field commonly use the term *"architecture"* [28].

The unclear definition of UNS contributes to the spread of misconceptions, making it difficult for even experienced professionals to understand the full scope of UNS. Some experts acknowledge spending years reading resources before feeling comfortable with their understanding [29].

As a summary, these are the key characteristics to understand what a Unified Namespace is:

1. **Single Source of Truth:** UNS aims to provide a unique centralized and consistent data representation and format for all the data across different devices and systems, simplifying data management.
2. **Standardized Data Model:** Ideally, in a UNS there should be a data model specified to organize data in a hierarchical structure of the business. Industrial standards like ISA-95 provides a solid format to use as a guideline. However, for some organizations, does not reflect the true structure of their businesses.
3. **Event-driven Architecture:** Implementations of UNS using an event-driven architecture facilitates communication by reacting to real-time events generated by connected devices and systems.
4. **Improve Interoperability:** By enforcing standardized communication protocols and data structures, UNS aims to improve the ability of various systems to seamlessly exchange information without point-to-point communications between different systems.
5. **Architectural Foundation:** UNS is often shown as the core architectural framework for future industrial automation and integration initiatives within the Industry 4.0 design.

3.3.2 *Implementation Architectures*

Unified Namespaces are often implemented using an event-driven architecture with a Message-Oriented Middelware (MOM) as the core. This approach shifts the architectural focus from device-oriented networks to the data flow that they generate and consume, promoting a more flexible and scalable system.

This section explores some of the most popular approaches for implementing event-driven architectures within the context of Unified Namespaces. Each method offers different advantages and disadvantages.

3.3.2.1 MQTT Broker Architecture

In this architecture, a Message Broker serves as the central communication hub of the UNS implementation. It facilitates data exchange between nodes within the architecture,

ensuring seamless information flow and reducing the need for complex, individual connections between systems.

The standardised Message Broker widely implemented, particularly in IoT and M2M applications, is a MQTT Broker. Its lightweight nature, low overhead, and support for publish-subscribe communication make it well-suited for the Message Broker within a Unified Namespaces. A quick introduction to MQTT is detailed in previous Section 3.1.3.

Devices, PLCs, SCADA systems, MES, ERP and any other systems act as "nodes" within the architecture. These nodes publish their data to MQTT broker, where it can be accessed and used by other nodes that subscribes to the desired topic. This creates a seamless flow of information, eliminating the need for complex, individual connections between systems. The following Figure 17 shows a typical UNS representation using a message broker.

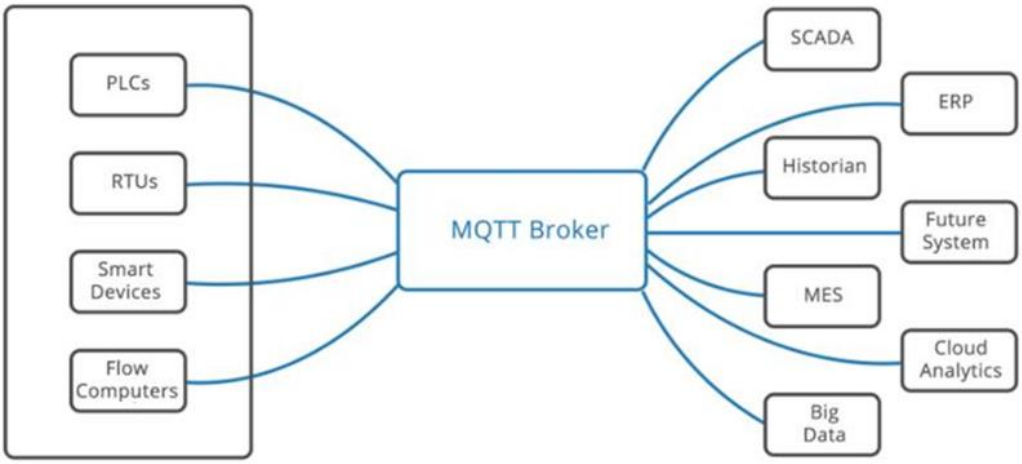


Figure 17. MQTT Unified Namespace architecture.

Adapted from: https://sparkplug.eclipse.org/

The following analysis explore the benefits and pitfalls of this architectural approach.

Benefits:

Simplified Integration: MQTT brokers offer an easy integration of new devices and applications into the UNS architecture. This reduces the complexity for both devices and the system managers, improving overall efficiency.

Scalability: MQTT brokers are inherently scalable, handling large numbers of connected devices and messages. This makes them well-suited deployments with hundreds of devices, ensuring the system can accommodate growing demands.

Clustering: MQTT brokers can operate in clusters, providing redundancy and fault tolerance for network issues. This ensures that the system remains available even if individual brokers fail.

Flexibility: MQTT supports multiple Quality of Service levels, allowing for a wide range of application requirements. This flexibility enables to balance reliability with performance and resource usage for the specific application needs.

Pitfalls:

OT Devices Integration: Many legacy OT devices only support proprietary protocols or industry standards like OPC UA and lack MQTT support. This requires IoT gateways to translate between these protocols and MQTT. Implementing IoT gateways can increase the overall cost and complexity of the system, requiring additional configuration and management.

IT Systems Integration: Many IT systems, such as MES and ERP systems, may not support MQTT yet. Developing custom microservices for integration can be necessary in some cases. As MQTT becomes more widely adopted, newer IT systems and applications are likely to incorporate support for MQTT, reducing the need for custom integrations.

Data Transformation: MQTT brokers are primarily designed for message routing and lack built-in mechanisms to understand the context or meaning of the data being transmitted. Without data structure validation, inconsistencies in the defined UNS data structure may occur.

Data Persistence: MQTT brokers typically do not store message history, making it difficult to access historical data. If a subscriber is disconnected or experiences a failure, data sent during that downtime will be lost.

Command/Response Exchange: MQTT lacks native support for transactional exchanges, such as command/response interactions. This means that does not provide a mechanism to directly indicate if a subscriber has acted upon a message [27]. Setting QoS level 2 only ensures that the subscriber received the message from the broker.

3.3.2.2 Sparkplug's proposed MQTT Architecture

As an improvement of the previous MQTT Architecture described in Section 3.3.2.1, the use of the MQTT protocol can be improved by integrating the Sparkplug specification. A detailed introduction to the Sparkplug specification is provided in Section 3.1.4.

The Unified Namespace can benefit from the Sparkplug specification in two primary ways. First, the design of the data mode structure for the UNS can be guided by the

Sparkplug's topic definition schema. Second, it can adopt the Sparkplug MQTT architecture proposed in the official specification document [10]. The following Figure 18 shows the MQTT Sparkplug Architecture.

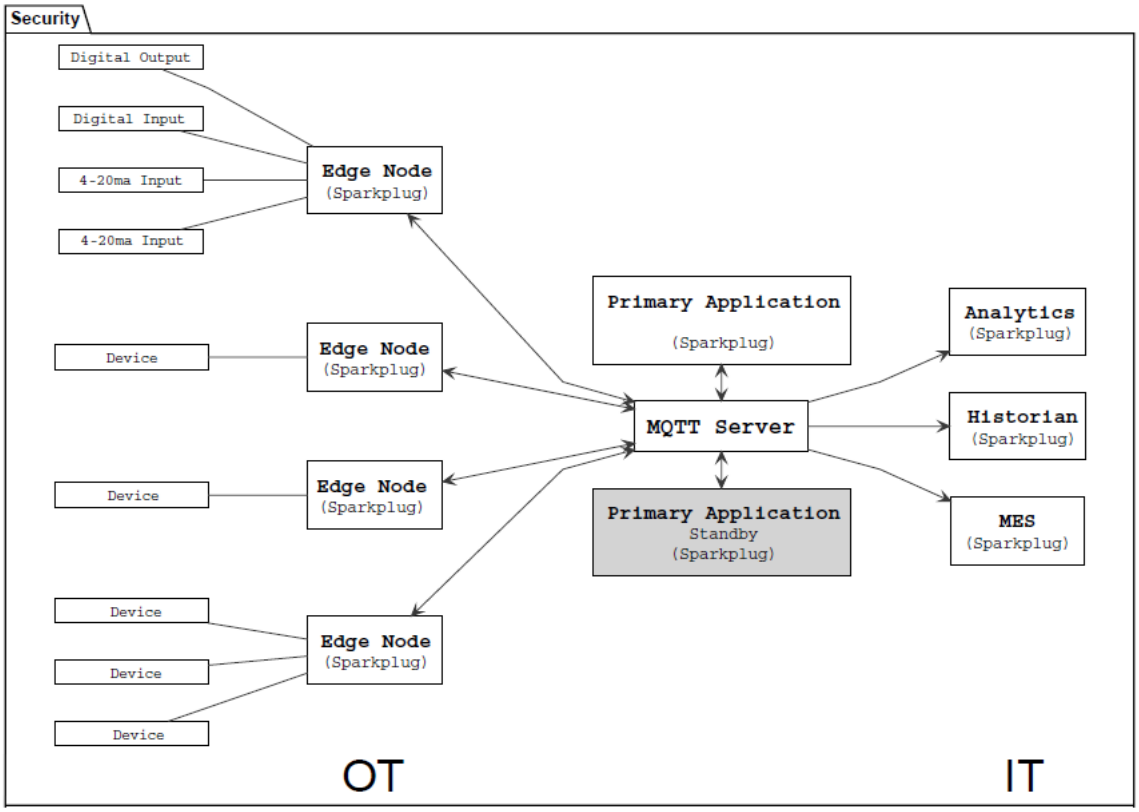


Figure 18. MQTT Sparkplug Architecture.

Adapted from: https://sparkplug.eclipse.org/specification/version/3.0/documents/sparkplug-specification-3.0.0.pdf

A comparison of the standard MQTT architecture (Figure 17) and the Sparkplug MQTT architecture (Figure 18) reveals that the core message broker in the UNS remains as a MQTT broker. However, the Sparkplug architecture introduces a new application node called the "Primary Application," which would typically be the SCADA software in an implementation.

The only requirement for implementing this architecture is that every node on the network, including devices and applications, must support the Sparkplug specification itself. This ensures utilization of all functionalities defined by Sparkplug, such as the state management by the Primary Application.

Integrating the Sparkplug specification format into the UNS architectural principles presents a specific challenge: the limited flexibility in defining the topic hierarchy. This becomes particularly evident when attempting to apply the ISA-95 data model for a

production plant hierarchy (enterprise/site/area/line/cell) to the two available parameters in the Sparkplug topic, groupID and nodeID [30].

To address the challenge of accommodating longer hierarchical structures within the Sparkplug topic namespace, two experts in the field have developed temporary workarounds. These solutions include the “Parris Method” by Matthew Parris, which includes the entire enterprise structure in the GroupID using delimiters to separate the categories, and the “Schulz Method” by David Schultz, which involves deploying multiple brokers at various levels (Area, Line, Cell) and use an IIoT platform to consume all data and republish everything to the main Unified Namespace MQTT Broker [31, 32].

Some solution vendors, after finding difficulties with integrating Sparkplug with the ISA-95 data model or extensive UNS data models, opted to define their own topic structures to accommodate more fields. For instance, the United Manufacturing Hub (UMH) data model structure discarded Sparkplug and developed its own topic definition [33].

As another solution Some experts in the field recommend to use Sparkplug only up to the SCADA level, especially when higher ISA-95 categories are required, and define a own UNS data model on top [34].

One important note to consider is **it-novum's experience.** Based on It-novum's previous Sparkplug integrations, they have not encountered the topic hierarchy limitations in their specific use cases.

The following analysis explore the benefits and pitfalls that are included to the previous MQTT Architecture by integrating the new Sparkplug support.

Benefits:

Automatic Data Discovery: Sparkplug's defined topics allow applications to automatically discover relevant data by subscribing to known topics. This simplifies the integration process and reduces the need for manual configuration.

Interoperability: Sparkplug defined topic and payload format ensures compatibility between devices and systems from different vendors.

Session Management: Integrating a primary application using Sparkplug can enable monitoring and management of edge device and node status. This provides valuable insights into the health and performance of the system.

Pitfalls:

Vendor Extensions: Some vendors may offer proprietary Sparkplug extensions, potentially leading to increased costs and reduced interoperability.

Limited Adoption: Many IT systems like MES and ERP may not yet support MQTT, requiring custom microservices for integration. However, this trend is changing as Sparkplug becomes more widely adopted.

Limited Topic Schema: Sparkplug's fixed topic structure limits customization options for the organizational data model. This can be challenging for complex UNS data models like ISA95 model.

Inefficient Communications: While applications can subscribe to desired topics and retrieve the data from one device, they receive all the parameter from that device, leading to less efficient communication in certain scenarios if only a set of parameters is needed.

3.3.2.3 MQTT & Kafka Architecture

The proposed architecture utilizes a Message Broker and an event-driven platform to establish a central hub for the UNS. MQTT, a lightweight messaging protocol designed for IoT devices, is ideally suited for connecting a large number of devices to the network. Kafka, a event-driven streaming platform, provides a scalable and reliable solution for real-time data ingestion, storage, and processing.

To clarify, MQTT and Kafka complement each other. Both have their strength and weaknesses, as it is analysed from the information in the following Table 2.

Table 2. Comparison between MQTT and Kafka.

Feature	MQTT	Apache Kafka
Network Requirements	Designed for poor connectivity and high latency scenarios (e.g., mobile networks).	Requires a stable and reliable network. High throughput.
Scalability	Can support an almost unlimited number of devices.	Limited support of thousands of connections.
Protocol Features	Most widely adopted IoT Protocol. ISO Standard	Lacks specific IoT features, such as Keep Alive.
Data Processing	No built-in data processing or integration capabilities.	Data stream processing and integration features.
Event Reprocessing	Unable to reproduce events.	Allows reprocessing of events.

Offline Capability	Clients can remain offline for extended periods.	Offers long-term log storage and buffering to decouple clients.

While MQTT and Kafka are often used in different use cases, they can complement each other to form a hybrid architecture. MQTT can act as the primary OT gateway for connecting IoT devices, Industrial devices, and IoT Gateways for legacy devices, while Kafka can handle data processing, storage, and integration with IT systems or applications.

This combined approach offers the best of both worlds: the lightweight, efficient communication of MQTT and the robust, scalable capabilities of Kafka [35]. In the following Figure 19 it is shown the data flow between MQTT and Kafka in the architecture.

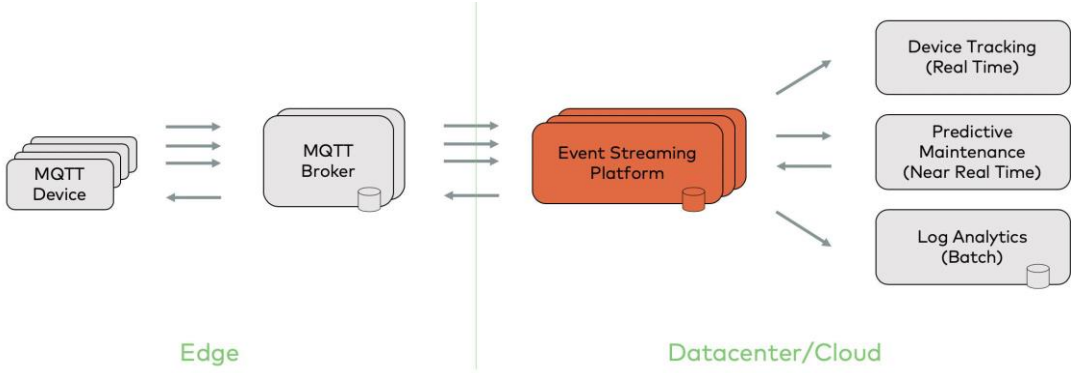


Figure 19. MQTT and Kafka architecture.

Source: https://www.confluent.io/blog/iot-streaming-use-cases-with-kafka-mqtt-confluent-and-waterstream/

The following analysis will explore the benefits and pitfalls of this architectural approach.

Benefits:

OT/IT Integration: MQTT ensures the best solution to achieve the integration of numerous OT devices, while Kakfa ensures a high throughput platform to route multiple streams of data into different IT applications.

Data Processing: By incorporating an event-streaming platform like Kafka, the limitation of being unable to process data directly within an MQTT Broker is solved.

Historical Data: Data logs can be store into Kafka topics for long periods of time, as MQTT is unable to do.

Increased Scalability: Both MQTT and Kafka are highly scalable, allowing organizations to handle increasing data volumes and device numbers. The possibility to deploy both services as clusters increases the overall scalability easily.

Increased Reliability: The integration of Kafka ensures high availability and fault tolerance mechanisms. If an application client has a network failure, it tracks the last received log and can continue the task even for large periods of downtime.

Pitfalls:

Complexity: Implementation is more complex than using a single technology. Careful planning and configuration are required to ensure optimal performance and reliability without creating bottlenecks.

Cost: Implementing and maintaining both MQTT and Kafka Clusters can involve an increase of costs, including hardware, software, and management.

3.3.3 *General Challenges and Solutions*

This section explores the most common challenges faced when designing and implementing a Unified Namespace.

3.3.3.1 Complex Data Models

A critical aspect of implementing a UNS is establishing a consistent data model hierarchy. This hierarchy defines how data is identified, accessed, and managed across the entire organization. However, this process can be complex and time-consuming, requiring collaboration across various departments and teams [27].

Also, it must be into account that Different systems may use incompatible data structures, making it difficult to represent and exchange data.

Another challenge shows when facing inconsistencies. Different systems and devices may use their own naming conventions, leading to confusion and integration difficulties.

As a possible solution, it is important to consider industry standards for data models as the ISA-95 standard for the industrial automation sector. These models provide pre-defined structures that can serve as a foundation for a unified naming scheme within the specific sector.

3.3.3.2 Inconsistent Data Structure

A common challenge is the incompatibility of data structures across different systems. This can lead to data inconsistencies, integration challenges, and difficulties in deriving meaningful insights from the data. For instance, systems may use varying data types

for the same data element, such as representing numbers as strings or date as integers, leading to inconsistencies in data analysis.

To address this issue, it's essential to implement dedicated data transformation services. These services act as intermediaries, converting data from incompatible formats into a standardized representation that can be seamlessly integrated into the UNS. By standardizing data structures, organizations can ensure data consistency, improve data quality, and facilitate data analysis.

Additionally, rigorous data model validation is crucial to maintain data integrity. Analyzing the structure of data messages before ingestion into the UNS helps identify and rectify inconsistencies or errors in the data.

3.3.3.3 Security

An UNS can present security challenges, as it becomes a central target for unauthorized access and potential data breaches. Protecting the UNS and the sensitive data it contains is crucial to maintain data integrity and prevent unauthorized access.

To ensure the security, it is essential to conduct a comprehensive security analysis of the entire architecture. This includes implementing mechanisms for authentication, authorization, and encryption to protect against unauthorized access and data breaches.

When working with wireless networks (WLAN) outside the internal business network, it is crucial to configure firewalls, intrusion detection and prevention systems (IDPS), and Distributed Denial of Service (DDoS) protection appropriately to mitigate risks.

