
# PolyglIoT

ISEP

2023 / 2024

**1210828 Ricardo Venâncio**

# PolyglIoT

ISEP

2023 / 2024

**1210828 Ricardo Venâncio**

## Degree in Informatics Engineering

June 2024

ISEP Advisor: **António Manuel de Sousa Barros**

Co-supervisor: **Luis Lino Ferreira**

*« A journey with a thousand miles begins with a simple step. »*

*Lao Tzu*

# Acknowledgements

I thank all the family members, friends, and colleagues who, in some way, supported me, directly or indirectly, in the completion of this work. Thank you to Professors António Barros and Ricardo Severino for their guidance, availability, and direction provided throughout the development of this project. Thanks to Bernardo Amaral Cabral, Filipe Sousa Duarte, Luís Filipe Ribeiro and Tiago Caló Fonseca for being always available to help me during the project's development. Finally, I would like to express my gratitude to Professor Luís Lino Ferreira for providing me with profound insights and essential advice. His mentorship has not only contributed significantly to the quality of this work but has also inspired me to strive for more in my academic endeavor.

This project was supported by the Arrowhead project. In collaboration with Edge4CPS, PolyglIoT was developed for the Arrowhead framework as an automatic multi-protocol translator.

# Abstract

The increasing number of IoT deployment scenarios and applications fostered the development of a multitude of specially crafted communication solutions, several proprietary, which are erecting barriers to IoT interoperability, impairing their pervasiveness. To address such problems, several middleware solutions exist to standardize IoT communications, hence promoting and facilitating interoperability. Although being increasingly adopted in most IoT systems, it became clear that there was no "one size fits all" solution that could address the multiple Quality-of-Service heterogeneous IoT systems may impose. Consequently, we witness new interoperability challenges regarding the usage of diverse middlewares. In this work, we address this issue by proposing a novel architecture – PolyglIoT – that can effectively interconnect diverse middleware solutions while considering the QoS requirements alongside the proposed translation. The performance and robustness of the solution are analyzed, demonstrating that this Multiprotocol Translator is feasible and can achieve high performance, thus becoming a fundamental component for enabling future highly heterogeneous IoT systems of systems.

**Keywords (Theme):** IoT, Middleware, Translator

**Keywords (Technologies):** Arrowhead, MQTT, Kafka, AMQP, DDS

# Table of Contents

# Table of Figures

# Table of Tables

# Table of Code Snippets

# Notation and Glossary

| | |
|---|---|
| **AMQP** | Advanced Message Queuing Protocol |
| **API** | Application Programming Interface |
| **DDS** | Data Distribution Service |
| **Framework** | Structured environment for development |
| **IoT** | Internet of Things |
| **IP** | Internet Protocol |
| **JDK** | Java Developer Kit |
| **JNI** | Java Native Interface |
| **JSON** | JavaScript Object Notation |
| **JVM** | Java Virtual Machine |
| **M2M** | Machine-To-Machine |
| **MQTT** | Message Queuing Telemetry Transport |
| **OOP** | Object-Oriented Programming |
| **OpenDDS** | DDS Implementation |
| **QoS** | Quality of Service |
| **RabbitMQ** | AMQP implementation |
| **SOA** | Service-Oriented Architecture |
| **TCP** | Transmission Control Protocol |
| **UDP** | User Datagram Protocol |
| **UML** | Unified Modeling Language |

# 1 Introduction

This chapter provides a concise overview of the primary concepts discussed in this report, including the problem the project addresses and the technologies employed to solve it.

## 1.1 Context

The Internet of Things (IoT) refers to a network of physical devices, vehicles, appliances, and other objects that are embedded with sensors, software, and network connectivity, allowing them to collect and share data. The potential applications of IoT are vast and varied, and its impact is already being felt across a wide range of industries. As the number of internet-connected devices continues to grow, IoT is likely to play an increasingly important role in shaping our world, transforming the way that we live, work and interact with each other (Oracle, “What is IoT?”).

Several communication protocols, such as MQTT, Kafka, DDS, and AMQP, have been developed and are used for the Internet of Things. These protocols were originally devised and proposed for specific scenarios or use cases. For example, MQTT is most suitable for environments with limited network bandwidth or other resource constraints (MQTT, “The Standard for IoT Messaging”). Kafka is ideal for handling high-volume, high-throughput, and low-latency messaging (Apache Kafka, “Documentation”). DDS excels in high-performance, real-time, scalable, and resilient systems (DDS Foundation, “Hoes does DDS Work?”), while AMQP is preferred for reliable and secure messaging (RabbitMQ, “RabbitMQ Documentation”).

According to N. Naik (2017), the choice of an effective messaging protocol for IoT is quite complex. The suitability of a particular protocol is determined by the specific requirements and context in which it will be used. Different applications and scenarios have unique needs and constraints, which dictates the use of varied protocols. No single protocol is universally applicable or optimal for all possible use cases due to the diversity in functionality, performance demands, security considerations and other factors. Consequently, the choice of protocol must be carefully evaluated to meet the specific needs of each individual use case.