Another critical security measure is to effectively manage and control the Software Bill of Materials (SBOM) of software components implemented in the UNS. Regularly updating the SBOM helps identify and address vulnerabilities breaches.

4 IMPLEMENTATION OF UNS FOR INDUSTRIAL IOT ARCHITECTURE

In this chapter it is described the process of design and implementation of an end-to-end IIoT solution as demonstration of an UNS architecture, following the presented requirements and goals in previous chapters.

4.1 OT Devices

This sub-chapter delves into the reasons behind employing simulated OT devices within the UNS architecture. It also outlines the methodology for data source used to populate these simulated devices.

4.1.1 *Simulated Devices*

A main design principle for this project involved simulating real-world devices, or at least real data patterns. This decision originated from two primary considerations. Firstly, the research prioritized investigating the internal structure of the UNS itself, particularly the data flow and conversion processes within an end-to-end environment. Employing real physical devices would have introduced complexities and dependencies beyond the focus of the research.

While generating data based on mathematical functions (e.g., linear increments, random binary outputs) is a common approach, the focus was to stablish an end-to-end implementation including the data visualization in dashboards. Therefore, a more realistic scenario was desired to show the UNS capabilities for potential clients and customers of it-novum. These simulated devices can be understood as real implementations, showing a more convincing demonstration.

Finally, real-world data from private companies is often subject to non-disclosure restrictions after system implementation. This industry practice created the necessity to acquire datasets from publicly accessible sources. This flexibility allows the system to be adapted to meet the specific needs of individual private clients.

For a detailed description of the simulated devices implementation, due to its close relationship with the OPC Server software, refer to the following Section 4.2.

4.1.2 *Data Source*

Kaggle has been the chosen platform for dataset acquisition. It is a prominent online platform specifically focused on data science and machine learning applications [36]. Kaggle offer an extensive repository with accessible datasets across a wide range of different domains, enabling researchers to train and validate AI models and algorithms.

While sharing PLC or hardware-specific datasets is not common, the primary focus of most datasets is on preprocessed or database-oriented data suitable for AI model training. Specific datasets with an industrial or automation systems background provided a sufficient large volumes of sensor data for simulating. Furthermore, datasets that include detailed documentation regarding data interpretation often withhold specific details about the underlying physical devices. This is particularly evident in datasets from industrial, commercial, or public sector environments (e.g., city-wide electricity consumption). This is likely due to privacy concerns or the sensitive nature of the underlying data.

For the testing phase, the dataset on Kaggle titled "BATADAL: Cyber Attacks Detection in Water Systems" [37] was chosen. The BATADAL dataset was specifically designed for the purpose of evaluating cyberattack detection algorithms within water distribution networks [38]. While it incorporates simulated data representing both normal operations and injected cyberattacks, for the present use this is not relevant.

The dataset is structured as hourly SCADA data, standard format for monitoring and controlling industrial processes. The dataset agroup variables such as tank levels, pump flow rates, valve positions, and pressure readings. The following Table 3 presents a set of variables, including their dataset name, a brief description, and some sample values.

Table 3. Example data from dataset.

L_T1 Water level of tank (m)	**F_PU1** Flowrate of pump (L/s)	**S_PU1** Status of pump (bit)	**P_J280** Pressure of junction (m)
0,73	98,93	1	2,97
0,69	97,95	1	2,97
0,9	96,82	1	2,98
1,11	96,76	1	2,98
1,27	94,77	1	2,99
1,54	94,58	1	2,99
1,85	94,94	1	3,00

It's important to note that the specific origin or focus of this dataset is of secondary importance. The primary interest lies in its suitability as a foundation for simulating data within the proposed system.

4.2 OPC UA Server

This sub-chapter explores the methods employed to simulate device data within the developed system. The ability to generate realistic sensor data is crucial in testing and validating the functionality of the system without relying on actual hardware. Two approaches were investigated for this purpose: utilizing a commercial OPC UA server software and developing a custom Python script. A brief introduction to OPC UA can be found in previous Section 3.1.2.

4.2.1 *KEPserverEX*

Initially, the generation of simulated data was intended to be accomplished using KEPserverEX, a proprietary OPC UA Server software. A brief introduction to KEPserverEX can be found in previous Section 3.2.3.

This approach was limited to create simulated devices that produced data based on predefined functions such as random number generation (Rand), linear progression (Ramp), or static arrays of string values as user inputs. The following Figure 20 shows the simulation parameters and results.

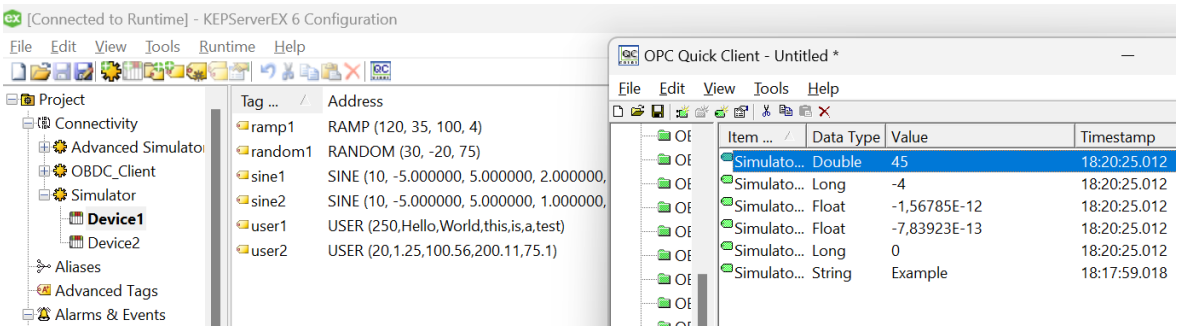


Figure 20. KEPserverEX simulated device.

To overcome these limitations, the software's "Advanced Simulator" component was explored as a potential solution. This feature offered the possibility of importing data from external sources like SQL tables or CSV files, thereby enabling a more realistic device simulation. This feature required certain configurations not included in the basic user guide and was detailed from the software's vendor's technical support. In the following Figure 21Figure 22 is shown the final configuration for simulated device in KEPserverEX.

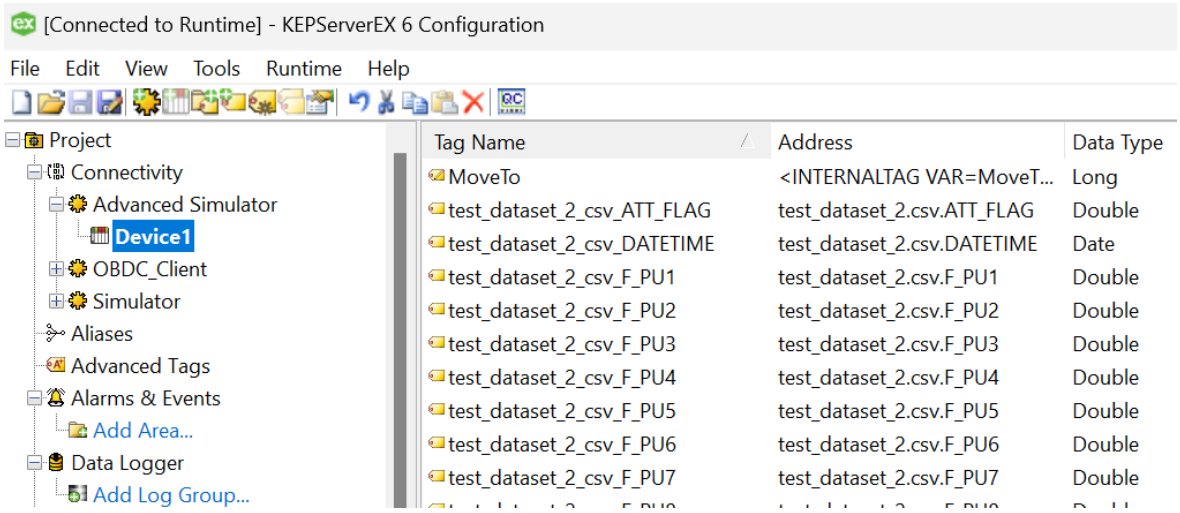


Figure 21. KEPserverEX advanced simulated device.

Unfortunately, this solution derived on unsatisfactory results. While the data could be imported from a CSV file as a table with variables at columns and value data at rows, the software exhibited issues in iterating through the data and was restricted to read-only operations, preventing future implementation of write functionalities.

4.2.2 *Python OPC UA Server*

Given the previous limitations, a Python script was developed to generate and manage simulated data. A brief introduction to Python can be found in previous Section 3.2.1.

The script deploys an asynchronous OPC UA Server, using an open-source library, and dynamically creates multiple simulated devices. Each device is created from a CSV file containing its data. The server continuously updates the corresponding OPC UA variables with the data from these CSV files, effectively simulating real-time sensor data.

As outlined in Section 4.1.1, a key objective of the system is to accommodate diverse data sources and configurations. The employed CSV-based structure, featuring a header row for variable names and subsequent rows for data values, facilitates this adaptability. This approach enables dynamic modifications to the system to align with specific client requirements and data representations.

In the following Figure 22 the script is represented in a diagram flowchart. Note: the flowchart omits error handling for brevity.

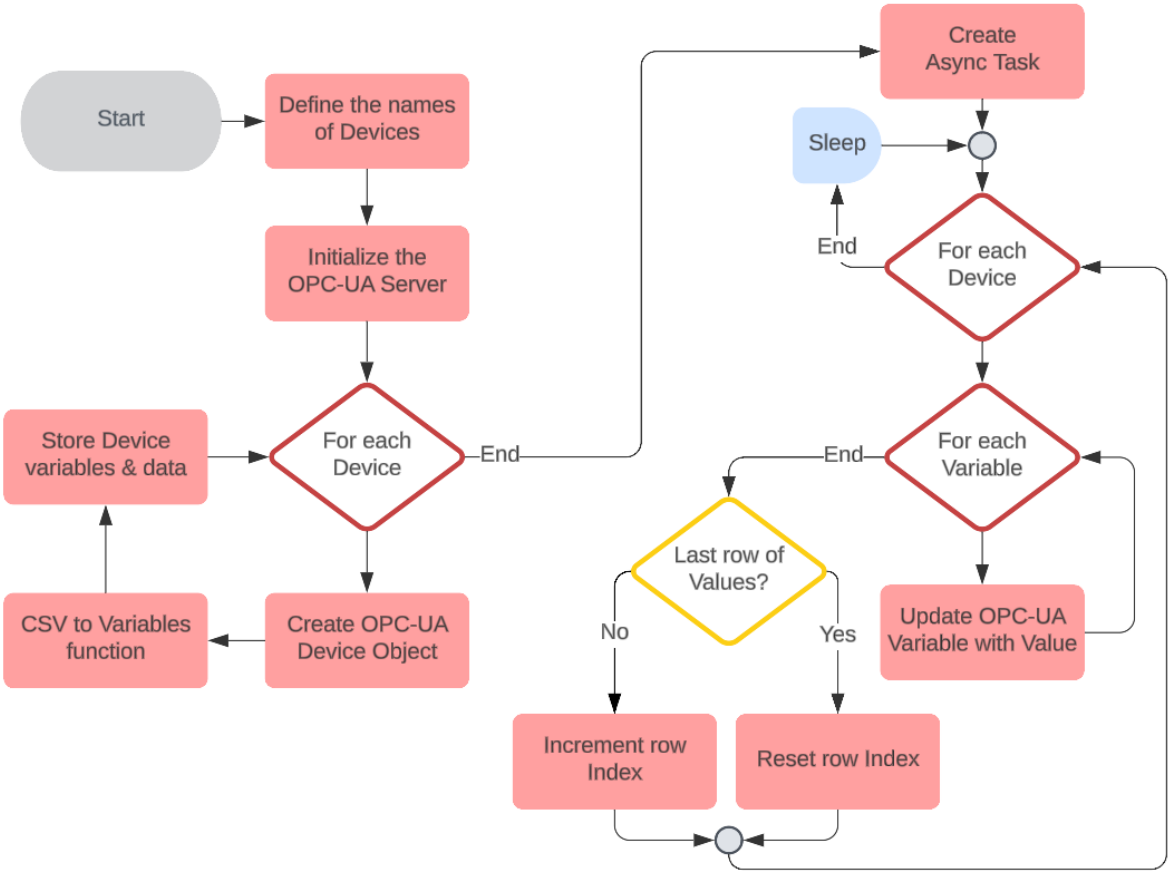


Figure 22. Python OPC UA Server script flowchart.

The OPC UA Server has been developed using the opcua-asyncio library [39]. This library's asynchronous architecture facilitates efficient code execution and potentially enhances performance compared to traditional synchronous approaches.

As shown in the top-left corner of the previous Figure 22, the script initiate by defining a list of device names associated with corresponding CSV files (e.g., "dataset_pressure.csv"). Each CSV file must be structured with the header row providing variable names as string values. The subsequent rows encapsulate the actual data values for each variable. The script is designed to accept any number of CSV files, upon available system memory.

The main function establishes the OPC UA server instance and configures its endpoint and namespace. It also creates a container object named "SimulatedDevices" within the namespace to serve as a hierarchical parent for all subsequently generated OPC UA objects representing the simulated devices.

Next, the program iterates through the list of device names. For each device name, a new device object is created under the "SimulatedDevices" object.

Then, a defined function, csv_to_variables, is called. It takes a CSV file path, an OPC UA object, and the OPC UA namespace id as input. It processes the file, extracts column names as variable names, and creates corresponding OPC UA variables within the specified namespace. Importantly, it allows clients to write to these variables by setting the writable attribute. Finally, the function returns a list of the created OPC UA variables ids and the data rows in the file. Additionally, a list is created to keep track of the current data row index for each device.

The server enters a infinite asynchronous loop (refer to previous Figure 22, top-right). Within this loop, each simulated device is iterated. For each device, the corresponding variables are updated sequentially with the next data point. This process involves retrieving the data value based on a tracked index and writing it to the respective OPC UA variable. Once all variables for a device have been updated, the index is incremented or reset to the beginning of the dataset as necessary. Finally, to mimic real-time behavior, a brief delay is introduced after each device variables have been updated and the infinite loop restart.

The full Python script code for this OPC UA server implementation is provided in Appendix 8.1.

4.3 OPC Router

The last component in the OT area must be a gateway able to convert from OT protocols to IT protocols. For this purpose, the software OPC Router has been chosen as one of the possible OT gateways. A brief introduction to OPC Router can be found in previous Section 3.2.4.

The OPC UA client serves as a crucial intermediary within the system, facilitating data exchange between the OPC UA Server and external networks. The client can interact with the OPC UA Server through two primary mechanisms, Request-Response, or Subscriptions to receive data updates from the server, leveraging the Report by Exception mechanism to optimize data transfer.

In this sub-chapter it is explained how the OPC Router software has configured to act as the gateway from OPC UA to MQTT, after applying the Sparkplug B format to the MQTT messages.

4.3.1 *Plugins configuration*

To establish a connection between the OPC Router and an OPC UA server, navigate to the Plugins section within the OPC Router interface and select the "OPC UA Client" plugin.

1. Create a New Connection by specifying the following parameters:
    - Name: Assign a unique identifier for the plugin connection.
    - Endpoint: Input the public address of the OPC UA server.
    - Access Path: Define the root node within the OPC UA server's address space from which data will be retrieved. This limits the scope of data accessible through this connection.
2. Configure Security (Optional)**:** If the OPC UA server requires authentication, select the appropriate security method as user login or certificate and provide the necessary credentials.
3. Configure Redundancy (Optional): If a backup or cluster server is available, specify its endpoint in the redundancy settings. This ensures uninterrupted data access in case of connection failures.

In the following Figure 23, it is shown the basic OPC UA client configuration within the OPC Router. In this example, a simple connection without security or redundancy has been established.

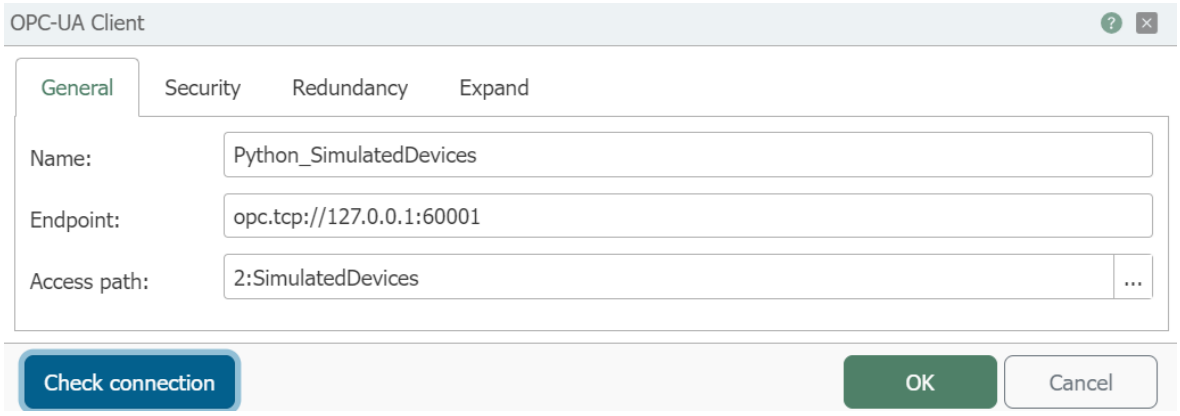


Figure 23. OPC Router configuration for OPC UA Client.

To stablish the connection with the MQTT broker, the "Sparkplug Connections" plugin in OPC Router software is used. This plugin allows for the configuration of MQTT connections while simultaneously enabling the use of the Sparkplug B messaging format.

1. Configure Sparkplug settings:
    - Name: Assign a unique identifier for the plugin connection.

- Variant: Select "Sparkplug B" as the message format (Sparkplug A is deprecated).
- Profile: Define whether the connection represents a "Node" (device) or an "Application" (command sender).
- Group ID and Node ID: Optionally provide a Group ID and Node ID for identification within the MQTT broker to be used as Default.
- Keep Alive: Specify the interval for sending keep-alive messages to maintain the connection.

2. Configure MQTT Connection settings**:**
   - Broker: Input the IP address or hostname of the MQTT broker.
   - Port: Specify the MQTT port number (typically 1883 or 8883).
   - Client ID: Provide a unique identifier as an MQTT client.
   - Security: If required, configure security settings such as SSL/TLS protocol, username, password, or client certificates.

The following Figure 24 illustrates the basic MQTT connection configuration used for the testing environments.

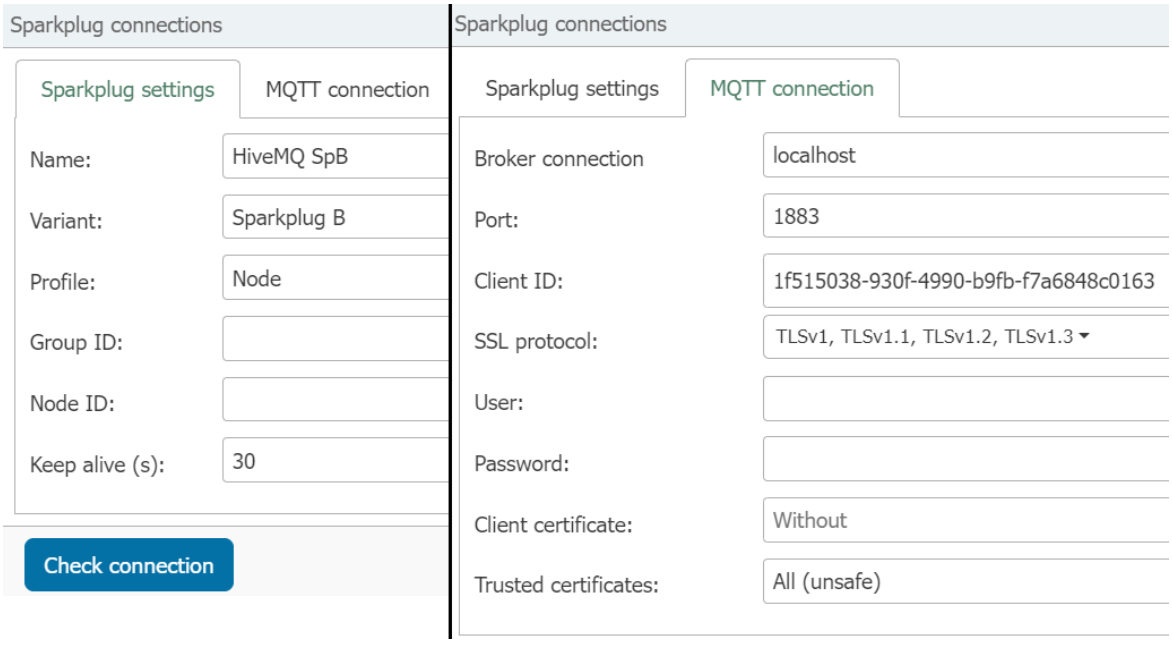


Figure 24. OPC Router configuration for Sparkplug Plugin.

4.3.2 *Uplink Connections*

To efficiently manage the mapping of multiple OPC UA variables to corresponding Sparkplug B messages, the OPC Router offers a template-based approach. This allows for the creation of a connection structure that can be replicated with internal defined variables.

The template created to define the core components of an OPC UA to Sparkplug B connection is detailed. This template includes the following two elements:

1. **OPC UA Trigger Data Change:** This element defines the conditions for triggering the connection. It requires the following parameters:
    a. Access Data: Specifies the plugin responsible for connecting to the OPC UA server.
    b. Trigger Item: Identifies the specific OPC UA object to monitor for data changes.
2. **Sparkplug Node Publish:** This element handles the creation and transmission of all the necessary Sparkplug B messages, not only Data ones but also Birth or Death:
    a. Connection: Specifies the established Sparkplug plugin.
    b. Group, Node, and Device ID: Specifies the desired Sparkplug B topic hierarchy.

To apply the template and create multiple connections, internal variables can be utilized to define the specific OPC UA variables and corresponding Sparkplug B parameters. By populating a template with these variables, a large number of connections can be generated at once.

The following Figure 25 shows a Template configuration interface within the OPC Router. Using the template, it is defined a generic connection with variables, shown as "{variable_name}", for the OPC UA address and other relevant parameters. Then, by creating instances of the template and specifying different OPC UA addresses for each instance, it can be quickly established connections to monitor multiple nodes. This is the Template used in demo architecture.

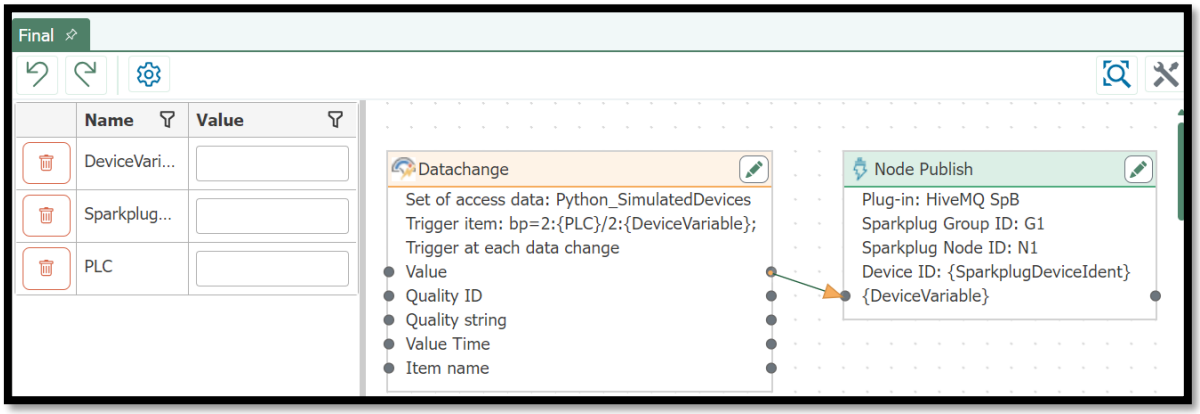


Figure 25. OPC Router implemented Template.

Each connection can be manually configured with the input fields showns on the left side on previous Figure 25. An alternative is to create an Excel file with columns representing each internal variable and their different values, giving the option to create all the desired connections simultaneously.

The following Table 4 shows a fraction of the Excel table used to populate the OPC Router Template with specific internal variable information. Header represents each internal Template variable name. Each row in the Excel template represents a new connection.

Table 4. Excel file to create multiple OPC Router Template instances.

InstanceName	DeviceVariable	SparkplugDeviceIdent	PLC
F_PU1	F_PU1	PU1	PLC_1
S_PU1	S_PU1	PU1	PLC_1
P_J280	P_J280	PU1	PLC_1
F_PU2	F_PU2	PU2	PLC_1
S_PU2	S_PU2	PU2	PLC_1
P_J269	P_J269	PU2	PLC_1
F_PU3	F_PU3	PU3	PLC_1
S_PU3	S_PU3	PU3	PLC_1
P_J300	P_J300	PU3	PLC_1

On the left side of the following Figure 26 it is shown a set of generated connections based on the previous template. On the right side it is shown the working status of a connection, their current values and each timestamp that the connection has been triggered.

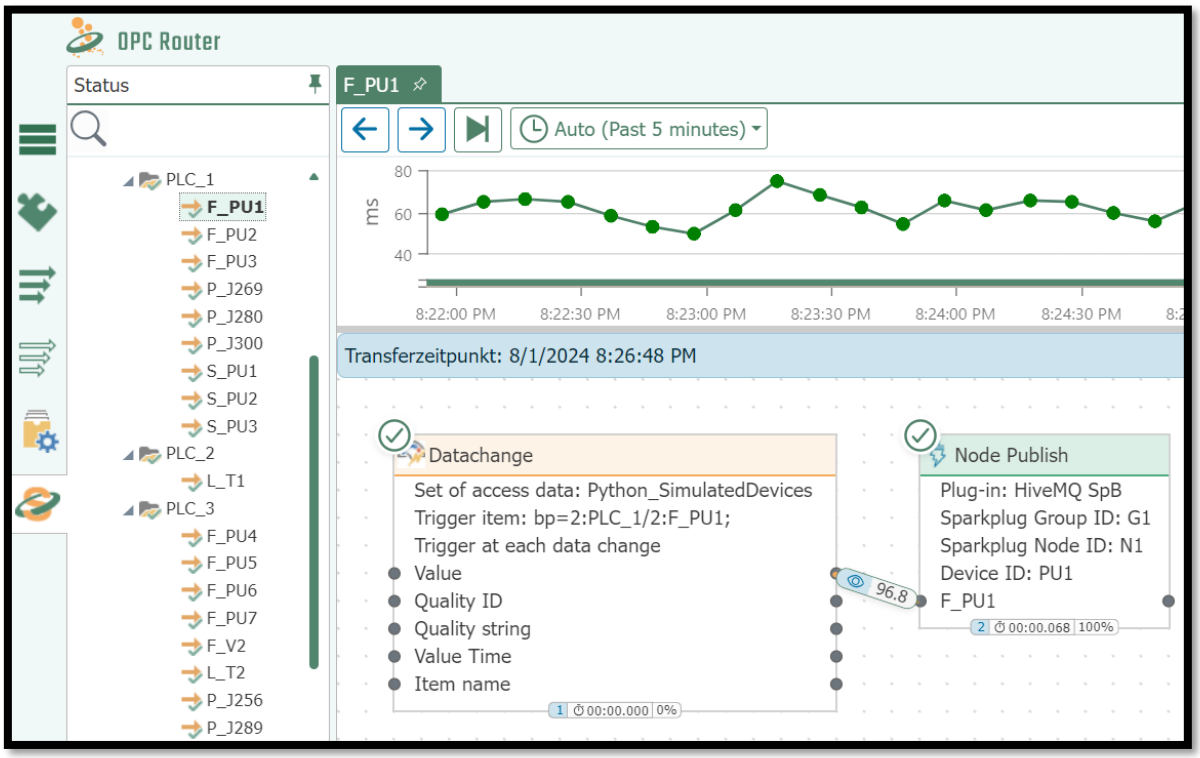


Figure 26.OPC Router Connection triggered.

4.3.3 *Downlink Connection*

To manage the mapping of MQTT Sparkplug messages to OPC UA variables, the OPC Router incorporates a Sparkplug Trigger named "Command Received." This trigger is designed to activate a connection when a specific command is received within a defined topic hierarchy.

If the Device ID component of the topic is omitted, the trigger subscribes to the NCMD (Node Command) topic for the specified Group ID and Node ID. Otherwise, if the Device ID is included, the subscription is done to the corresponding DCMD (Device Command) topic hierarchy. The following Figure 27 illustrates a Connection configured to receive a command for a specific Device ID and subsequently write the message value to an OPC UA variable.

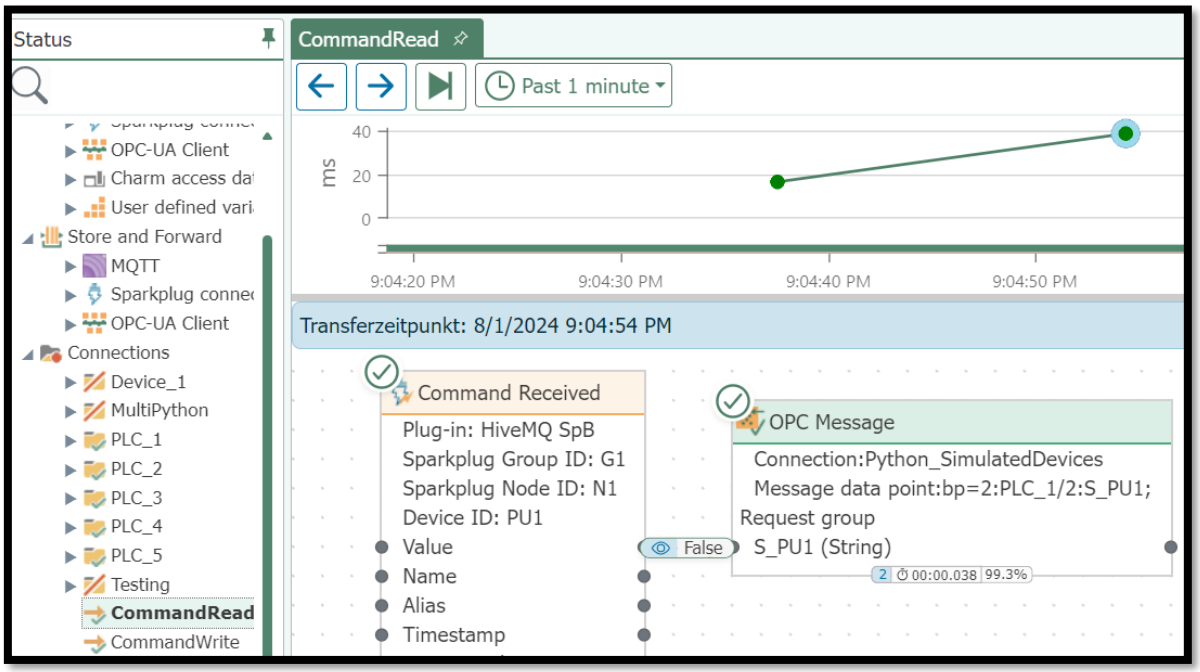


Figure 27. OPC Router Connection receiving Sparkplug command.

It is important to highlight that the Command Received Sparkplug Trigger is currently under development and may exhibit unexpected behavior in production environments. Instances of the trigger failing without apparent reason have been observed. As a temporary solution provided by the official OPC Router technical support, it is required to manually trigger one node called “Sparkplug command” using a separate connection. This connection triggered activates the internal Sparkplug plugin which failed previously.

The following Figure 28 provides an example of this workaround, demonstrating how to manually send a Sparkplug B command to activate the trigger.

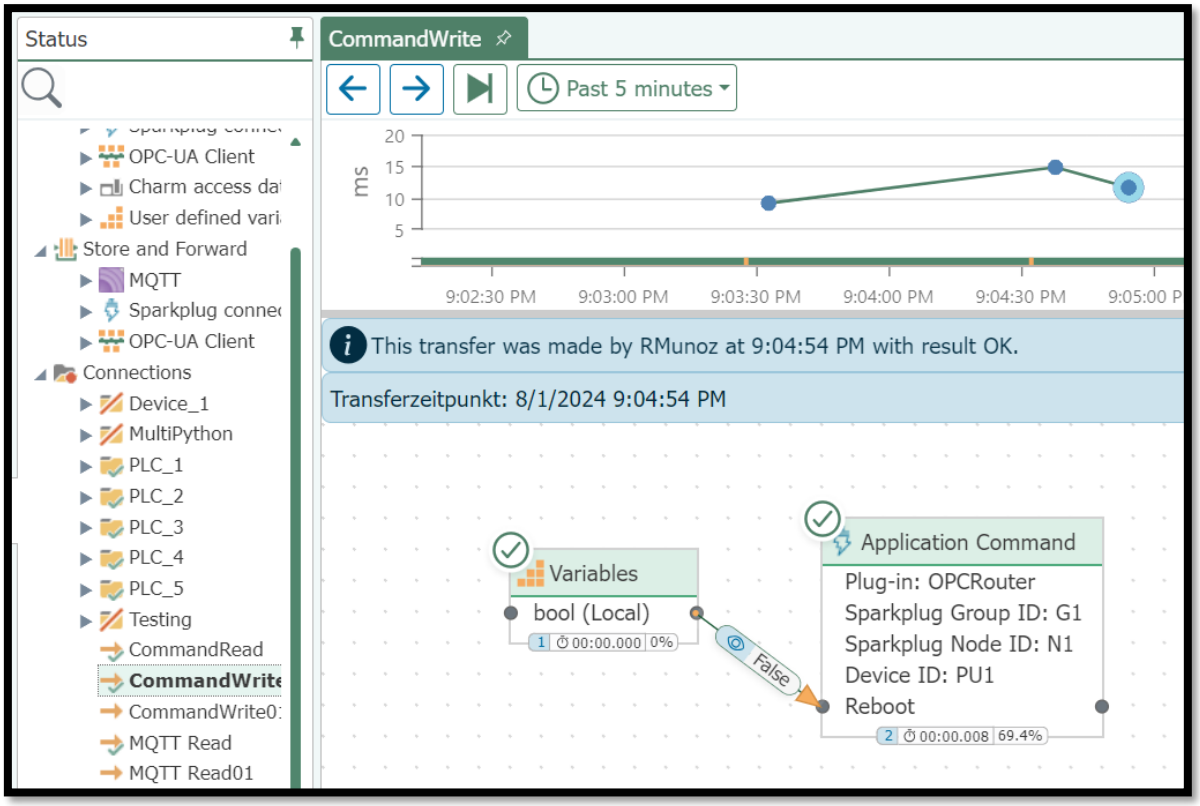


Figure 28. OPC Router Connection to manually trigger Sparkplug.

4.4 HiveMQ Broker

The Message Broker serves as the core component of the UNS architecture. More precisely, an MQTT broker is usually selected for this critical role. While the MQTT protocol itself is a light and easy to implement protocol, the quality of its implementation can vary significantly across different software solutions. Many MQTT brokers and clients offer lightweight solutions that may compromise on QoL features or performance.

By using an integrated MQTT broker in another system or service, the quality of performance would decrease in exhaustives environments or high demand timeslots. In the other hand, using a dedicated MQTT broker, as HiveMQ provides, the full potential of the MQTT protocol can be achieved. Moreover, HiveMQ's open-source nature enhances its appeal for this application. A brief introduction to MQTT protocol and HiveMQ can be found in previous Section 3.1.3 and Section 3.2.5 respectively.

4.4.1 *Deploying Broker*

To simplify the deployment and management of the HiveMQ broker, a Docker container was deployed. This approach offers several advantages, including enhanced portability, isolation, and reproducibility. By encapsulating the HiveMQ instance within a

Docker container, potential conflicts with other services, such as those using the default MQTT ports 1883 or 8883, have been effectively mitigated.

The initial step to deploy the Docker instance is to acquire the official HiveMQ Docker image from the repository Docker Hub and then running the container. This can be achieved using the following commands:

```
> docker pull hivemq/hivemq:latest
> docker run -d -p 8080:8080 -p 11883:1883 hivemq/hivemq4:latest
```

The last command creates a HiveMQ instance, mapping port 8080 of the container to port 8080 of the host for the HiveMQ Control Center, and port 1883 to port 11883 for the MQTT broker. The -d flag ensures the container runs in detached mode.

Once the container is running, the HiveMQ Control Center can be accessed through a web browser at http://localhost:8080. In the following Figure 29 it is shown the interface through which the broker can be supervised.

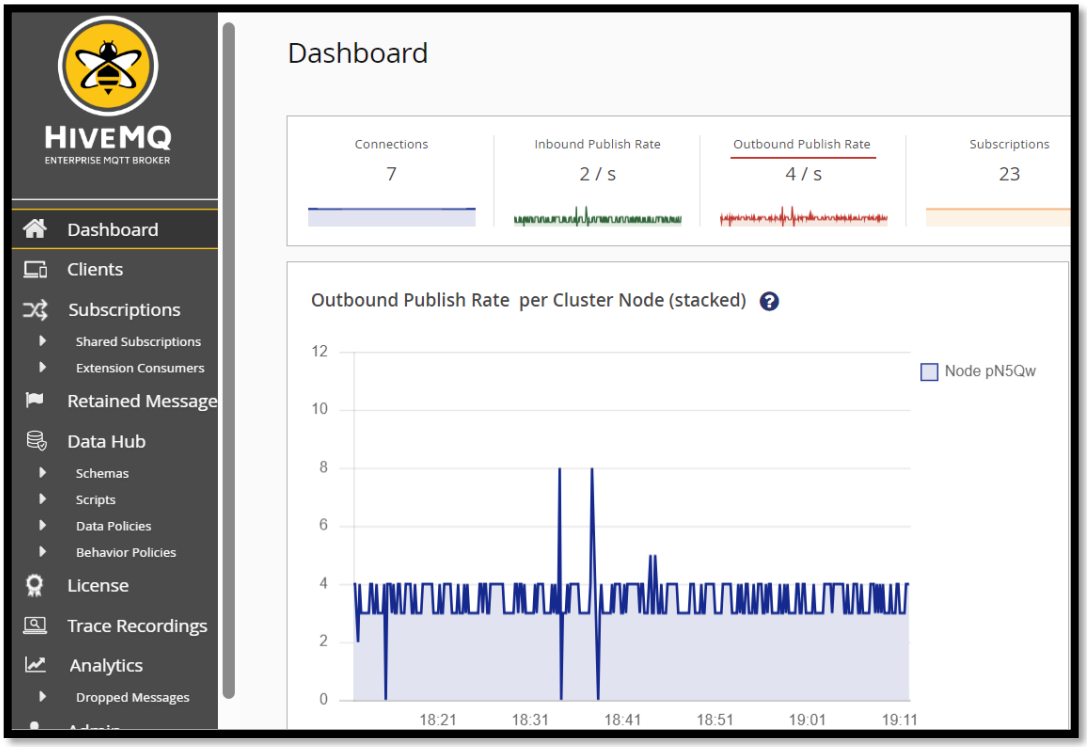


Figure 29. HiveMQ Control Center interface.