Let’s assume a system that encompasses various IT levels, each with distinct characteristics and different use cases. As an example, MQTT is more efficient in handling IoT applications, but Kafka is more adequate for IT applications. In each scenario, a distinct protocol is employed that aligns optimally with the specific conditions and requirements, a decision

characterized by rationality. However, for the different parts of the system to communicate with each other, it will be necessary to translate between the different protocols in use, allowing data to flow between the parts of the system and enabling the information to be used more effectively between the various levels. To address this common problem, PolyglIoT was developed as middleware to facilitate transparent communications between different protocols. This allows the strengths of each protocol to be used without concern for communication limitations. PolyglIoT currently supports multiple translations between the following protocols: MQTT, AMQP, Kafka and DDS. The term “multiple translations” is used to describe the capacity to translate, simultaneously, different protocols with different Quality of Service.

The term “Quality of Service" (QoS) refers to the establishment of a mutual understanding between the entities involved in message transmission, identifying the assured degree of delivery reliability for a given message (HiveMQ Team, “What is MQTT Quality of Service (QoS) 0,1 & 2? – MQTT Essentials: Part 6”). Therefore, PolyglIoT addresses the differences between protocol QoS properties, mapping them to assure three levels of quality: (1) At Most Once, (2) At Least Once and (3) ImpExactly Once.

This project is framed within the Arrowhead fPVN project and with the Eclipse Arrowhead Framework [https://arrowhead.eu/eclipse-arrowhead-2/].

## 1.2 Objectives

The overarching objective of this project is to refine and expand the capabilities of the existing system, with a focus on enhancing interoperability, scalability and performance. This involves a multi-faceted approach, as defined in the following detailed goals:

1. Enhance the code quality and documentation of the existing implementation by addressing the lack of essential documentation, eliminating code redundancy, and improving readability.
2. Redefine QoS mapping between the supported protocols due to gaps from the previously implemented protocols and the necessity from the new one.
3. Expand the application to support the DDS protocol, specifically, production and consumption between every supported protocol and DDS.
4. Integrate PolyglIoT with the Arrowhead framework and Edge4CPS to scale PolyglIoT’s usage.
5. Evaluate and improve the system’s performance to assess its efficiency.

**Commented [AB1]:** Não percebo esta frase.

**Commented [R(2R1]:** Aqui, quero dizer que um objetivo é redefinir o mapeamento QoS, ou seja, a criação da Table 4.

**Commented [AB3]:** Não percebo esta frase.

**Commented [R(4R3]:** Aqui está mal escrito, era suposto ser só "Evaluate and improve the system's perfomance"

## 1.3 Approach

The approach articulates the specific actions and methodologies employed to achieve the designated goals. To achieve the above-mentioned objectives, the following tasks were performed:

- Study of the P. S. Costa (2023) report for a better understanding of how everything works and the possible problems.
- Research about all implemented communication protocols (Kafka, AMQP and MQTT) and the new one (DDS).
- Plan and study how to ensure QoS levels for each communication protocol.
- Research OpenDDS framework and Java Native Interface (JNI).
- Research the Arrowhead framework.
- Test on a dedicated network, using three different devices, to measure PolyglIoT performance and resource consumption.

Commented [AB5]: Falta aqui uma menção ao relatório, que deve constar na bibliografia.

Commented [R(6R5]: Entendido, vou colocar agora mesmo e citar.

## 1.4 Contribuition

The main result of this work is an improved implementation of the PolyglIoT protocol translator. This improved version provides several innovative contributions.

1. PolyglIoT application, capable of performing translation between different communication protocols, with a more refined mapping of the QoS delivery levels.
2. Added DDS support to PolyglIoT.
3. Communication protocol translation service (PolyglIoT) added to the Arrowhead framework that is open for use by other developers.
4. Open-source solution allowing developers to implement the system.

Commented [AB7]: Não percebo esta frase.

Commented [R(8R7]: Como se trata de um repo open-source, quem quiser contribuir (por ex, adicionar suporte para um novo protocolo) ou até mesmo utilizar o sistema, tem essa possibilidade.

## 1.5 Work Planning

This project was structured into five stages: (1) Research, (2) Analysis and Design, (3) Implementation, (4) Testing, and (5) Integration with Edge4CPS within the Arrowhead framework. This work plan is illustrated in Figure 1.

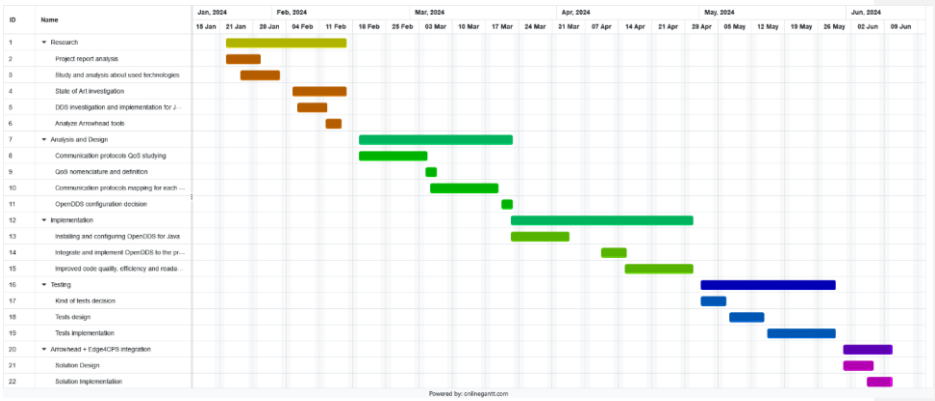


*Figure 1 – Project Schedule*

The first phase comprised the study of the previous project report to fully understand the problem addressed and the design and implementation of the previous protocol translator, and conclusions. A review of the current state-of-the-art was carried out, focusing on relevant technologies and related works. This included an exhaustive examination of the DDS communication protocol and its implementation in Java, given the constraints imposed by both the programming language and the protocol. Furthermore, a comprehensive review of the Arrowhead documentation was conducted.