The HiveMQ Control Center serves as a centralized management console, offering real-time monitoring and configuration capabilities. Its primary function is to track metrics such as client connections, message rates through dashboards. This feature is particularly

valuable when operating with broker clusters, enabling centralized oversight and management of multiple instances in different locations.

No other configuration for basic functionality is required, the broker is ready to receive Publishers and Subscribers petitions. Nevertheless, HiveMQ offers more advanced features as configure the retencion policies for messages queues or Inbound / outbound policies as an firewall measurement, but neither of this advanced features have been prioritized in the realization of this thesis.

Using a traffic analyzer within the docker network, it can be seen how the messages are received as MQTT with Sparkplug B format. In the following Figure 30 it is shown how a MQTT Sparkplug published message is received in the broker, and the next message is sent by the broker, meaning that there is a subscription active for the received message.

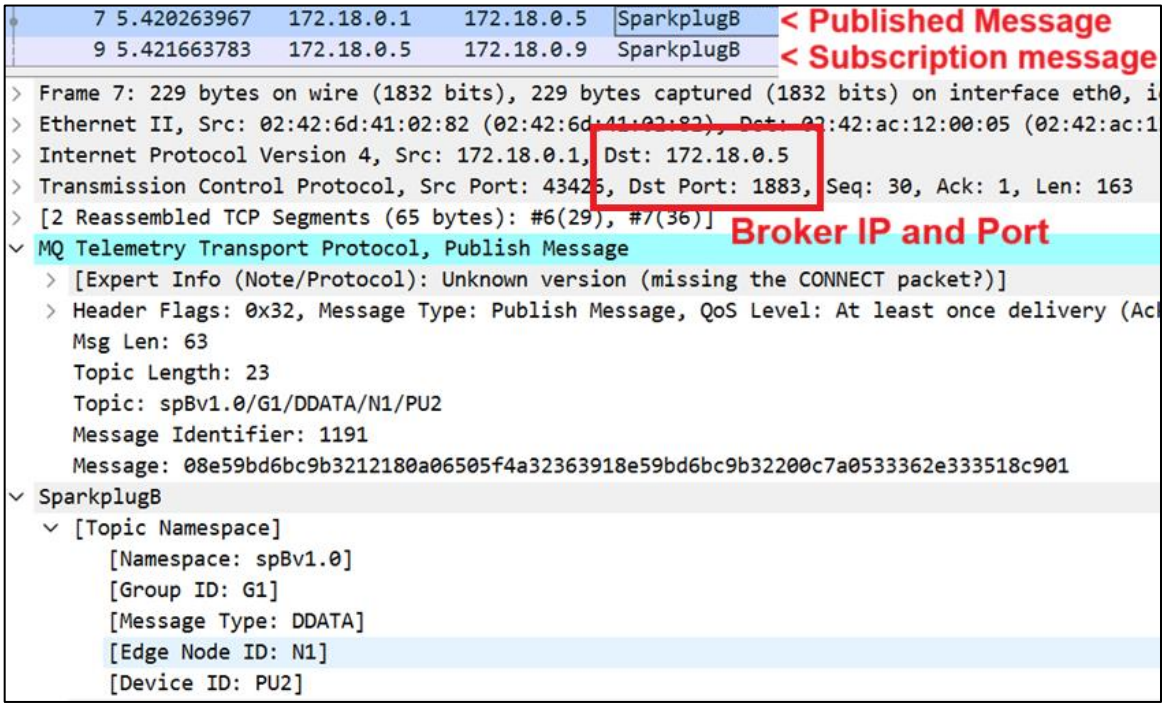


Figure 30. MQTT Sparkplug traffic.

4.5 Confluent Platform

To address the limitations of the MQTT Broker and create a more efficient architecture, as discussed in Section 3.3.2.3, the implementation of an event-driven platform is crucial. Instead of using Apache Kafka directly, Confluent, a specialized Kafka distribution for industrial applications, is chosen. For a quick overview of Apache Kafka and Confluent, please refer to Section 3.1.5 and Section 3.2.6, respectively.

Confluent offers both a managed cloud service and a self-hosted platform. While the cloud service has a free trial, the self-hosted version is selected to maintain control over the

deployment. To simplify the process and ensure reproducibility, Docker Compose is used to manage Confluent Platform service containers.

4.5.1 *Docker deployment.*

Confluent Platform is composed of multiple services. To streamline deployment and configuration, a Docker Compose file is used. This file defines the services and their dependencies, allowing the entire platform to be started and stopped with a single command.

A suitable Docker Compose file is found in the Confluent GitHub repository [40]. Specifically, the "cp-all-in-one" configuration is used. However, the Kafka Connect service needs to be modified to include a command that downloads and installs the MQTT connector. This is necessary for creating MQTT Source and Sink connectors later described in Section 4.5.3. The final configuration file can be found in Appendix 8.4 for more details.

The final Docker Compose file defines and deploys the following set of Confluent Platform services. The description and diagram of port mapping and connections can be found in Section **Error! Reference source not found.**.

1. **zookeeper**: Apache ZooKeeper, a distributed coordination service for distributed applications. Provides high availability and ensures consistency across the cluster.
2. **broker**: Apache Kafka, a distributed streaming platform for handling real-time data feeds. Processes high-throughput streams of messages published by producers and consumed by consumers.
3. **schema-registry**: Confluent Schema Registry, a central repository for storing and evolving Kafka message schemas.
4. **connect**: Confluent Connect, a framework for integrating Kafka with various data sources and sinks. Enables data ingestion and export from diverse sources.
5. **control-center**: Confluent Control Center, a web-based UI for managing and monitoring the Confluent Platform.
6. **ksqldb-server:** Confluent KSQL Server, a stream processing engine for running continuous queries on Kafka streams.
7. **ksqldb-cli:** Confluent KSQL CLI, a command-line tool for interacting with KSQL Server.
8. **rest-proxy:** Confluent Kafka REST Proxy, a REST API for interacting with Kafka topics.

To deploy the services, the following command must be executed from a command-line console located in the directory of the Docker Compose file:

```
> docker compose up -d
```

After the command is executed and all services are initialized, the Confluent Control Center can be accessed in any web browser using the designated external port. In this configuration file, the port is 9021. The following Figure 31 shows the main menu interface.

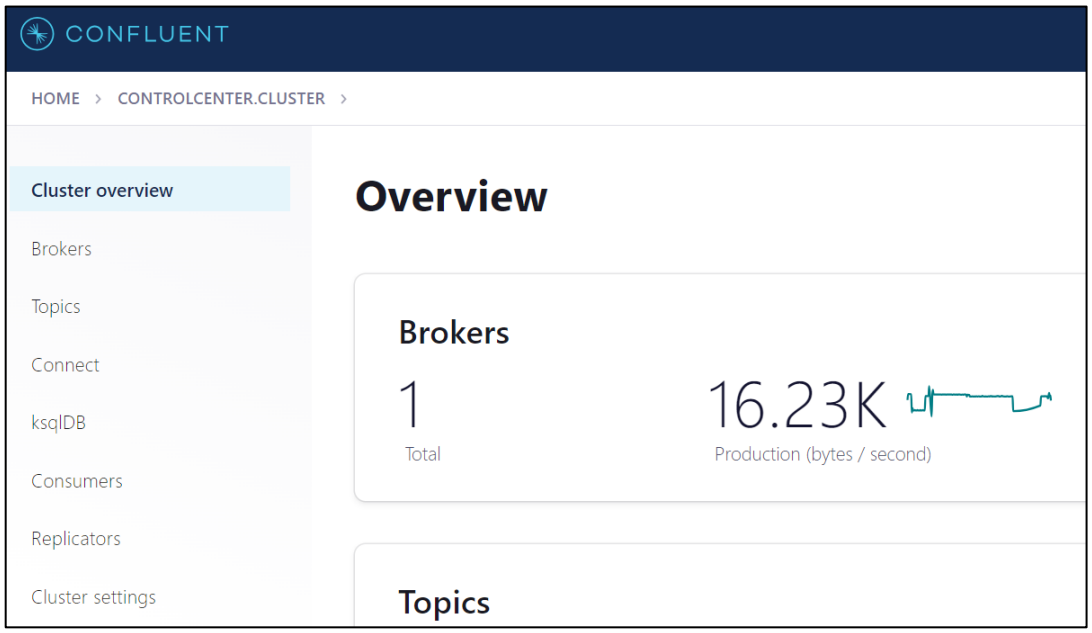


Figure 31. Confluent Control Center interface.

4.5.2 *Topic creation*

The first step is to create the designated topics where data will be inserted and obtained. The following steps explain how to create a topic:

1. Click on the "Topics" section in the lateral menu. This will display a table listing each topic, showing statistics such as current status, partition numbers, data produced and consumed, consumer and producer count, and the last time data was inserted. Even in a new Confluent Platform deployment, there will already be topics, producers, and consumers created by Confluent's internal functions.
2. Click the "Add Topic" button in the upper right corner of the screen.
3. Enter the topic name and the desired number of partitions.
4. Customize settings (optional):
    a. **Availability:** Select the desired number of replicas for the topic across other brokers in the cluster.
    b. **Storage Policy:** Configure retention time and size limits.

c. **Maximum Message Size:** Set a maximum message size (in bytes). Messages exceeding this size will be discarded.

When a topic is clicked in the table, a detailed view with four different windows will appear. The Overview window shows dashboards for Production and Consumption (in Bytes per second), as well as other parameters like availability and partitions. The Messages window displays a list of newly inserted messages into the topic. Additionally, messages can be searched by their offset and partition number.

The following Table 5 lists the Kafka topics created in this implementation, along with a brief description of their usage.

Table 5. Kafka Topics created in this implementation.

Kafka Topic Name	Description
mqtt_data	Stores all the devices data messages received from the MQTT broker in Sparkplug B protobuf format.
json_data	Stores all the devices data messages parsed from protobuf to JSON format.
json_command	Stores the commands messages received from IT platform in JSON format.
mqtt_command	Stores the command messages parsed from JSON to Sparkplug B protobuf format and will be sent to MQTT Broker.

4.5.3 *Kafka Connect configuration*

After creating the Kafka topic, the next step is to produce data into it. In this scenario, this will be achieved by ingesting messages from the MQTT Broker. Since the data flow is from Kafka to an external system, Kafka Connectors will be used. These connectors subscribe and publish to the MQTT Broker.

4.5.3.1 MQTT Source connector

The MQTT Source connector retrieves messages from the MQTT Broker (subscribing to a designated topic) and inserts them into the configured Kafka topic. Follow the next steps:

1. Access the Connect interface in the lateral menu of the Confluent Control Center.
2. Click "Add new Connector."

3. Select "MqttSourceConnector." If this option is missing, manual installation of the connector plugin is required (already covered in the Docker Compose file).
4. Enter the Server URIs. In this implementation, the HiveMQ container is located at "tcp://hivemq:1883".
5. Enter the Kafka Topic name. In this implementation, it is "mqtt_data".
6. Enter the MQTT Topics to subscribe. Wildcards are accepted ("+" for single-level and "#" for multi-level). Multiple MQTT topics can be subscribed to. For this implementation, the following topics are subscribed:
    a. spBv1.0/+/DDATA/#
    b. spBv1.0/+/NDATA/#

    The selection of this two MQTT topics means that all the Data messages, from devices and Edge Nodes, will be inserted into the designated Kafka topic

In the following Figure 32, it is shown the overview of all the parameters configured for the MQTT Source connector as previously described.

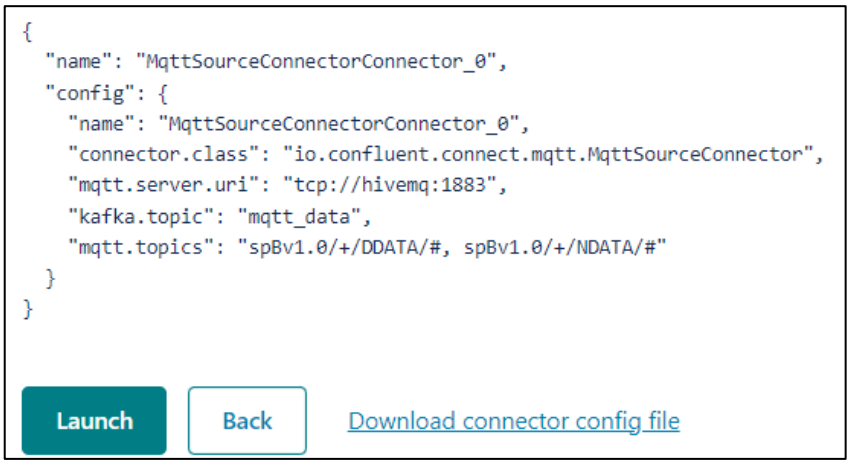


Figure 32. Kafka MQTT Source connector configuration.

After configuring the connector, all incoming messages to the selected MQTT topics will be automatically ingested into the designated Kafka topic. The following Figure 33 shows the reception of messages from the MQTT Broker. Note that the message payload is unreadable as it a raw byte representation of Sparkplug B Protobuf payload format originated from OPC Router Protobuf.

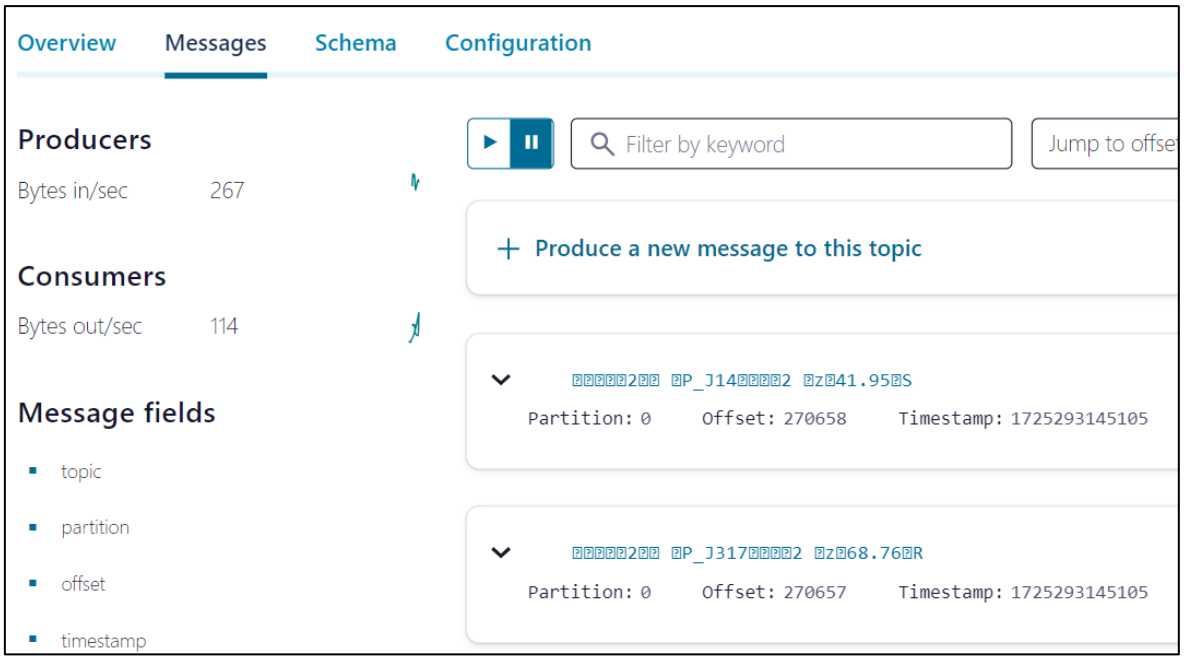


Figure 33. Kafka Topic storing Sparkplug B protobuf data.

4.5.3.2 MQTT Sink connector

The MQTT Sink connector sends messages from the Kafka topic to the configured MQTT topic (publishing to the designated MQTT topic).

A challenge faced was to define the MQTT topic to which is desired to publish the data. The MQTT Sink Connector automatically uses the Kafka topic name as the MQTT topic value. However, Kafka topics cannot include the separator "/", which is used to define a hierarchy in MQTT topics. To address this, a Regex transformation is used in the MQTT Sink Connector configuration.

Regex, or Regular Expression, is a pattern that defines specific sequences of characters within a string. By applying a regex transformation, parts of a text string matching a particular pattern can be extracted, replaced, or modified.

To configure the Sink Kafka Connector, follow the next steps:

1. Access the Connect interface in the lateral menu of the Confluent Control Center.
2. Click "Add new Connector."
3. Select the desired Kafka topic to get messages from. In this implementation, it is "mqtt_command."
4. In Value converter class, introduce the following: “org.apache.kafka.connect.converters.ByteArrayConverter”

This specifies the desired format for the message value. By default, Confluent uses Avro format. To send the message in Protobuf, the ByteArray class is used.

5. In Transforms, introduce any value, for example "Replacement."
6. A new section, “Transforms: Replacement”, will appear:
    a. Transform Type: “org.apache.kafka.connect.transform.RegexRouter”
    b. Regex: Include the Kafka topic between parentheses. In this implementation, "(mqtt_command)".
    c. Replacement: Include the full MQTT topic for publishing. In this implementation, "spBv1.0/group_id_1/NCMD/node_id_1".

    This Regex transformation matches the exact name of the Kafka topic and replaces it with the desired MQTT topic for publishing.
7. Enter the Server URIs. Following this implementation, the HiveMQ container is located at "tcp://hivemq:1883."

In the following Figure 34 is shown the overview of all the parameters configured for the MQTT Sink connector as previously described.

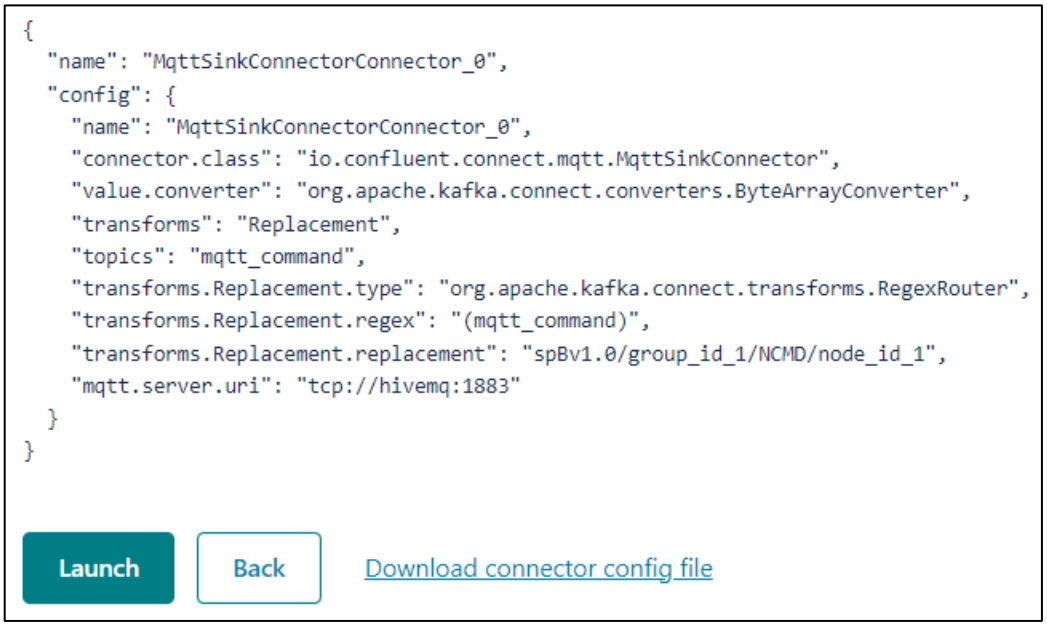


Figure 34. Kafka MQTT Sink connector configuration.

Once the connector is running, all incoming messages to the selected Kafka topic will be automatically published to the designated MQTT topic.

4.5.4 *Schema creation*

To establish the data format in the topic, a schema must be created. This ensures that each message inserted into the topic is validated against the established schema and discarded if it does not comply.

A challenge encountered was Confluent's support for three schema formats: JSON, Avro, and Protobuf. The messages are received in Sparkplug B format, which is encoded in Protobuf. Cirrus Link Solutions provide the Protobuf schema defined in a .proto file [41]. However, Confluent does not support complex schemas with redundant definitions, which is the case for the Sparkplug Protobuf schema

As a solution, Sparkplug messages are stored as raw bytes protobuf format. A set of microservices running Python scripts are used to transform Sparkplug raw bytes into Sparkplug JSON format and viceversa.

4.5.4.1 Protobuf to JSON microservice

This microservice is a Python script that translates Sparkplug B Protobuf data from one Kafka topic to JSON data in another topic. The full code of the Python Script can be found in the Appendix 8.2. Here is presented a brief description of its workflow.

First, It sets up Kafka configuration parameters, including the IP and port of the Kafka broker, consumer and producer group IDs, and topic names. Then, It creates a Consumer client object to subscribe to the topic containing Sparkplug B data and a Producer client object to publish messages to the topic that will receive the JSON data.

The script enters an infinite loop, continuously polling the consumer topic for incoming messages. If a message is received, it processes the message by converting the Sparkplug B message into a JSON string. As addition, The MQTT topic is inserted into the JSON message as a new key-value for later integration with application that requires the original Sparkplug B topic.

The processed data is validated. If valid, the processed JSON data is produced to the JSON topic. After manual termination, a summary of errors encountered during the script's execution is printed to the console log to check for any parsing errors..

In the following Figure 35 it is shown the results of the script, a new Kafka topic is republished with the messages in JSON format.

Figure 35. Data transformation from Protobuf (left) to JSON (right).

4.5.4.2 JSON to Protobuf microservice

This microservice is a Python script that translates JSON data from one Kafka topic into Sparkplug B messages for transmission on another topic. The full code of the Python Script can be found in the Appendix 8.3. Here is presented a brief description of its workflow.

First, it sets up Kafka configuration parameters, including the IP and port of the Kafka broker, consumer and producer group IDs, and topic names. Then, it creates a Consumer client object to subscribe to the topic containing JSON data and a Producer client object to publish messages to the topic that will receive the Sparkplug B data. The script enters an infinite loop, continuously polling the consumer topic for incoming messages

If a message is received, it processes the message by using the parse_json_to_proto function from the "google.protobuf.json_format" library to convert the JSON data into a Payload message object defined by the Sparkplug B protobuf schema. If the conversion is successful, the message object is serialized into a byte string. The serialized byte string representing the Sparkplug B message is produced to the configured producer topic.

After manual termination, a summary of errors encountered during the script's execution is printed to the console log to check for any parsing errors.

In the following Figure 36, it is shown the results of the script, a new Kafka topic is republished with the messages in JSON format.

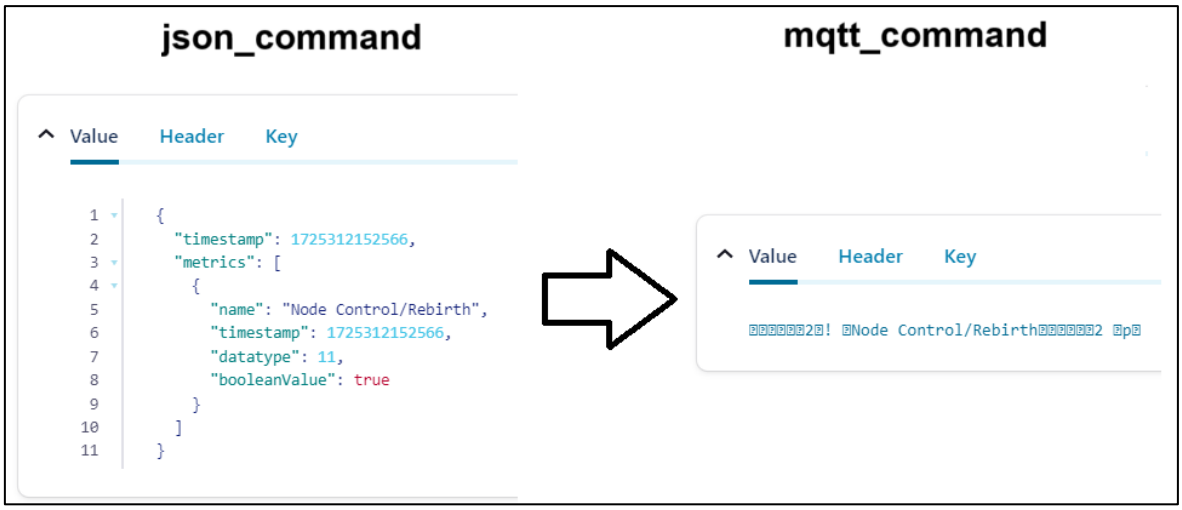


Figure 36. Data transformation from JSON (left) to Protobuf (right).

4.6 ThingsBoard

As the final step in the architecture, the event-driven platform should provide multiple data streams to the IT side of the business. For this purpose, ThingsBoard was chosen. A brief introduction to ThingsBoard can be found in Section 3.2.7.

ThingsBoard act as an IoT platform to monitor devices through dashboards. It can also interact with devices by sending commands in the uplink direction of the communication, for example, to reset a device. While ThingsBoard has built-in features to store data received from devices (historian functionality), this feature was not explored in this implementation.

4.6.1 *Docker deployment*

ThingsBoard consists of two services, which can be deployed separately. To simplify deployment and configuration, a Docker Compose file was used to deploy both container services simultaneously.

The official ThingsBoard documentation provides guides for different installation methods [42], including standalone ThingsBoard PE installation using Docker in Windows.

ThingsBoard can use various messaging systems for storing messages and communication between ThingsBoard services, such as Kafka, Azure Service Bus, or Google Pub/Sub. However, for this demonstration, the In-Memory queue (a built-in system) was used to keep the implementation as simple as possible.

The final Docker Compose file defines and deploys the following set of ThingsBoard services. For additional details, the complete configuration file can be found in later

Appendix 8.5. The description and diagram of port mapping and connections can be found in Section **Error! Reference source not found.**.

1. **mytbpe**: This service runs the ThingsBoard Platform Enterprise application. The purpose is to manage devices, data, and applications. Following ports are exposed:
    i. 9090:8080: Maps host port 9090 to container port 8080, allowing access to the TB-PE web interface on the host machine at port 9090.
    ii. 8883:1883: Maps host port 8883 to container port 1883 for MQTT communication. Commonly used by IoT devices, but unused in this implementation.
    iii. 7070:7070: Exposes port 7070 for WebSocket connections.
    iv. 5683-5688:5683-5688/udp: Exposes a range of UDP ports for specific communication protocols within ThingsBoard.
2. **postgres**: This service runs a PostgreSQL database, which is used by TB-PE to store device data, user information, and other critical data. Exposes port 5432 on the host machine, allowing connections to the PostgreSQL database.

To deploy the services, run the following command in a terminal located in the directory of the Docker Compose file:

```
> docker compose up -d
```

After the services have initialized, access the ThingsBoard web interface in any web browser at the designated external port (9090 in this configuration). ThingsBoard PE offers different default users with varying privileges:

1. **System Administrator**
    - Highest level of access. Manages the entire ThingsBoard platform. Adds new organizations, configuring system resources, managing user accounts.
    - User / Password: sysadmin@thingsboard.org / sysadmin
2. **Tenant Administrator**
    - Manages a specific tenant (organization). Configures dashboards, creates rules, assigns user roles.
    - User / Password: tenant@thingsboard.org / tenant
3. **Customer User**

- Lowest level, assigned by tenant admin. Monitores device data, performes basic device actions.
- User / Password: customer@thingsboard.org / customer

For the following steps, access ThingsBoard with Tenant privileges. The following Figure 37 shows the main menu after logging in with Tenant credentials.

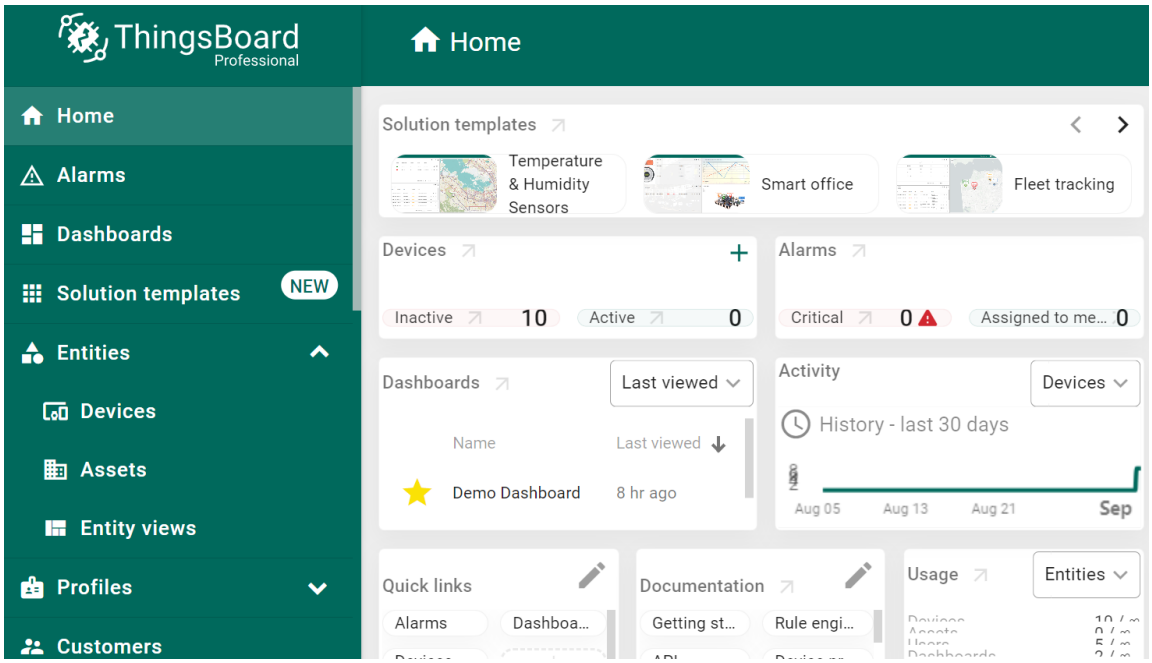


Figure 37. ThingsBoard PE home interface.

4.6.2 *Kafka Integration*

The next step is to integrate with the Kafka broker in the Confluent platform to receive data messages. The following Figure 38 shows a diagram of the data flow from Kafka to ThingsBoard.

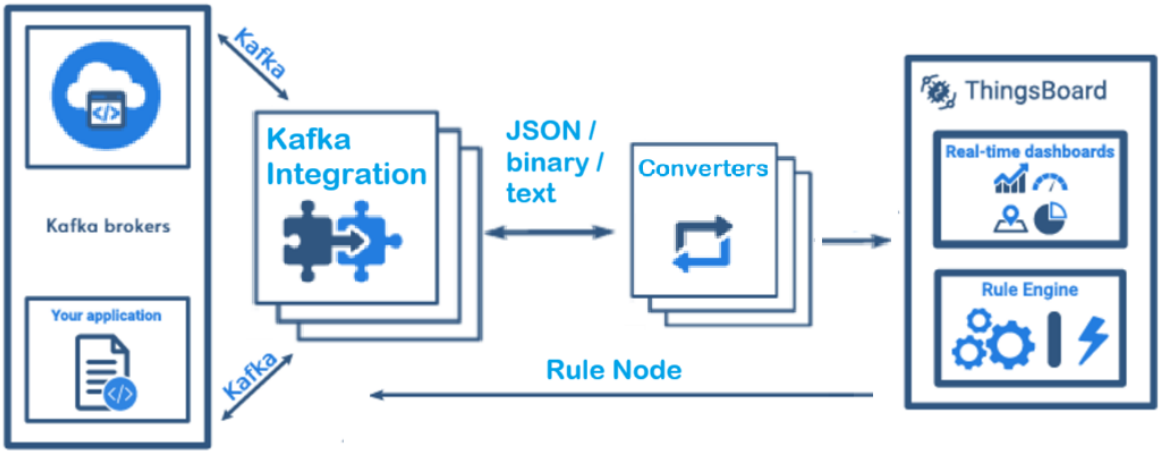


Figure 38. ThingsBoard Kafka Integration.

Adapted from: https://thingsboard.io/docs/paas/user-guide/integrations/kafka/

There exist two elements enabling Kafka message integration. First, an “Integration”, that defines the configuration needed to stablish connection with the Kafka broker, and then a “Data Converter”, which is the element that must transform the message received from Kafka into the data model structure required by ThingsBoard.

4.6.2.1 Data Converter Configuration

On the ThingsBoard interface, navigate to the Integrations Center and select "Data Converters." This section lists existing Data Converters with their creation time, name, and type. To create a new Data Converter, click the "+" button on the upper right side, enter a name, select "Uplink" as the type, and enable "Debug mode".

This Data Converter is JavaScript code that transforms Kafka messages into the specific ThingsBoard data model. The script varies depending on the data payload received from Kafka.

For this implementation, a basic script is provided in Appendix 8.6. The script defines a function that must return a valid JSON document with the following requirements:

1. Must contain deviceName and deviceType, which identify the device (unique within the tenant). If the device doesn't exist and "Allow to create devices" is enabled for the integration, a new entity will be created.
2. Must contain a telemetry object that represents time-series data of the message received.

4.6.2.2 Integration Configuration.

Navigate to the Integrations Center and select "Integrations." This section lists existing integrations with their creation time, name, type, and current status.

To create a new Kafka Integration, click on the button (+) on the upper right side of the screen and follow the following steps.

1. Basic Settings:
    a. Integration Type: Kafka
    b. Activate Enable integration, Debug mode and Allow create devices.
2. Uplink Data Converter: Select the previously created converter in Section 4.6.2.1.
3. In the Connection Tab:
    a. Select an identifier for Thingsboard Group and Client

b. Introduce the Kafka Broker address. In this implementation, “broker:29092”, as this is the address and port internal from Docker. Ensure that both services are in the same Docker internal network.

c. Introduce the Kafka topic to consume the data from. In this implementation, json_data is the topic.

Once these steps are completed, the Kafka integration should be ready.

4.6.2.3 Verifying Integration

To check for new messages being received, it can be accessed the Integration details or Data Converter details. As the Debug option has been activated, in the Events sections, new upcoming Events should be showing, as it is seen in the following Figure 39.

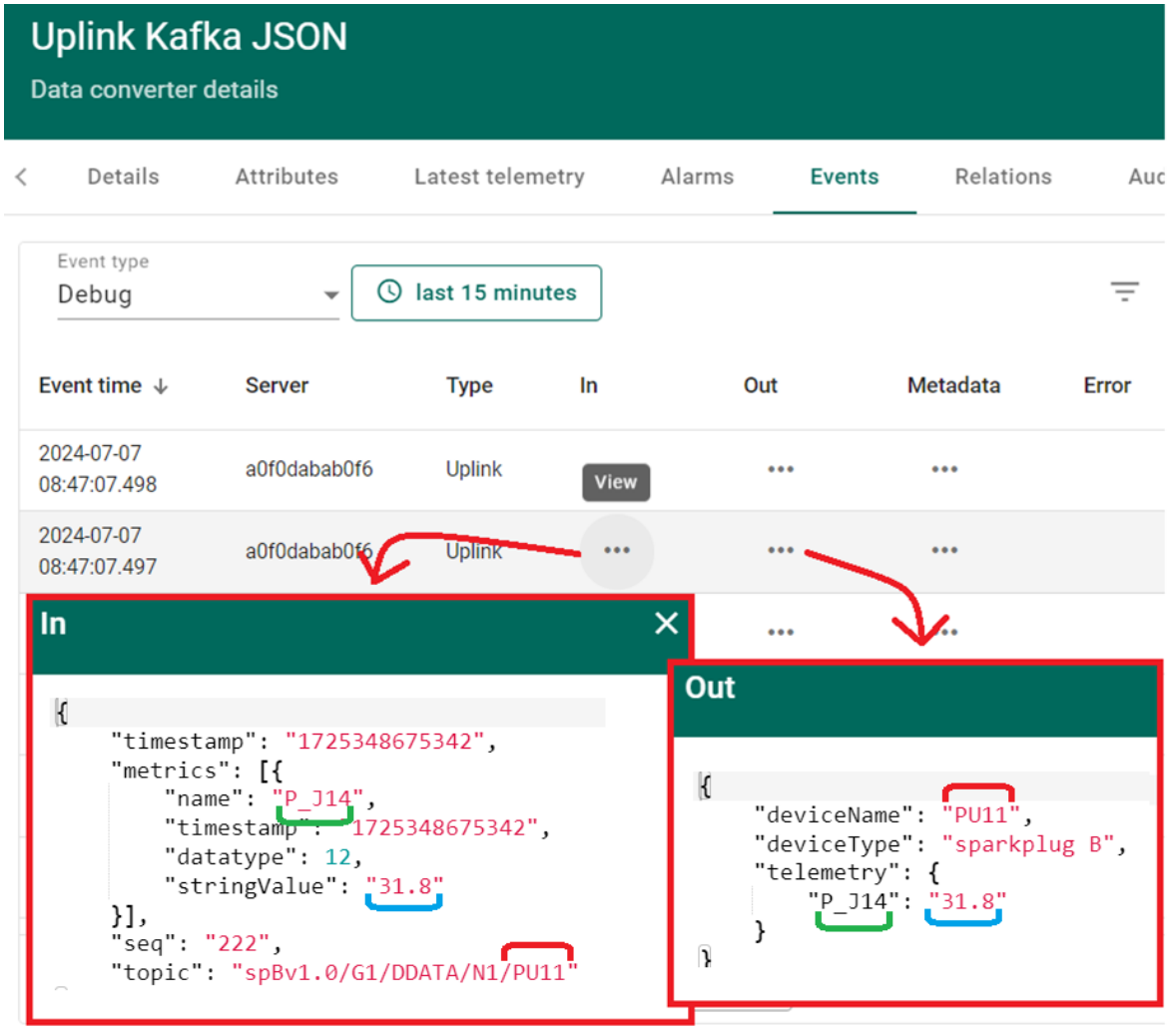


Figure 39. ThingsBoard data transformation from Sparkplug JSON (left) to ThingsBoard data format (right).