In the analysis and design phase, we established the mapping between the QoS levels of the supported communication protocols. In this phase, we discussed and defined the architecture of the application, considering the requirements and constraints of the system.

The implementation phase involved the implementation and inclusion of the DDS protocol: including the configuration of the DDS technology and development of the producer and consumer code. A second task was to update the application to adopt the redefined QoS mapping. Additionally, a global refinement of source the existing source code improved readability and maintainability of PolyglIoT.

The testing phase encompassed the design and subsequent implementation of tests.

Finally, in the integration phase, PolyglIoT was integrated with Edge4CPS within the Arrowhead environment.

### 1.5.1 Meetings

Meetings were held presential in Instituto Superior de Engenharia do Porto (ISEP). These meetings were used to plan the project, discuss design, and demonstrate the progress made

throughout the weeks. Additionally, Microsoft Teams were employed to manage PolyglIoT's related files.

## 1.6 Report Structure

The report is structured into diverse chapters. Chapter 2 (State of the Art) reviews the most relevant works in the problem domain, providing a critical overview of the technologies used. Chapter 3 (Analysis) covers the analysis of functional and non-functional requirements, problem interpretation, domain model, and use cases. Chapter 4 (Design) presents a detailed explanation of the solution's design, including diagrams following the C4 model. Chapter 5 (Implementation) describes the practical aspects of the solution, detailing how the design was implemented. Chapter 6 (Tests) demonstrate the testing phase, providing evidence that the implementation runs as expected. Finally, Chapter 7 (Conclusion) reflects on the project's future development, comparing the planned goals with the achieved outcomes.

# 2 State of the Art

This chapter presents the research of analogous projects, definitions, and the usability of used technologies. Additionally, it examines the relationship between this project and other similar ones and the previous work.

## 2.1 Previous Work

This work is an evolution of an application still in a preliminary version, that performed translation between protocols AMQP, Kafka and MQTT, without fully functional QoS capabilities. QoS implementation contained some flaws and there was no integration with the Arrowhead framework and Edge4CPS (both Arrowhead framework and Edge4CPS technologies are explained in the next section).

## 2.2 Existing Technologies

Communication protocols share a common objective: communication. However, they exhibit notable distinctions in terms of their usability, applicability, implementation, design, and many other specifications. According to N. Lukman (2021) and EMXQ Team (2024), AMQP, MQTT, DDS and Kafka are predominant communication protocols in IoT, and the following sections elucidate the principles of each protocol.

### 2.2.1 AMQP

AMQP protocol uses an “Exchange/Queue” type of messaging (Figure 2), instead of publishers and subscribers. Messages are published to Exchanges, which then decides by its Routing Key (the topic), to redirect the message to Queues from which consumers can read directly. These queues can store messages indefinitely, until a consumer has finally read them. (RabbitMQ, ”RabbitMQ Documentation”).

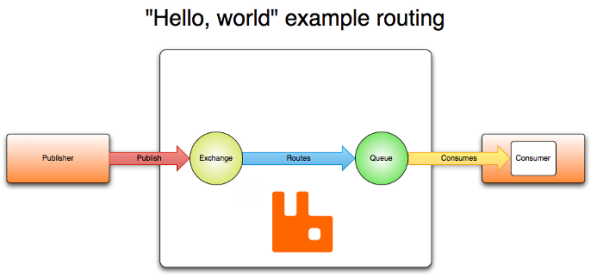


*Figure 2 – AMQP Routing Scheme (https://www.rabbitmq.com/tutorials/amqp-concepts)*

### 2.2.2 MQTT

MQTT is a broker-based publish/subscribe messaging protocol (Figure 3) with low transport overhead, and a small code footprint, reducing network traffic and improving its efficiency, making it a popular choice in less powerful network or devices. Messages are published to topics managed by a broker, which are then redirected to subscribers on the other end. MQTT brokers allow their clients to connect with persistent sessions, so that in the event of a connection terminating because of an unreliable network connection, clients can resume their previous session, reducing the time for reconnections, and enabling reception of queued messages for consumers. (MQTT).

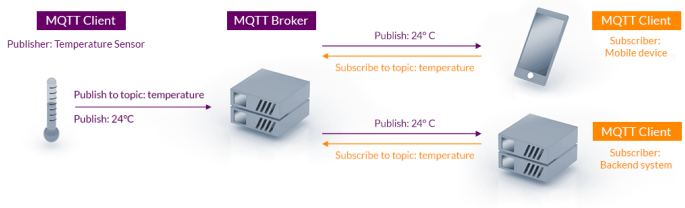


*Figure 3 – MQTT Architecture (https://mqtt.org/)*

### 2.2.3 Kafka

Kafka is a scalable, durable, and fault-tolerant protocol based on the publish-subscribe paradigm, built to handle substantial amounts of data (Figure 4). Topics in Kafka are organized into partitions, which provide load balancing by splitting topics into multiple smaller divisions, which can be handled by less busy brokers. Fault tolerance is also ensured by replicating each of these partitions across a cluster, meaning that in the event of one broker failing, the others can still handle clients communicating with that topic and ensure no message is lost. (Apache Kafka, “Documentation”).

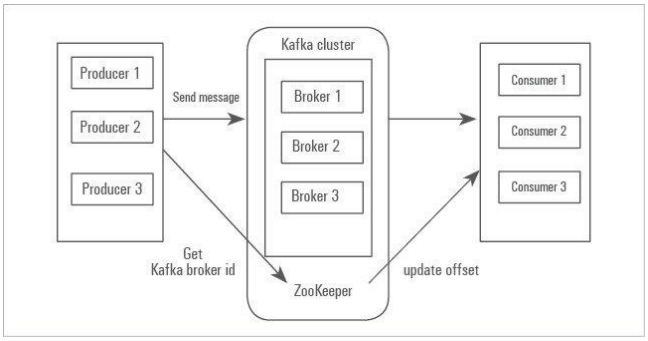


*Figure 4 – Kafka Architecture (https://waytoeasylearn.com/learn/kafka-architecture/)*

### 2.2.4 DDS

Data Distribution Service (DDS) is a middleware standard that enables scalable, real-time, high-performance data distribution and integration in distributed systems (Figure 5). It focuses on facilitating efficient and reliable data exchange among various nodes without centralized control, unlike MQTT or Kafka. DDS is ideal for applications that require low latency, high throughput, and deterministic communication, such as Industrial IoT and real-time control systems. It aims at simplifying data distribution and provides a standardized communication model, streamlining the development of distributed applications and ensuring interoperability and robustness across diverse environments. It uses a publish/subscribe communication model, where data is distributed based on the content itself, there is not any broker and data exchange is directly between the producer and consumer. There are built-in mechanisms for reliable data delivery with configurable quality of service (QoS) policies to ensure message delivery and integrity. It lacks widespread adoption and ecosystem maturity compared to MQTT and Kafka, meaning that DDS is not used/supported as often as these other protocols. (DDS Foundation, “Hoes does DDS Work?”).

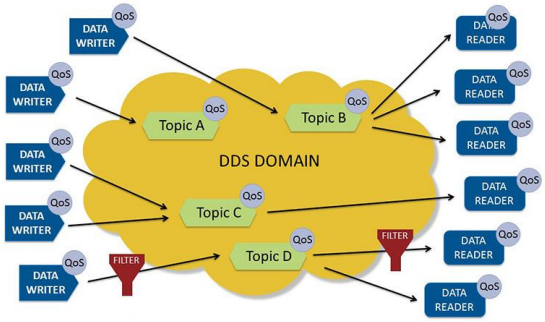


*Figure 5 – DDS Architecture (https://www.dds-foundation.org/what-is-dds-3/)*

### 2.2.5 Edge4CPS

Edge4CPS is an open-source system for deploying applications across multiple architectures. It provides pre-configured services such as protocol brokers (Kafka and MQTT), a protocol translator, an API, and a UI to simplify Kubernetes deployment, making it accessible to less experienced users (Figure 6). The system is also designed to contribute to projects of varying sizes, supporting a wide range of deployment requirements (B. Cabral, 2023).

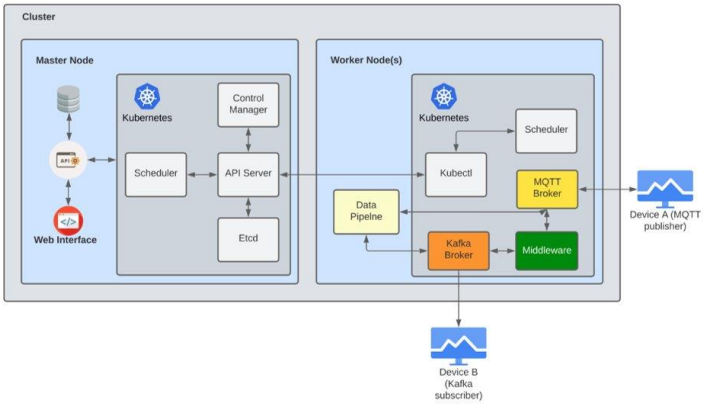


*Figure 6 – Edge4CPS Architecture (B. Cabral, 2023)*

### 2.2.6 Arrowhead

Based on (Eclipse Arrowhead," A framework and implementation platform for SoS, IoT and OT integration"), Eclipse Arrowhead provides a framework and platform for creating automation and digitalization solutions (Figure 7). It leverages microservice and microsystem architecture based on service-oriented architecture principles. The platform includes an engineering process, various tools, and core microsystems. It offers proven core microsystems, libraries, template code for application systems, and supports model-based engineering with SysML and UML.

This framework is particularly suitable for designing, implementing, and deploying solutions that align with Industry 4.0 architectures such as Rami4.0. It offers extensive interoperability support for use in highly heterogeneous environments, including autonomous protocol translators and adapters for numerous legacy and IT technologies like OPC-UA, Modbus TCP, Z-wave, IO-link, and Web of Things.

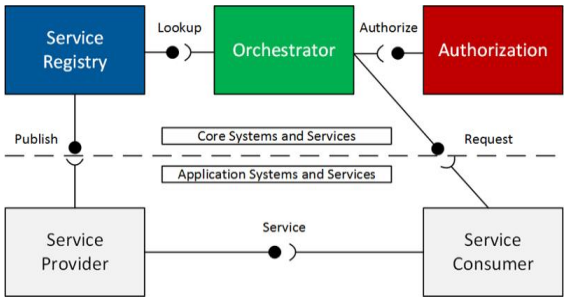


*Figure 7 – Arrowhead Framework Architecture*

*(https://arrowhead.eu/technology/architecture/)*

The use of the Arrowhead framework allows for increased flexibility, scalability, and availability, allowing for a consumer to request any device information on a specific middleware, by issuing a request to the multi-protocol translation service, all managed by Eclipse Arrowhead.