Additionally, it can be used a traffic analyzer between containers to capture traffic data, as can be seen in the following Figure 40, where TCP data is received from the Kafka Broker internal port.

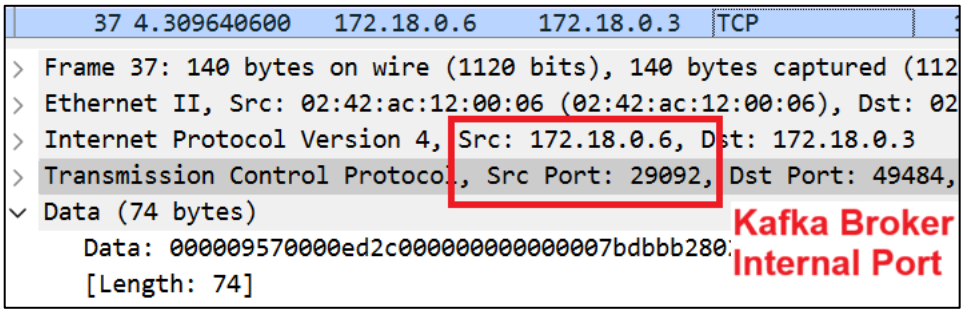


Figure 40. TCP traffic from Confluent to ThingsBoard.

As data is generated for each device, they will be automatically created in ThingsBoard if the "Allow create new devices" has been activated in the integration. This can be checked in the Devices interface, found in the lateral menu under the Entities section. The following Figure 41 shows a set of devices automatically created for this implementation.

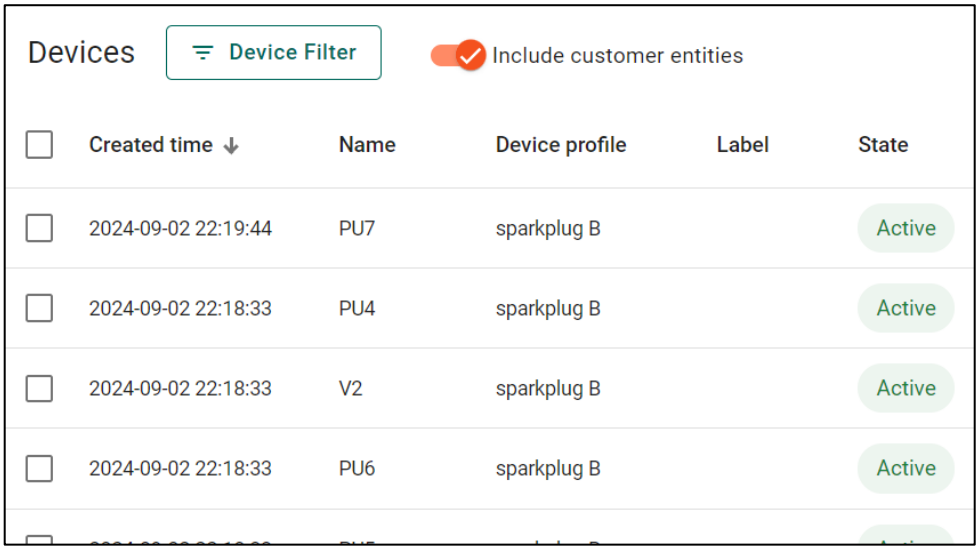


Figure 41. ThingsBoard automatically created devices.

4.6.3 *Create Dashboard*

Once time-series data flows into ThingsBoard, dynamic dashboards can be created to visualize it using various panels. The specific panels used will depend on the data itself.

In the following Figure 42 shows an example dashboard representing real-time measurements received from Kafka, including flow rate of pumps and water level of tanks from different devices.

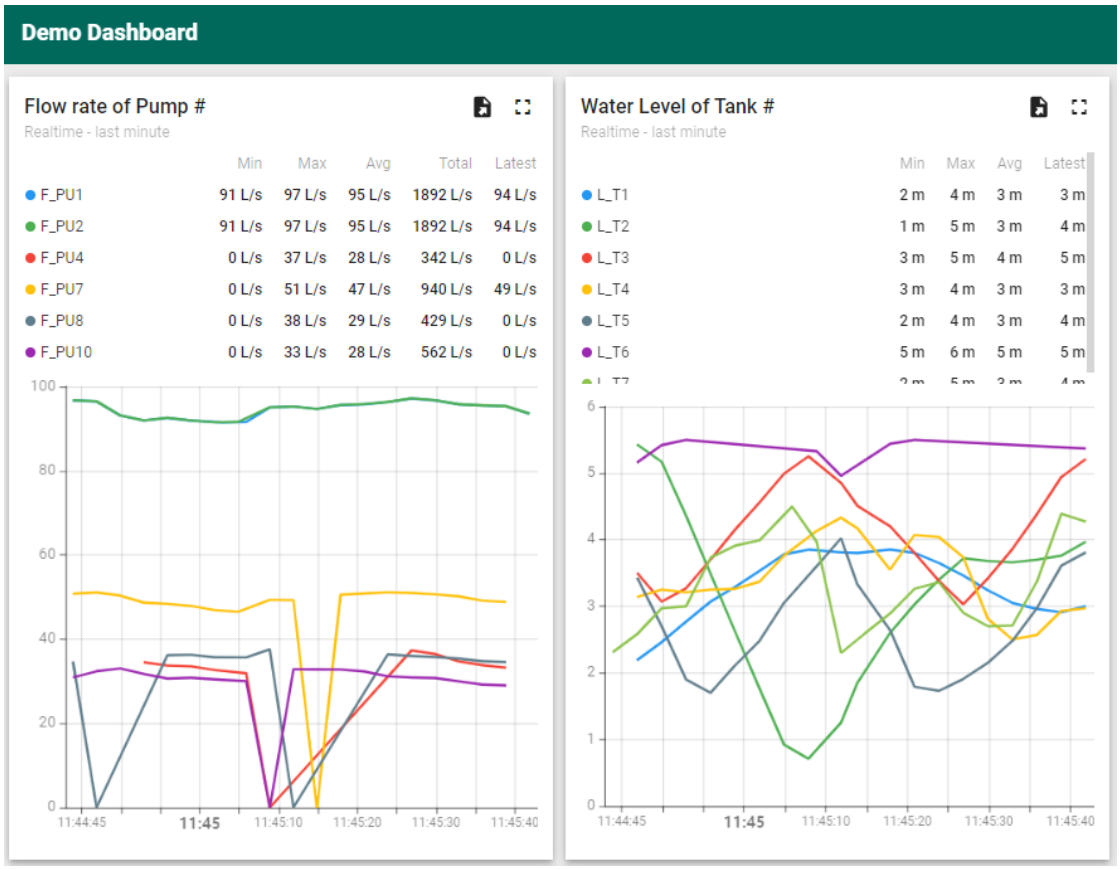


Figure 42. ThingsBoard Dashboard with time-series data.

In the following Figure 43 is shown the final version of the Dashboard, with the addition of a time-series representation of the pressure of junction, a led indicator panel representing the status of one valve, and a “Reboot Device” button.

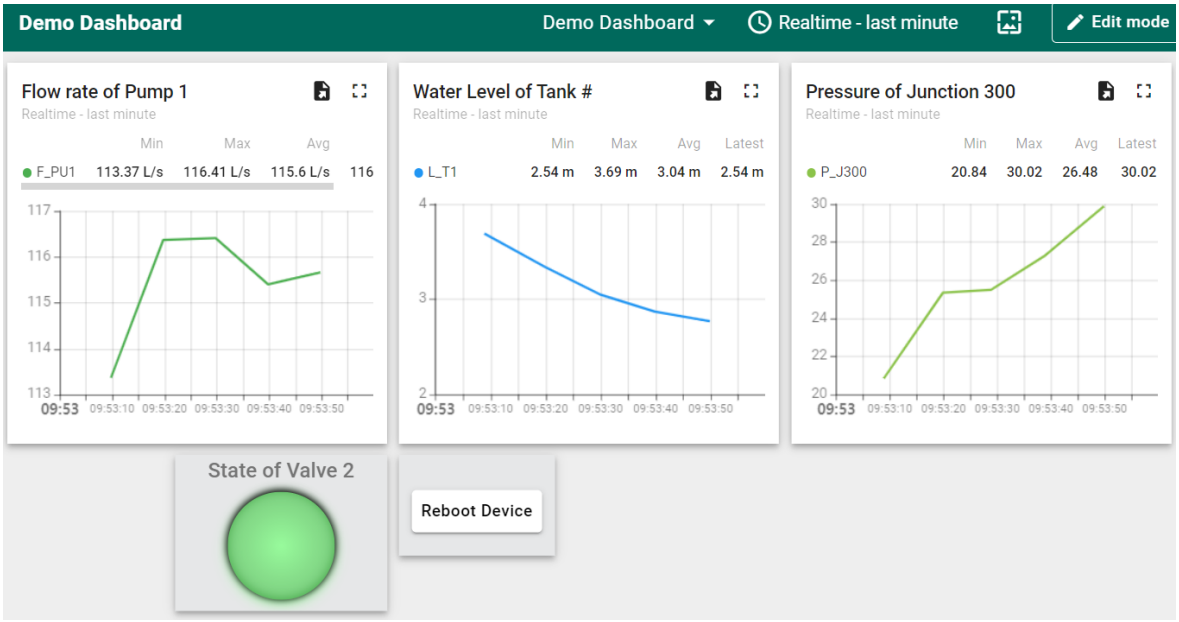


Figure 43. ThingsBoard Dashboard.

4.6.4 *Rule chain configuration*

To send commands back to Kafka, a Rule Chain must be configured. This is triggered by an "Update Device Attribute" button, like the "Reboot Device" button in the previous Figure 43.

This button updates a server-side Attribute of the device with the desired parameter. Following Figure 44 shows the button configuration, including the attribute parameter key-value pair.

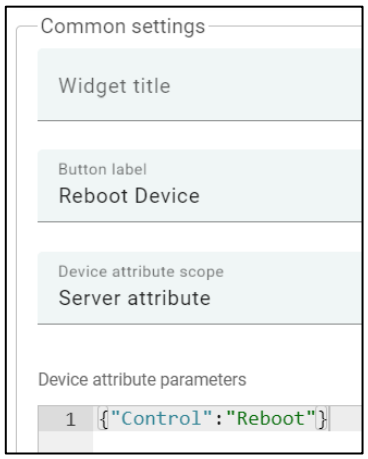


Figure 44. ThingsBoard Update Device Attribute button configuration.

To create the Rule Chain flow, access the Rule Chains windows located in the lateral menu. Modify the Root rule chain, as it is the main rule chain that is activated for each message in ThingsBoard. The final structure of the modifications can be shown in the following Figure 45. Below there is a description of the steps to reproduce.

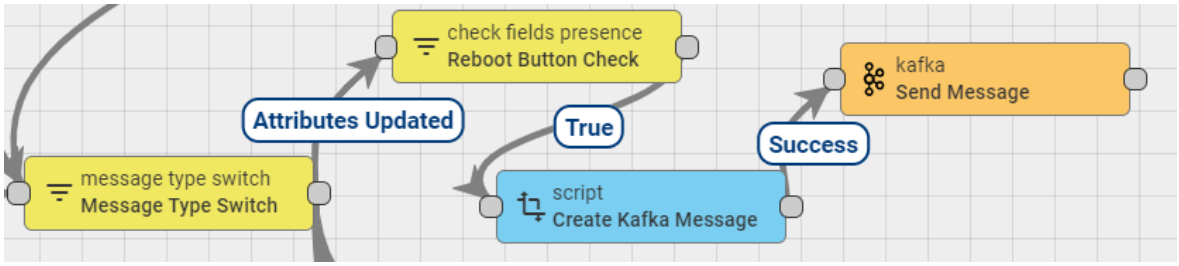


Figure 45. ThingsBoard Rule Chain modification.

1. From the Root rule chain, a new "Check Fields Presence" node can be linked to the "Message Type Switch" node.
2. The "Check Fields Presence" node (yellow block) can be configured to trigger only when attributes are updated and to check if the updated attribute matches the parameter field defined in the "Reboot Button" configuration

3. A "Script" node (blue block) can be linked to the "Check Fields Presence" output, triggered only when the value is True. This script generates a JSON-formatted Sparkplug B message, as it cannot directly create protobuf format.

```
if (msg.Control=="Reboot"){
      newMsg= {
            "timestamp": Date.now(),
            "metrics": [{
                  "name": "Node Control/Rebirth",
                  "timestamp": Date.now(),
                  "datatype": 11, // 11 is Boolean type (sparkplug.proto)
                  "booleanValue": true
            }]
      }
}
return {msg: newMsg};
```

4. To finalize, the "Script" node can be connected to a "Kafka" node (orange block) to establish a connection with the broker and send the JSON message. The "Kafka" node must be configured with the following parameters:
    a. Kafka topic to which the message will be sent. In this implementation, “json_command”
    b. Broker address. In this implementation, “broker:29092”. Ensure that the Kafka broker and Thingsboard are in the same Docker network.
    c. Key serializer: "org.apache.kafka.common.serialization.StringSerializer"
    d. Value serializer: "org.apache.kafka.common.serialization.StringSerializer"

5 EVALUATION OF THE IMPLEMENTATION

This chapter evaluates the end-to-end implementation, including a general overview of the network and data flow with detailed explanations of data conversion steps in both directions. It also summarizes the challenges faced in each section of the previous chapter. To finalize, the general benefits and pitfalls within the implementation are validated.

5.1 Deployed Network

In the following Figure 46 it is shown the network diagrm of all the services and applications that have been deployed.

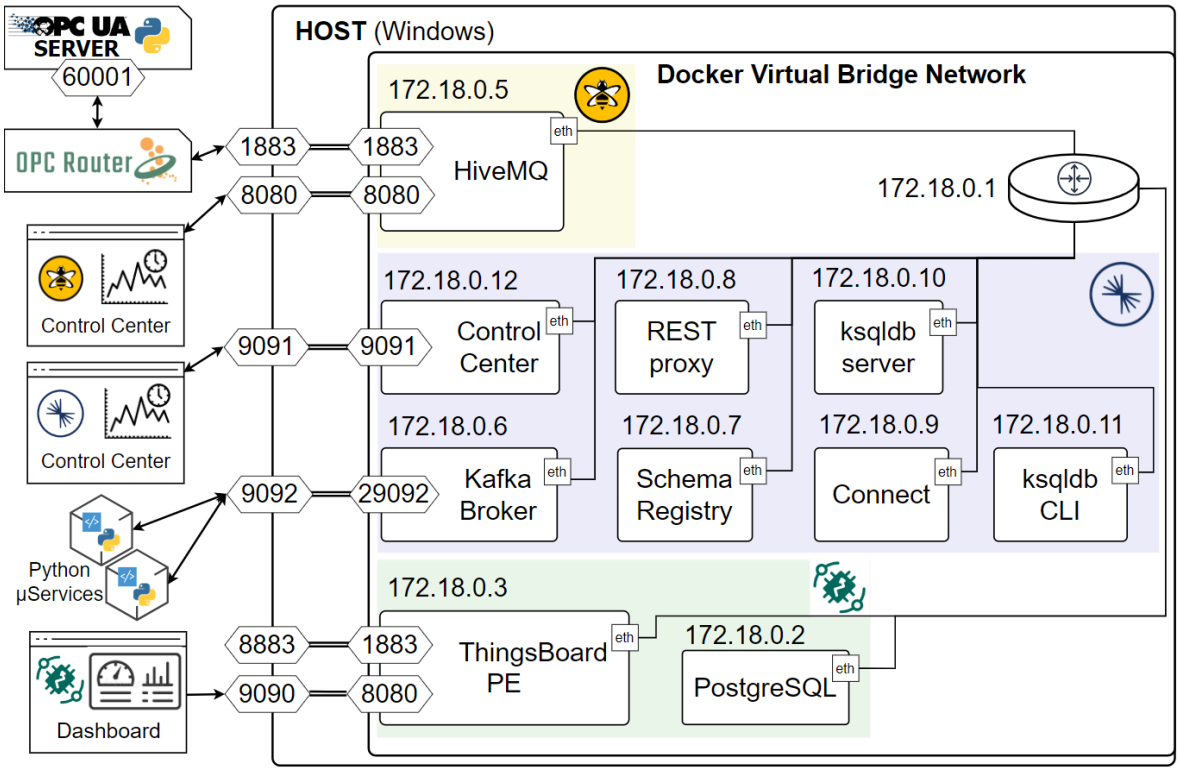


Figure 46. Network Diagram.

In the following Table 6 it is shown the complete list of TCP port mapping configured for HiveMQ.

Table 6. HiveMQ port mapping.

Container name	Host port	Internal Docker port
HiveMQ	1883	1883
	8000	8000
	8080	8080

In the following Table 7 it is shown the complete list of TCP port mapping configured for Confluent Platform.

Table 7. Confluent Platform port mapping

Container name	Host port	Internal Docker port
Kafka Broker	9092	29092
	9101	9101
Connect	8083	8083
Control Center	9021	9021
Ksqldb CLI	-	-
Ksqldb Server	8088	8088
Rest Proxy	8082	8082
Schema Registry	8081	8081
Zookeeper	2181	2181

In the following Table 8 it is shown the complete list of TCP port mapping configured for ThingsBoard IoT Platform.

Table 8. ThingsBoard IoT Platform port mapping.

Container name	Host port	Internal Docker port
ThingsBoard PE	8883	1883
	7070	7070
	9090	8080
PostgreSQL	50667	5432

5.2 End-to-End Uplink Data Flow

To describe the data flow and conversions from the PLC simulated all the way until the ThingsBoard IoT Platform, the following Figure 47 shows the architecture diagram outlining the communication steps from the simulated PLC to the ThingsBoard IoT Platform.

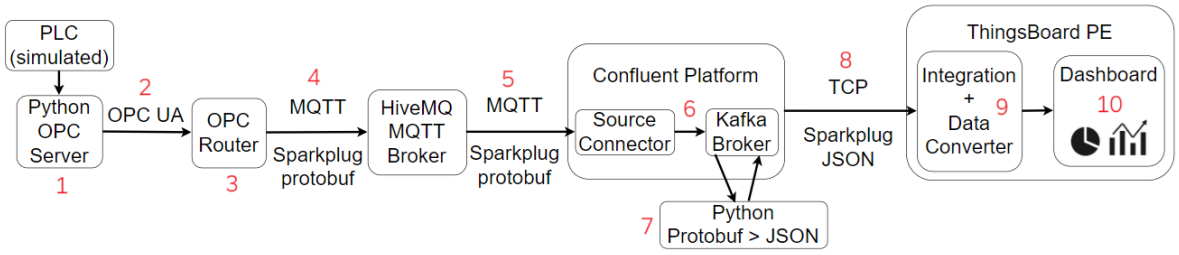


Figure 47. Uplink data flow from end-to-end architecture.

1. The Python OPC Server generate the simulated PLC data structure and variables. It updates the entry of each defined variable of each device in an infinite loop.
2. OPC Router is configured to monitors the designated OPC UA objects and triggers connections for each data update using Report By Exception communication with the OPC Server
3. OPC Router generates MQTT messages in Sparkplug B format for each OPC UA data received.
4. OPC Router publishes MQTT Spkarplug B messages to the HiveMQ MQTT Broker using the correspondan Sparkplug B topic for each device.
5. HiveMQ receives the published messages and route them for subscribed clients without any processing.
6. The MQTT Source connector in Confluent Platform is previously subscribed to the MQTT Broker and produces the received messages (Sparkplug Data messages) into a Kafka topic.
7. A microservice running a Python script consumes the Kafka topic, parses Protobuf messages into JSON messages, and produces the JSON messages into a new Kafka topic.
8. ThingsBoard, configured to integrate with Kafka, consumes the JSON messages from the Kafka topic.
9. A Data Converter script within ThingsBoard transforms the messages into the ThingsBoard data format and store it in the corresponding device
10. Real-time data can be monitored on a ThingsBoard Dashboard.

5.3 End-to-End Downlink Data Flow

To describe the data flow and conversions from the PLC simulated all the way until the ThingsBoard IoT Platform, refer to the following Figure 48 to see a diagram of the architecture to see the communication steps.

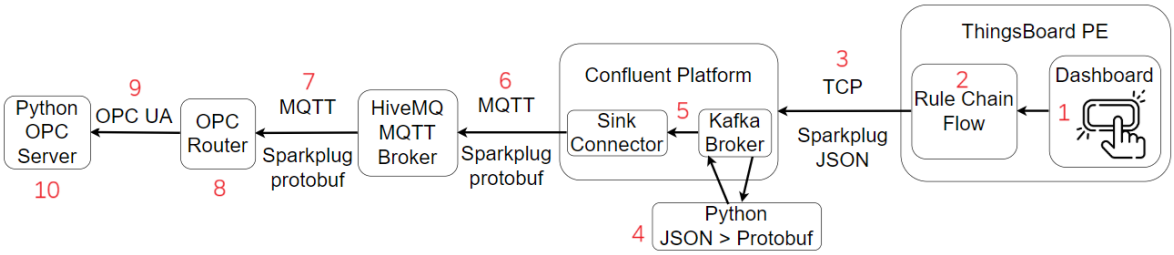


Figure 48. Downlink data flow from end-to-end architecture.

1. A Dashboard in ThingsBoard contains a "Reboot Device" button. Pressing this button triggers a Rule Chain Flow
2. In the Rule Chain, a script is activated which generates a Sparkplug B message in JSON format.
3. The final node in the Rule Chain produces the JSON message into a Kafka topic.
4. A microservice running a Python script consumes the Kafka topic, parses the JSON message into Sparkplug B Protobuf format, and produces the result to a new Kafka topic.
5. The MQTT Sink connector in Confluent Platform consumes the Protobuf data
6. The MQTT Sink connector publishes the data as a MQTT message to the HiveMQ Broker using the configured Sparkplug B Topic.
7. HiveMQ receives the published messages and route them for subscribed clients without any processing.
8. OPC Router, receives the MQTT message and triggers a connection. Generates a OPC UA message to the received parameter.
9. OPC Router sends an OPC UA update variable message to the OPC Server,
10. The Python OPC Server changes the variable value, which the device understand as a Reboot trigger.

5.4 Challenges and Solutions

This section summarizes the challenges and solutions that have been faced during the implementation of the end-to-end architecture.

5.4.1 *Data Source Limitation*

A more in-depth discussion of this challenge can be found in Section 4.1.

Challenge: One of the assumptions for the master's thesis was to use simulated devices. Real-world data from private companies is often subject to non-disclosure agreements due to privacy and security concerns.

Solution: Obtain device data from public datasets available, while many public datasets are preprocessed or database-oriented, making them less suitable for simulating real OT device data.

5.4.2 *OPC Server*

A more in-depth discussion of this challenge can be found in Section 4.2.1.

Challenge: KEPserverEX was initially considered for simulating real devices but was limited to simulating data using mathematical functions like Random or Ramp.

Solution: A custom OPC Server script in Python was implemented to successfully read prepared Excel files, where each column represents a device variable, and each row represents a different iteration of its value.

5.4.3 *OPC Router Sparkplug Plugin*

A more in-depth discussion of this challenge can be found in Section 4.3.3.

Challenge: The Sparkplug plugin in OPC Router for receiving Sparkplug commands encountered internal errors, as it is in development phase.

Solution: With official support, a workaround was necessary to manually trigger the Sparkplug command using a separate connection.

5.4.4 *Confluent Self-managed Connectors*

A more in-depth discussion of this challenge can be found in Section 4.5.1.

Challenge: The public Docker image of the Kafka Connect service lacked the MQTT connector for integration.

Solution: The connector was manually downloaded and installed within the Docker container. A better solution is to configure the Docker Compose file to execute commands during deployment, ensuring the desired MQTT connectors are always installed.

5.4.5 *Kafka Integration in ThingsBoard.*

A more in-depth discussion of this challenge can be found in Section 4.5.4.

Challenge: ThingsBoard does not natively decode Sparkplug B (protobuf) messages.

- ThingsBoard officially support Sparkplug B messages. Unfortunately, Sparkplug API is for decoding MQTT messages. Does not integrate with Kafka.
- Convert Protobuf messages to JSON using Kafka Schemas was proposed. Created a new Challenge: Kafka does not support recursives Schemas as the Sparkplug Protobuf schema.

Solution: Microservices running Python scripts were implemented to consume Kafka topics, transform Sparkplug Protobuf to JSON (and vice versa), and produce the transformed data to new Kafka topics. ThingsBoard is able to decode JSON messages.

5.5 Validation of the Architecture

This section aims to validate the architecture implemented in previous Chapter 4. Based on the architectural methodology benefits and pitfalls proposed in Section 3.3.2.3, the following statements are confirmed:

Benefits:

OT/IT Integration: HiveMQ ensures the best solution to achieve the integration of OT devices, as the OPC Router acting as a Gateway, while Confluent Platform with its Kafka broker ensures a high throughput platform to route multiple streams of data into IT applications, as the ThingsBoard IoT Platform.

Data Processing: By incorporating an event-streaming platform like Confluent, the limitation of being unable to process data directly within an HiveMQ is solved.

Historical Data: Data logs can be store into Confluent Kafak topics for long periods of time, as topics in HiveMQ is unable to do.

Increased Scalability: Both HiveMQ and Confluent are highly scalable as cluster services. This benefit has not been fully tested in this implementation, as the number of devices and data throughput is limite.

Increased Reliability: The integration of Confluent ensures high availability and fault tolerance mechanisms. If ThingsBoard has a network failure, it tracks the last received log and after the reconnection all generated messages during the downtime are received.

Pitfalls:

Complexity: Implementation is more complex than using a single technology. The complexity can be reduced by using Cloud Managed services as Confluent Cloud. This would increment the costs.

Cost: Implementing and maintaining both HiveMQ and Confluent involve an increase of costs, including hardware, software capabilities, and management.

Based on the general challenges in the implementations of UNS systems proposed in Section 3.3.3, the following statements are confirmed:

Challenges:

Complex Data Models: This challenge has not been faced as the use of Sparkplug specification includes the Payload Structure, Payload Coding and Topic

Hierarchy and Message Types, facilitating the integration of an easy to use UNS Data Model.

Inconsistent Data Structure: This challenge has been faced, as the incompatibility of data structures from Kafka Schemas and Sparkplug B Protobuf, or the incompatibility of data structure from Sparkplug B to ThingsBoard own data structure have been faced. At least, these challenges have been solved using data transformation methods.

Security: This challenge has not been faced as the scope of the thesis is to do an evaluation and review of Unified Namespaces in Industrial IoT architectures does not cover the extensive analysis necessary to test the security vulneratbilities of the implementation.

6 CONCLUSSIONS

As the primary conclusion, it can be stated that the main objective of the master's thesis has been successfully achieved. A comprehensive evaluation and review of Unified Namespaces in Industrial IoT architectures have been conducted in Section 3.3, and Chapters 4 and 5.

Additionally, secondary objectives presented in Chapter 2 have been met:

- Unified Namespace motivation and justification for its use have been discussed in Sections 3.1.1, 3.3 and 5.4.
- Comparative analysis of Unified Namespaces and alternative architectures have been discussed in Section 3.3.2.
- Industry and research landscape of Unified Namespaces have been discussed in Chapter 1 and Section 3.3.
- Design and implementation of a Unified Namespaces End-to-End Architecture have been carried on in Chapter 4.
- Evaluation of benefits and pitfalls for Unified Namespaces architectures have been developed in Section 3.3 and 5.4.

Through the literature research, design, implementation, and writing of this master's thesis, the following additional conclusions have been drawn:

- Deployment of architectures for Industry 4.0 and Digital Transformation initiatives within businesses can follow the UNS principles to obtain a better understanding of their real-time data management from their sources to destinations.
- Unified Namespace and ISA 95 are actually complementary to each other. It is possible to design a smart manufacturing system based on UNS and use ISA-95 standard for modeling the data objects inside the UNS. Therefore, it is possible to use an innovative way of organising components of manufacturing system in conjunction with a globally used functional modeling standard.
- The combination of MQTT and Kafka offers a complementary relation, where both technologies effectively address the limitations of each other.
- The complementary nature of MQTT and Kafka make them a perfect fit for an IIoT architecture core. OT devices obtain a wide adopted protocol to send data and IT applications obtain an efficient event-driven platform with high real-time data throughput.

- Sparkplug is gaining popularity among IIoT device vendors, making it a desirable feature for clients. There exists the possibility that Sparkplug evolve into a widely adopted standard, ensuring long-term compatibility
- Sparkplug is not intended to replace OPC UA as the unique industrial communication protocol.
- Standardization is key to achieving interoperability between devices and applications from different vendors. This allows devices and applications from various manufacturers to seamlessly exchange data within the same MQTT infrastructure.
- Unified Namespaces can get beneficial results by using the Sparkplug specification using their data model and categorization, addressing the complex task to define a dedicated UNS data model for each business orgaization.

7 FUTURE LINES

This section outlines potential future research directions to further enhance the implemented architecture.

- **Deploying the architecture using cloud services** instead of local deployments could provide additional insights. This option was discarded during the thesis under the possibility to introduce complexities related to networking and security firewalls, rather than focusing on the main objectives.
- **Integrating real devices** would provide valuable insights into the architecture's performance in a practical setting. However, this would have been time-consuming within the scope of the thesis.
- **Deploying clusters** of HiveMQ, Kafka, and ThingsBoard would allow for testing the architecture's scalability limits and identifying the resource consumptions of each service in a real scenario.
- **Conducting a comprehensive performance evaluation** would involve identifying bottlenecks, measuring response delays, and determining the maximum throughput capacity of the network. This would provide valuable insights for optimizing the architecture and ensuring its suitability for various workloads.

8 APPENDICES

These appendices provide the Python microservices code and Docker Compose files necessary to deploy the Confluent Platform and Thingsboard PE Platform. These resources serve as practical examples and helps to understanding the implementation.

8.1 Python OPC UA Server

```
import asyncio
import csv
import logging
from asyncua import Server

# Define a list of device names to be processed
devices_string = ["dataset_pressure", "dataset_flowpump"]

async def csv_to_variables(filename, obj, ns):
    """
    Read a CSV file, creates variables in the namespace based on column names,
    and returns the list of variables and remaining data rows.

    Args:
        filename (str): The name of the CSV file to read.
        obj : The OPC UA object to add variables into.
        ns : The namespace id to use for the variables.
    Returns:
        - Array: List of variables OPC UA objects.
        - Matrix: Data rows from the CSV file (excluding the header row)
    """
    try:
        with open(filename, 'r') as file:
            reader = csv.reader(file)
            # Load all rows from the CSV file into memory
            rows = list(reader)
            # Extract column names from the first row
            column_names = rows[0]

            # Create a list to store the created variables
            variables = []
            for var_name in column_names:
                # Create a new OPC UA variable
                new_var = await obj.add_variable(ns, var_name, "")
                # Allow clients to write to the variable
                await new_var.set_writable()
                variables.append(new_var)

            # Return the created variables and the remaining data rows (excluding
header)
```

```
            return variables, rows[1:]
    except Exception as e:
        print("Error reading CSV file:", e)
        return None, None

async def main():
    """
    Sets up the OPC UA server, reads CSV files for each device,
    creates variables, and continuously updates the variables with data from the CSV
files.
    """
    _logger = logging.getLogger(__name__)

    # Create an OPC UA server instance
    server = Server()
    await server.init()
    # Set the endpoint for the server
    server.set_endpoint("opc.tcp://127.0.0.1:60001/freeopcua/server/")

    # Register a namespace for the server
    uri = "http://examples.freeopcua.github.io" # whatever uri
    ns = await server.register_namespace(uri)

    # Create a parent object to hold device objects created after
    simulated_devices = await server.nodes.objects.add_object(ns, "SimulatedDevices")

    # Create each device objects based on the device names strings,
    devices_obj = []
    devices_variables = []
    devices_values_rows = []
    devices_n_rows = []

    for device_string in devices_string:
        new_device_obj = await simulated_devices.add_object(ns, device_string)
        devices_obj.append(new_device_obj)
        # Then, initialize lists to store variables, data rows, and row counters
        variables, values_rows = await csv_to_variables(device_string + ".csv",
new_device_obj, ns)
        devices_variables.append(variables)  # Store variables of the device
        devices_values_rows.append(values_rows)  # Store data rows of the device
        devices_n_rows.append(len(values_rows) - 1)  # Calculate number of rows of the
device
        # (len-1 bc will be used to check index values)

    # Array to track the current row index for each device
    iterable_n = [0] * len(devices_n_rows)

    _logger.info("Starting server!")
```

```
    async with server:
        while True:
            '''
            Update Device (d) OPC variables with values of one row (n)
            Check if (n) is the last row, then next will be initial row 0
            Hop to next Device (d)
            '''
            for d, device_variables in enumerate(devices_variables):
                print("DEBUG ####################################")
                # print(" Device d: "+str(d))
                # print(" Row n: "+str(iterable_n[d]))
                print("Setting variables of device *{}* to
values:\n".format(devices_string[d]), devices_values_rows[d][iterable_n[d]])
                print("#################################### END ")

                for var, value in zip(device_variables,
devices_values_rows[d][iterable_n[d]]):
                    # Write the current data value to the OPC UA variable
                    await var.write_value(value)
                    _logger.info("Set value of %s to %s", var, value)

                # Check if the end of the data rows for the device is reached
                if iterable_n[d] == devices_n_rows[d]:
                    iterable_n[d] = 0  # Reset the row index to the beginning
                else:
                    iterable_n[d] += 1  # Increment the row index

                # Adjust the sleep time as needed
                await asyncio.sleep(5)

if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO)
    asyncio.run(main(), debug=True)
```

8.2 Python Sparkplug B Protobuf decoder to JSON format

```
from confluent_kafka import Consumer, Producer
from google.protobuf.json_format import MessageToJson
from sparkplug_b_pb2 import Payload
import json

errors=[] # debug: if exception at parse_proto_json, value is stored

def parse_proto_json(value):
  try:
    message_bytes = value[5:] #slice the first 5 bytes (Who includes this?
Kafka/Hivemq/OPCRouter)
    # Parse the message
    message = Payload()
    message.ParseFromString(message_bytes)

    # Convert to JSON string
    message_json = MessageToJson(message)

    return message_json
```

```
  except:
    print(f"Error decoding message: {value}")
    errors.append(value)
    return None  # Or handle decoding errors differently

# Kafka configuration (assuming Confluent Platform runs locally)
bootstrap_servers = 'localhost:9092'
consumer_group_id = 'my-python-consumerx'
producer_group_id = 'my-python-producerx'
consumer_topic = 'mqtt_data'
producer_topic = 'json_data'

# Create Consumer and Producer objects
consumer = Consumer({
    'bootstrap.servers': bootstrap_servers,
    'group.id': consumer_group_id,
    'auto.offset.reset': 'earliest'
})

producer = Producer({
    'bootstrap.servers': bootstrap_servers,
})

# Subscribe to the "mqtt_all" topic
consumer.subscribe([consumer_topic])

try:
  while True:
    msg = consumer.poll(timeout=1.0)  # Poll for new messages with a 1-second
timeout
    if msg is None:
      continue

    if msg.error():
      print("Error: {}".format(msg.error()))
      break

    # Process the message
    processed_data = parse_proto_json(msg.value())
    # Add "topic" key-value pair to the JSON data
    data = json.loads(processed_data)  # Convert JSON string to dictionary
    data["topic"] = msg.key().decode('utf-8')  # Add topic key with message key
value
    processed_data = json.dumps(data)  # Convert dictionary back to JSON string
    print(processed_data)

    # Check if data is valid after processing
    if processed_data is not None:
      # Produce the processed data to the producer topic
      producer.produce(producer_topic, processed_data, headers=msg.headers(),
key=msg.key())
      producer.poll(0)  # Flush outstanding deliveries

except KeyboardInterrupt:
  pass

finally:
  # Close consumer and producer connections
  consumer.close()
  producer.flush()  # Flush any remaining messages

print("Program terminated.\n ERRORS:")
print(errors)
```