### 2.2.7 Protocols Comparison

While AMQP is the only protocol that uses the "Exchange/Queue" messaging model among the supported protocols, every other protocol uses the "Pub/Sub" model. In AMQP, messages are published to exchanges, which use routing keys to redirect messages to specific queues. In contrast, in MQTT, messages are published to topics managed by a broker. Kafka is similar to MQTT, but topics are divided into partitions, which enables load balancing and replication across a cluster for fault tolerance. DDS's consumers and producers exchange data directly based on content with no broker. AMQP is well-suited to applications that require message storage and selective message routing. MQTT is an optimal choice for less powerful networks or devices, as well as scenarios with unreliable network connections, Kafka is an excellent fit for high-throughput and scalable data processing applications. Finally, DDS is an ideal choice for applications requiring deterministic communication, such as real-time control systems.

Table 1 presents the main aspects of each communication protocol previously approached.

*Table 1 – Communication protocols main aspects*

| | **Protocols** | | | |
|---|---|---|---|---|
| | AMQP | DDS | Kafka | MQTT |
| **Architecture** | Exchange/Queue | Pub/Sub | Pub/Sub | Pub/Sub |
| **Broker** | Yes | No | Yes | Yes |

| | | | | |
|---|---|---|---|---|
| **Fault Tolerance** | High | High (with redundance) | High (with redundance) | Moderate |
| **Usage** | Message storage, selective routing | Real-time, high-performance data distribution | Scalable data processing applications, high-throughput | Unreliable network connections, less powerful network/device |

## 2.3 Related Works

IoT/CPS systems are composed of devices connected to the internet. They are characterized, above all, by the production of data related to the environment in which they are involved. This data is then processed and analyzed to provide valuable insights, enabling more efficient and effective operations within that environment.

Different scenarios, with different characteristics, have led to the emergence of various communication protocols. Each protocol has its own specifications, effective for different scenarios with different requirements and impositions. MQTT is a protocol employed in a variety of contexts, including smart homes, manufacturing and automotive (MQTT, "The Standard for IoT Messaging"). Kafka is used in log aggregation, monitoring, stream processing and analytics (M. Khatri, 2023). AMQP is employed in telecommunications and financial services (RabbitMQ, "AMQP 0-9-1 Model Explained"). DDS is used in automotive, healthcare, aerospace, and defense (DDS Foundation, "User Experiences"). However, there are instances where systems approach multiple scenarios, such as Smart Cities and Industry 4.0 (I. Jawhar .et. al, 2018). There are several solutions that try to address multiprotocol translations which will be discussed in the following subchapter.

### 2.3.1 Protocol Converters

Protocol converters are software tools that facilitate the transformation and movement of data between different systems, often using different protocols or formats (M. Rouse, 2017).

With so many possibilities, a challenge arises when a system needs to interconnect producers and consumers using different protocols. Several solutions have been proposed to address this issue, including protocol plugins or adapters such as Apache Kafka Connect, which facilitate building connectors between Kafka and various external systems such as MQTT,

thereby enabling a seamless data flow between the two protocols. Stream processing frameworks, such as Apache Flink, Apache Samza, and Apache Spark Streaming offer APIs and libraries for ingesting data from different brokers, processing it, and forwarding it to Kafka topics or vice versa. However, these stream processing frameworks are often complex to set up and configure, especially for tasks such as protocol translation, which can result in significant delays. Additionally, many impose vendor or technology lock-in. For instance, Microsoft's IoT Hub provides protocol translation and a robust architecture with security in mind, however it cannot automatically redirect any MQTT message to a Kafka broker or quickly modify translation logic or support a new protocol by directly altering the code.

Given this scenario, developers often turn to custom application integration for protocol translations. This approach allows for fine-grained control over the translation process and customization based on specific use cases. However, it comes at the cost of higher development expenses and reduced flexibility. This approach was recently followed in (P. H. M. Pereira, 2023), aiming to mitigate interoperability issues by combining standard ontologies and standardized digital representation, but it resulted in a rigid MQTT and OPC UA gateway, validated only via simulation without support for flexible translation services. Alternatively, C. -H. Lee .et. al (2016) proposed an IoT framework based on a software-defined network (SDN) that intercepts packets from CoAP to MQTT and vice versa. The SDN redirects the packets to a proxy for translation upon configuration, but no evaluation results were provided. In another approach, H. Derhamy (2015) introduced a protocol translator leveraging an SOA approach with a direct hub-to-spoke method, supporting HTTP and CoAP. However, no implementation results were provided, and QoS conversion issues, although mentioned, were not addressed. The work in N. Ahmed (2021) attempted to mimic a protocol translation architecture based on H. Derhamy (2015), supporting MQTT, HTTP, and CoAP, with additional syntactic interoperability using semantic sensor network ontology. However, by not relying on the SOA architecture provided by Arrowhead, the resulting solution became centralized, impairing its flexibility.

Although these solutions are relevant, none completely addresses the problem at hand. Notably, none considers the challenge of matching different QoS delivery requirements, or the protocol extensibility provided by PolyglIoT.

# 3 Analysis

This chapter presents a comprehensive analysis of both functional and non-functional requirements. It offers a clear interpretation, providing a clear understanding of the issues at hand. Additionally, it includes the development of two domain models that define the system's insertion in the real world. The chapter also presents detailed use cases that illustrate how the “system” will interact with the PolyglIoT and the various scenarios that may arise during its operation.

## 3.1 Problem Domain

This project addresses the rise of various communication protocols and the significant complexity in choosing the most appropriate one. PolyglIoT functions as a middleware, just like a program that acts as an intermediary to establish communication between different communication protocols. There are several concepts approached and it is noteworthy to say that PolyglIoT works on two distinct scenarios, standalone or integrated with Edge4CPS and Arrowhead framework.

Broker is an intermediary entity that allows clients to communicate through production and consumption, and registers itself on Arrowhead framework. External producers/consumers are the user devices. For instance, an external producer might be a temperature sensor transmitting meteorology information to a MQTT broker and the external consumer might be a computer that is consuming and processing that data to a database, through Kafka. Internal producers/consumers are the ones responsible for consuming data from the external producer (internal consumer), translating it, and producing it for the external consumers (internal producer). Arrowhead is a SOA framework that allows the automation of this process without adding any complexity. The user (external producers/consumers) requests Arrowhead to produce/consume data to/from a subject, and Arrowhead redirects this request and configures PolyglIoT to be able to translate it.

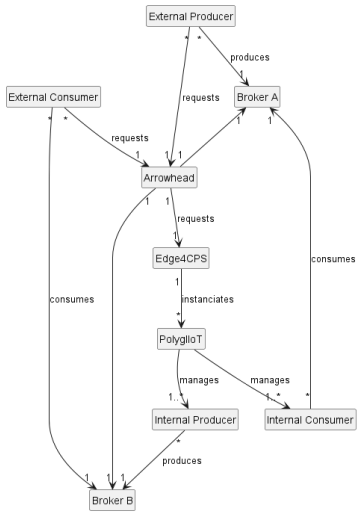


*Figure 8 – Domain Model with Arrowhead and Edge4CPS*

An external producer produces only for a single broker, while the same broker receives production from many external producers. This same dynamic applies to the external consumer, internal producer and internal consumer (along with their respective brokers). PolygIIoT creates and manages at least one internal producer and consumer, both are only managed by the same PolygIIoT. Edge4CPS is responsible for instantiating one or more PolygIIoT's when necessary. Arrowhead receives requests from external producers (to register itself as producer or request broker's information) and external consumers (to search for a producer match or request broker's information). It also communicates with Edge4CPS to request the PolygIIoT initialization regarding the translation's requirements.

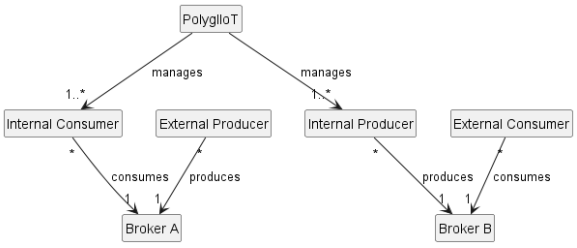


*Figure 9 – Domain Model without Arrowhead and Edge4CPS*

Figure 9 depicts the domain model without Arrowhead and Edge4CPS integration. This configuration is simpler, and it works similarly. However, external producers/consumers do

not request for publishes/consumptions. Instead, the configuration files must be set up to the desired kind of translation.

*Table 2 – Glossary*

| **Term** | **Definition** |
|---|---|
| Arrowhead | A Service-Oriented Architecture (SOA) framework that automates the process of communication between external producers and consumers without adding complexity. It orchestrates requests and configures PolyglIoT for message translation. |
| Broker | An intermediary entity that allows clients to communicate through production and consumption. |
| Consumer | An entity that receives and processes data. There are external consumers (e.g., a computer processing data from a broker) and internal consumers that consume data from external producers for translation purposes. |
| Edge4CPS | A component responsible for instantiating one or more instances of PolyglIoT as needed. It works based on the requests from Arrowhead to configure and initialize PolyglIoT. |
| Producer | An entity that generates and sends data. There are external producers (e.g., a temperature sensor transmitting data to a broker) and internal producers that translate data for external consumers. |

## 3.2 Requirements

In this section, functional and non-functional system requirements will be addressed in detail. Specific features and functionalities will be described by use cases. The actor involved will be explained, identified and defined, detailing his role and interactions within the system.

### 3.2.1 Functional Requirements

This type of requirement (QRA Corp, "Functional vs. Non-functional Requirements."), defines the specific behaviors, tasks, and operations a software system must perform to meet user needs. These requirements clarify the system's intended functionality, detailing expected input, processing, and output mechanisms.

**FR 1.** The system must support simultaneous translation between multiple messaging protocols, including:

- MQTT.
- Kafka.
- AMQP.
- DDS.

**FR 2.** The system must be able to specify a QoS level for each producer/consumer regarding the possible values:

- At Most Once.
- At Least Once.
- Exactly Once.

**FR 3.** The system must be able to specify which consumers consume from each producer.

**FR 4.** The system must be able to request a broker in Arrowhead framework.

**FR 5.** The system must be able to request to register itself as a producer in Arrowhead framework.

**FR 6.** The system must be able to request to find, automatically, a producer match interested in the same subject in Arrowhead framework.

### 3.2.2 Non-Functional Requirements

This type of requirement defines how a system should behave and imposes constraints on the system’s functionality (M. Glinz, 2007).