8.3 Python Sparkplug B Protobuf coder from JSON format

```
from confluent_kafka import Consumer, Producer
from google.protobuf.json_format import Parse, ParseDict
from sparkplug_b_pb2 import Payload
import json

errors = []  # debug: stores undecodable messages

def parse_json_to_proto(json_data):

    # Parse JSON string to dictionary
    #data = json.loads(json_data)

    # Create a new empty Payload messag
    # or ParseDict(data, Payload())
    message = Parse(json_data, Payload())
    print(message)
    return message


# Kafka configuration
bootstrap_servers = 'localhost:9092'
consumer_group_id = 'my-python-consumerX'
consumer_topic = 'json_command'
producer_topic = 'mqtt_command'

# Create Consumer and Producer objects
consumer = Consumer({
    'bootstrap.servers': bootstrap_servers,
    'group.id': consumer_group_id,
    'auto.offset.reset': 'earliest'
})

producer = Producer({
    'bootstrap.servers': bootstrap_servers,
})

# Subscribe to the "json_downlink" topic
consumer.subscribe([consumer_topic])

try:
  while True:

    msg = consumer.poll(timeout=1.0)  # Poll for new messages with a 1-second
timeout
    if msg is None:
      continue

    if msg.error():
      print("Error: {}".format(msg.error()))
      break

    # Process the message
    processed_data = parse_json_to_proto(msg.value())
    #val = b'{"timestamp":1720443357807,"metrics":[{"name":"Node
Control/Reboot","timestamp":1720443357807,"datatype":11,"booleanValue":true}]}'
    #processed_data = parse_json_to_proto(val)
    # Check if data is valid after processing
    if processed_data is not None:
      # Serialize message to bytes
      message_bytes = processed_data.SerializeToString()

      # Produce the serialized data to the producer topic
      producer.produce(producer_topic, message_bytes, headers=msg.headers(),
key=msg.key())
      producer.poll(0)  # Flush outstanding deliveries
```

```
except KeyboardInterrupt:
  pass

finally:
  # Close consumer and producer connections
  consumer.close()
  producer.flush()  # Flush any remaining messages

print("Program terminated.\n ERRORS:")
print(errors)
```

8.4 Confluent Platform Docker Compose File

```
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:7.6.1
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  broker:
    image: confluentinc/cp-server:7.6.1
    hostname: broker
    container_name: broker
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
      - "9101:9101"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://broker:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_METRIC_REPORTERS:
io.confluent.metrics.reporter.ConfluentMetricsReporter
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
      KAFKA_CONFLUENT_LICENSE_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_CONFLUENT_BALANCER_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_JMX_PORT: 9101
      KAFKA_JMX_HOSTNAME: localhost
      KAFKA_CONFLUENT_SCHEMA_REGISTRY_URL: http://schema-registry:8081
      CONFLUENT_METRICS_REPORTER_BOOTSTRAP_SERVERS: broker:29092
      CONFLUENT_METRICS_REPORTER_TOPIC_REPLICAS: 1
      CONFLUENT_METRICS_ENABLE: 'true'
      CONFLUENT_SUPPORT_CUSTOMER_ID: 'anonymous'

  schema-registry:
    image: confluentinc/cp-schema-registry:7.6.1
    hostname: schema-registry
    container_name: schema-registry
    depends_on:
      - broker
    ports:
      - "8081:8081"
    environment:
      SCHEMA_REGISTRY_HOST_NAME: schema-registry
      SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS: 'broker:29092'
```

```
      SCHEMA_REGISTRY_LISTENERS: http://0.0.0.0:8081

  connect:
    image: cnfldemos/cp-server-connect-datagen:0.6.4-7.6.0
    hostname: connect
    container_name: connect
    depends_on:
      - broker
      - schema-registry
    ports:
      - "8083:8083"
    environment:
      CONNECT_BOOTSTRAP_SERVERS: 'broker:29092'
      CONNECT_REST_ADVERTISED_HOST_NAME: connect
      CONNECT_GROUP_ID: compose-connect-group
      CONNECT_CONFIG_STORAGE_TOPIC: docker-connect-configs
      CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_OFFSET_FLUSH_INTERVAL_MS: 10000
      CONNECT_OFFSET_STORAGE_TOPIC: docker-connect-offsets
      CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_STATUS_STORAGE_TOPIC: docker-connect-status
      CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1
      CONNECT_KEY_CONVERTER: org.apache.kafka.connect.storage.StringConverter
      CONNECT_VALUE_CONVERTER: io.confluent.connect.avro.AvroConverter
      CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL: http://schema-registry:8081
      # CLASSPATH required due to CC-2422
      CLASSPATH: /usr/share/java/monitoring-interceptors/monitoring-interceptors-
7.6.1.jar
      CONNECT_PRODUCER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"
      CONNECT_CONSUMER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"
      CONNECT_PLUGIN_PATH: "/usr/share/java,/usr/share/confluent-hub-components"
      CONNECT_LOG4J_LOGGERS:
org.apache.zookeeper=ERROR,org.I0Itec.zkclient=ERROR,org.reflections=ERROR
    command:
      - bash
      - -c
      - |
        echo "Installing Connector"
        confluent-hub install --no-prompt confluentinc/kafka-connect-mqtt:1.7.0
        # IMPORTANT - WITHOUT THIS CONFIGURATION, CONFLUENT DOES NOT HAVE MQTT
CONNECTOR TO SINK/SOURCE
        echo "Launching Kafka Connect worker"
        /etc/confluent/docker/run &
        #
        sleep infinity

  control-center:
    image: confluentinc/cp-enterprise-control-center:7.6.1
    hostname: control-center
    container_name: control-center
    depends_on:
      - broker
      - schema-registry
      - connect
      - ksqldb-server
    ports:
      - "9021:9021"
    environment:
      CONTROL_CENTER_BOOTSTRAP_SERVERS: 'broker:29092'
      CONTROL_CENTER_CONNECT_CONNECT-DEFAULT_CLUSTER: 'connect:8083'
      CONTROL_CENTER_KSQL_KSQLDB1_URL: "http://ksqldb-server:8088"
      CONTROL_CENTER_KSQL_KSQLDB1_ADVERTISED_URL: "http://localhost:8088"
      CONTROL_CENTER_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
      CONTROL_CENTER_REPLICATION_FACTOR: 1
      CONTROL_CENTER_INTERNAL_TOPICS_PARTITIONS: 1
      CONTROL_CENTER_MONITORING_INTERCEPTOR_TOPIC_PARTITIONS: 1
```

```
      CONFLUENT_METRICS_TOPIC_REPLICATION: 1
      PORT: 9021

  ksqldb-server:
    image: confluentinc/cp-ksqldb-server:7.6.1
    hostname: ksqldb-server
    container_name: ksqldb-server
    depends_on:
      - broker
      - connect
    ports:
      - "8088:8088"
    environment:
      KSQL_CONFIG_DIR: "/etc/ksql"
      KSQL_BOOTSTRAP_SERVERS: "broker:29092"
      KSQL_HOST_NAME: ksqldb-server
      KSQL_LISTENERS: "http://0.0.0.0:8088"
      KSQL_CACHE_MAX_BYTES_BUFFERING: 0
      KSQL_KSQL_SCHEMA_REGISTRY_URL: "http://schema-registry:8081"
      KSQL_PRODUCER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringProducerInterceptor"
      KSQL_CONSUMER_INTERCEPTOR_CLASSES:
"io.confluent.monitoring.clients.interceptor.MonitoringConsumerInterceptor"
      KSQL_KSQL_CONNECT_URL: "http://connect:8083"
      KSQL_KSQL_LOGGING_PROCESSING_TOPIC_REPLICATION_FACTOR: 1
      KSQL_KSQL_LOGGING_PROCESSING_TOPIC_AUTO_CREATE: 'true'
      KSQL_KSQL_LOGGING_PROCESSING_STREAM_AUTO_CREATE: 'true'

  ksqldb-cli:
    image: confluentinc/cp-ksqldb-cli:7.6.1
    container_name: ksqldb-cli
    depends_on:
      - broker
      - connect
      - ksqldb-server
    entrypoint: /bin/sh
    tty: true

  rest-proxy:
    image: confluentinc/cp-kafka-rest:7.6.1
    depends_on:
      - broker
      - schema-registry
    ports:
      - 8082:8082
    hostname: rest-proxy
    container_name: rest-proxy
    environment:
      KAFKA_REST_HOST_NAME: rest-proxy
      KAFKA_REST_BOOTSTRAP_SERVERS: 'broker:29092'
      KAFKA_REST_LISTENERS: "http://0.0.0.0:8082"
      KAFKA_REST_SCHEMA_REGISTRY_URL: 'http://schema-registry:8081'
```

8.5 ThingsBoard PE Docker Compose File

```
version: '3.0'
services:
  mytbpe:
    restart: always
    image: "thingsboard/tb-pe:3.6.4PE"
    ports:
      - "9090:8080"
      - "8883:1883"
      - "7070:7070"
      - "5683-5688:5683-5688/udp"
    environment:
      TB_QUEUE_TYPE: in-memory
```

```
      SPRING_DATASOURCE_URL: jdbc:postgresql://postgres:5432/thingsboard
      TB_LICENSE_SECRET: z9tZ***************
      TB_LICENSE_INSTANCE_DATA_FILE: /data/license.data
    volumes:
      - mytbpe-data:/data
      - mytbpe-logs:/var/log/thingsboard
  postgres:
    restart: always
    image: "postgres:15"
    ports:
    - "5432"
    environment:
      POSTGRES_DB: thingsboard
      POSTGRES_PASSWORD: postgres
    volumes:
      - mytbpe-data-db:/var/lib/postgresql/data
volumes:
  mytbpe-data:
    external: true
  mytbpe-logs:
    external: true
  mytbpe-data-db:
    external: true
```

8.6 ThingsBoard Data Converter

```
// decode payload to JSON
var data = decodeToJson(payload);

// Split the topic string into an array
var topicParts = data.topic.split("/");

// Check if message is from a Device (3rd element is DDATA)
if (topicParts.length >= 3 && topicParts[2] === "DDATA") {
  // Extract device name from the last element (5th element)
  var deviceName = topicParts.pop();

  // Build telemetry object with key/value from two metrics name and stringvalue
  var telemetry = {};  // Extra opening brace here (not needed)
  //telemetry[data.metrics[0].stringValue] = data.metrics[1].stringValue;
 var key = data.metrics[0].name;
 var value = data.metrics[0].stringValue;
 telemetry[key] = value;
  // Determine deviceType based on your logic (replace with your logic)
  var deviceType = 'sparkplug B';

  var result = {
    deviceName: deviceName,
    deviceType: deviceType,
    telemetry: telemetry,
  };

  return result;
} else {

  return null; // Or handle non-device messages differently
}
```

References

[1] G. S. Sestito *et al.,* "A Method for Anomalies Detection in Real-Time Ethernet Data Traffic Applied to PROFINET," *IEEE Trans. Ind. Inf.*, vol. 14, no. 5, pp. 2171–2180, 2018, doi: 10.1109/TII.2017.2772082.

[2] G. Shapira, T. Palino, R. Sivaram, and K. Petty, *Kafka: The definitive guide real-time data and stream processing at scale*. Sebastopol CA: O'Reilly, 2022. Accessed: Sep. 5 2024.

[3] L. Sujay Vailshery, *Number of IoT connections worldwide 2022-2033, with forecasts to 2030.* [Online]. Available: https://www.statista.com/statistics/1183457/iot-connected-devices-worldwide/ (accessed: Sep. 5 2024).

[4] CORSO SYSTEMS, *Everything You Need to Know About ISA-95.* [Online]. Available: https://corsosystems.com/posts/isa-95-101 (accessed: Sep. 5 2024).

[5] International Society of Automation (ISA), *ISA Standards by Topic: Enterprise-Control System Integration (ISA95).* [Online]. Available: https://www.isa.org/standards-and-publications/isa-standards/find-isa-standards-by-topic (accessed: Sep. 5 2024).

[6] J. Theocharis, *Operational Technology (OT): Automation pyramid.* [Online]. Available: https://learn.umh.app/lesson/introduction-into-it-ot-automation-pyramid/ (accessed: Sep. 6 2024).

[7] OPC Router, *What is OPC UA? A practical introduction: 3. OPC Classic vs. OPC UA.* [Online]. Available: https://www.opc-router.com/what-is-opc-ua/#OPC-Classics-OPC-UA (accessed: Jul. 20 2024).

[8] OPC Foundation, *OPC 10000-1: UA Part 1: Overview and Concepts: 3.2.1 AddressSpace.* [Online]. Available: https://reference.opcfoundation.org/Core/Part1/v105/docs/3.2.1 (accessed: Jul. 20 2024).

[9] OPC Foundation, *NodeSets.* [Online]. Available: https://reference.opcfoundation.org/nodesets (accessed: Sep. 5 2024).

[10] Eclipse Sparkplug Working Group, "Sparkplug 3.0.0: Sparkplug Specification," [Online]. Available: https://sparkplug.eclipse.org/specification/version/3.0/documents/sparkplug-specification-3.0.0.pdf

[11] Eclipse Sparkplug Working Group, *Sparkplug Compatible Program.* [Online]. Available: https://sparkplug.eclipse.org/compatibility/ (accessed: Sep. 5 2024).

[12] Apache Software Foundation, *Apache Kafka.* [Online]. Available: https://kafka.apache.org/ (accessed: Sep. 5 2024).

[13] Apache Software Foundation, *Apache Kafka: Powered By.* [Online]. Available: https://kafka.apache.org/powered-by (accessed: Sep. 5 2024).

[14] Confluent, Inc, *Event-Driven Architecture (EDA): A complete introduction.* [Online]. Available: https://www.confluent.io/learn/event-driven-architecture/ (accessed: Sep. 6 2024).

[15] Berktorun, *Understanding Apache Kafka: From LinkedIn's Data Streams to Worldwide Communication.* [Online]. Available: https://medium.com/@berktorun.dev/understanding-apache-kafka-from-linkedins-data-streams-to-worldwide-communication-8e199d7140f7 (accessed: Sep. 6 2024).

[16] Confluent, Inc, *Putting Apache Kafka To Use: A Practical Guide to Building an Event Streaming Platform (Part 1).* [Online]. Available: https://www.confluent.io/blog/event-streaming-platform-1/ (accessed: Sep. 6 2024).

[17] Docker, Inc, *Docker Hub Container Image Library.* [Online]. Available: https://hub.docker.com/ (accessed: Sep. 5 2024).

[18] PTC, *KEPserverEX OT Connectivity Platform: What is KEPServerEX?* [Online]. Available: https://www.ptc.com/en/products/kepware/kepserverex# (accessed: Sep. 5 2024).

[19] PTC, *Download the KEPserverEX demo: About the KEPServerEX demo.* [Online]. Available: https://www.ptc.com/en/products/kepware/kepserverex/demo-download (accessed: Sep. 5 2024).

[20] Confluent, Inc, *Confluent Pricing.* [Online]. Available: https://www.confluent.io/confluent-cloud/pricing/?pillar=stream (accessed: Sep. 6 2024).

[21] ThingsBoard, Inc, *ThingsBoard - Open-sorce IoT Platform.* [Online]. Available: https://thingsboard.io/ (accessed: Sep. 5 2024).

[22] J. Theocharis, *The Unified Namespace as the strongest architectural proposal for Industry 4.0.* [Online]. Available: https://learn.umh.app/blog/the-unified-namespace-as-the-strongest-architectural-proposal-for-industry-4-0/ (accessed: Sep. 6 2024).

[23] Reynolds, Walker, *What Is The Unified Namespace (UNS)?* [Online]. Available: https://www.linkedin.com/feed/update/urn:li:activity:7191867463301537793/ (accessed: Sep. 5 2024).

[24] HighByte, Inc, *Unified Namespace | HighByte Intelligence Hub.* [Online]. Available: https://www.highbyte.com/intelligence-hub/unified-namespace

[25] UMH Systems GmbH, *Unified Namespace (UNS) | IT/OT integration platform.* [Online]. Available: https://www.umh.app/solutions/unified-namespace (accessed: Sep. 5 2024).

[26] Real Time Automation, Inc., *What is a Unified Namespace? (UNS).* [Online]. Available: https://www.rtautomation.com/rtas-blog/what-is-unified-namespace/ (accessed: Sep. 5 2024).

[27] P. DeBeasi and Gartner Research, “Reference Architecture for Integrating OT and Modern IT,” 2023. [Online]. Available: https://www.gartner.com/en/documents/4586599

[28] Berg, Christian, *Unified Namespace: What is UNS?* [Online]. Available: https://www.clarify.io/learn/unified-namespace-uns (accessed: Sep. 5 2024).

[29] D. DuFresne, *My Ongoing Journey to Discover the Unified Namespace.* [Online]. Available: https://www.linkedin.com/pulse/my-ongoing-journey-discover-unified-namespace-dylan-dufresne-eutcc/ (accessed: Sep. 5 2024).

[30] P. Larochelle, *MQTT Sparkplug B or Not to B? Unified Namespace Architecture.* [Online]. Available: https://neomatrixinc.com/blog/sparkplug-b-or-not-to-b-unified-namespace-architecture-considerations/ (accessed: Sep. 5 2024).

[31] K. Manditereza, *Implementing Unified Namespace (UNS) With MQTT Sparkplug.* [Online]. Available: https://www.hivemq.com/blog/implementing-unified-namespace-uns-mqtt-sparkplug/ (accessed: Sep. 5 2024).

[32] CORSO SYSTEMS, *Combining the Power of ISA-95 and Sparkplug B.* [Online]. Available: https://corsosystems.com/posts/using-isa-95-and-sparkplug-b-together (accessed: Sep. 5 2024).

[33] J. Theocharis, *Data Modeling in the Unified Namespace: From Topic Hierachies over Payload Schemas to MQTT/Kafka*.

[34] J. Knepper. [Online]. Available: https://t.ly/CrEwY (accessed: Sep. 5 2024).

[35] EMQ Technologies, *MQTT with Kafka: Supercharging IoT Data Integration.* [Online]. Available: https://emqx.medium.com/mqtt-with-kafka-supercharging-iot-data-integration-d2fc3074ecc5 (accessed: Sep. 5 2024).

[36] Kaggle, *Kaggle: Your Machine Lerning an Data Science community.* [Online]. Available: https://www.kaggle.com/

[37] *BATADAL: Cyber Attacks Detection in Water Systems.* [Online]. Available: https://www.kaggle.com/datasets/minhbtnguyen/batadal-a-dataset-for-cyber-attack-detection (accessed: May 9 2024).

[38] R. Taormina *et al.,* “Battle of the Attack Detection Algorithms: Disclosing Cyber Attacks on Water Distribution Networks,” *J. Water Resour. Plann. Manage.*, vol. 144, no. 8, 2018, doi: 10.1061/(ASCE)WR.1943-5452.0000969.

[39] Github, *opcua-asyncio: OPC UA library for python >= 3.7.* Accessed: Sep. 5 2024. [Online]. Available: https://github.com/FreeOpcUa/opcua-asyncio

[40] Confluent, Inc and Github, *docker-compose.yml files for Apache Kafka Confluent Platform.* [Online]. Available: https://github.com/confluentinc/cp-all-in-one/tree/7.5.0-post (accessed: Sep. 5 2024).

[41] Cirrus Link Solutions, *sparkplug_b.proto file.* [Online]. Available: https://github.com/eclipse/tahu/blob/master/sparkplug_b/sparkplug_b.proto (accessed: Sep. 5 2024).

[42] ThingsBoard, Inc, *ThingsBoard Professional Edition installation options.* [Online]. Available: https://thingsboard.io/docs/user-guide/install/pe/installation-options/ (accessed: Sep. 5 2024).