Performance in a software system refers to how well the system executes tasks under specific conditions. This includes aspects such as response time, throughput, and resource utilization. High performance means the system can handle tasks efficiently without significant delays. Scalability is the ability of a system to handle an increasing amount of work or its potential to accommodate growth. Interoperability is the ability of different systems, applications, or components to communicate, exchange data, and use the information that has been exchanged effectively (H. Akhtar, 2023).

The following list, enumerates the identified non-functional requirements:

**NFR1. Performance -** The application translation must be fast, efficient (apply parallelism concepts and low complexity algorithms) and effective (capable of reach its purpose).

**NFR2. Scalability -** The application must be scalable and extensible for eventual new features/implementations.

**NFR3. Interoperability -** The application must integrate all different technologies (OpenDDS, MQTT, Kafka and RabbitMQ) across the project.

## 3.3 Actors

This project is situated within the context of the Internet of Things (IoT), with communication occurring in a machine-to-machine (M2M) environment. Machine-to-machine communication refers to the direct exchange of information between devices without human intervention. This technology forms the backbone of the Internet of Things (IoT), enabling various devices to communicate and work together seamlessly (A. Feiszli, 2023).

The only actor involved is defined as the system. The system must specify what kind of translation it wants to apply, the related QoS deliveries as well as brokers setup and configurations.

## 3.4 Use Cases

A use case defines the objective that the primary actor aims to achieve with the system's defined responsibilities. It encompasses a series of scenarios involving interactions between the system in question and different actors, illustrating the potential outcomes where the primary actor's goal could either be fulfilled or not (A. Cockburn, 2000).

Regarding the system requirements, the following use cases were created:

*Table 3 – Use Cases*

| Use Cases | Acceptance Criteria |
|---|---|
| Produce and specify the respective QoS delivery. | Messages must be, automatically, translated between different protocols. QoS levels must be either:<br>- At Most Once.<br>- At Least Once.<br>- Exactly Once. |
| Consume and specify the respective QoS delivery. | Messages must be, automatically, translated between different protocols. QoS levels must be either:<br>- At Most Once.<br>- At Least Once.<br>- Exactly Once. |
| Register as a producer. | The producer register request must register this producer in Arrowhead framework with the “Publisher” service definition. |
| Requests to consume for a specific topic and, automatically, find a publisher match. | The publisher and the consumer must be interested in producing/consuming to/from the same topic. Arrowhead must request Edge4CPS to instantiate PolyglIoT with the respective settings. |

# 4 Design

Based on the previous chapter, design depicts every decision, why and how they have been taken. Based on the insights and analyses presented in the Analysis chapter, the Design chapter meticulously defines every decision made during the design process. It explains the rationale behind each decision, including the specific requirements and constraints that influenced these choices. Additionally, the chapter provides a detailed description of how each decision was implemented, illustrating the steps taken to achieve the desired outcomes.

## 4.1 Architectural Design

The architecture follows the C4 model pattern (A. Vázquez-Ingelmo et. al., 2020).

PolyglIoT consists of a middleware translator for communication protocols, regarding different QoS levels. Figure 10 colors represent:

- Blue: external producers (user devices, external to PolyglIoT).
- Orange: internal consumers and producers,
- Green: broker or exchange shared by the external producers/consumers and internal consumers/producers, respectively (only brokers are going to be mentioned, however it refers to exchanges in RabbitMQ case).
- Red: external consumers (user devices, external to PolyglIoT).
- Black: thread.

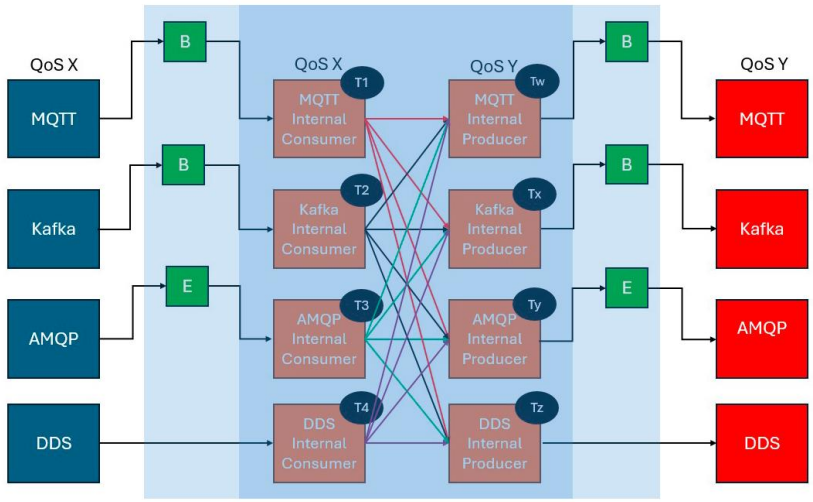


*Figure 10 – PolyglIoT architecture diagram*

There are two rectangles that indicate how much PolyglIoT might scatter. The most transparent rectangle indicates that brokers might be in PolyglIoT's machine, whereas the least transparent rectangle indicates that brokers are external to PolyglIoT. Several associations between internal consumers and producers are depicted, indicating the potential for translations. It should be noted that each protocol can translate for any other protocol. Each internal consumer has a separated thread waiting for data and every time it consumes something, a thread is created for each linked internal producer. For example, a MQTT external producer publishes a message to the shared broker. After the internal MQTT consumer gathers that data, it creates a thread to run the linked internal producer(es) on the required output protocol. It is noteworthy to remember that DDS doesn't have neither a broker nor an exchange due it distributed nature, producer and consumer communicate directly.

### 4.1.1 Quality of Service Handling

Each one of the supported communication protocols has their own specifications and configurations related to QoS delivery. PolyglIoT establishes delivery QoS levels to ensure end-to-end seamless data protocol translation in this regard. PolyglIoT's strategy was to take MQTT and Kafka delivery QoS levels as inspiration, choosing for each protocol a set of properties that could guarantee a similar QoS level. Therefore, the following QoS levels are considered (Confluent, "Message Delivery Guarantees"):

- At Most Once – Also known as fire-and-forget, this level of QoS sends a packet without guarantees of delivery.
- At Least Once – As suggested, it sends at least one packet, ensuring its delivery, but duplications may occur.
- Exactly Once – With the highest reliability, sends exactly one packet, ensuring there is no duplication.

To guarantee each one of the previously listed QoS's, there are specific protocol's properties that need to be activated and configured. Table 4 depicts the mapping for each protocol and QoS, followed by the definition and explanation of each used property.

*Table 4 – Quality of Service Mapping*

| Producer | Consumer | At Most Once | At Least Once | Exactly Once |
|---|---|---|---|---|
| Kafka | MQTT | acks=0, enable_idempontence=false | acks='all', enable_idempotence=false | acks='all', enable_idempotence=true, max_in_flight_requests_per_connections=1 |
| | RabbitMQ | | | |
| | OpenDDS | | | |
| MQTT | Kafka | qos=0, retained=false, clean_session=true | qos=1, retained=true, clean_session=false | qos=2, retained=true, clean_session=false |
| | RabbitMQ | | | |
| | OpenDDS | | | |
| RabbitMQ | MQTT | confirmSelect=false | confirmSelect=true, prefetchCount=10 | confirmSelect=false, prefetchCount=1 |
| | Kafka | | | |
| | OpenDDS | | | |
| OpenDDS | MQTT | reliability.kind=best_effort | reliability.kind=reliability, deadline=5, liveliness.kind=automatic_liveliness_qos | |
| | RabbitMQ | | | |
| | Kafka | | | |

Starting with Kafka, it is considered the following properties:

acks – specifies if delivery confirmations are expected before considering a request complete (if zero, no confirmation is expected). It is enabled in reliable QoS levels.

enable_idempotence – this setting has an effect in the possible duplication of messages. If enabled, one message is received at the most, without duplication (due to overwriting).

max_in_flight_requests_per_connection – the maximum number of unacknowledged requests the client will send on a single connection before blocking. This property is specified only in the Exactly Once QoS level, due to activating idempotence. It is use case specific, and the value is set to 5.

For MQTT is enabled the equivalent QoS levels and ensured the following properties:

qos – Defines the QoS level at which the message should be sent (the QoS levels in MQTT are as defined in PolygIIoT).

retained – Indicates whether the message should be retained by the broker and delivered to new subscribers, it is enabled for higher QoS levels.

clean_session – Indicates whether the session should be cleaned or maintained after a disconnection. It is enabled for higher QoS levels, to minimize impact of network shortages. Upon reestablishing connection, non-delivered data can be received.

For RabbitMQ, the properties used were:

confirmSelect – Activates delivery confirmations by the producer making sure published messages have safely reached the broker. It is activated for reliable QoS levels.

prefetchCount – The maximum number of unacknowledged messages on a channel (unlimited if zero). To make sure messages are being sent exactly once, an auxiliar data structure is used to keep track of acknowledged messages. If not acknowledged, a message is going to be transmitted until successfully sent.

Finally, regarding DDS, it is considered the following properties to assure the QoS levels:

RELIABILITY.kind – By choosing "MAX_EFFORT" or the "RELIABLE" setting, it can be defined whether there will be resources to ensure that information is successfully received.

DEADLINE – Establishes a contract for the time allowed between messages. For consumers it defines the maximum amount of time allowed to pass between receiving messages. For producers it establishes the maximum amount of time allowed between sending messages. Although application specific, it is considered these for higher QoS.

LIVELINESS.kind – For consumers it defines the level of reporting that they require from the producers to which they are subscribed. For publishers it establishes the level of reporting that they will provide to subscribers that they are alive. It is imposed on higher QoS levels.

Importantly, because DDS internally guarantees no duplication of data, it is not possible to map such QoS levels in an equivalent fashion. Instead, when At Least Once level is required, PolyglIoT relies on the Exactly Once QoS level.

### 4.1.2 Logical View

This visualization represents structural elements and relationships within the system, focusing on component interaction. There are represented three granularity levels.

#### 4.1.2.1 Level 1

The most abstract logical view level is represented by Figure 11 – level 1. PolyglIoT provides an interface which is consumed by the Edge4CPS system, for the PolyglIoT's management. From the same port, all communication protocols technologies are consumed by PolyglIoT, to connect to the brokers.

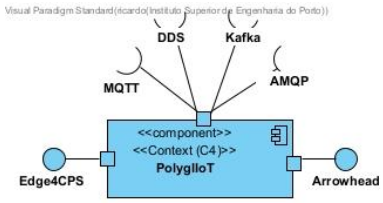


*Figure 11 – Logical View, level 1*

#### 4.1.2.2 Level 2

In the level 2 diagram (Figure 12), PolygIIoT is represented in more detail. The MiddlewareSetup component reads all the setup configuration files and checks connectivity with the Arrowhead core services (only if integrated with it). It also creates internal producers and consumers and establishes the connection between it and the respective brokers.

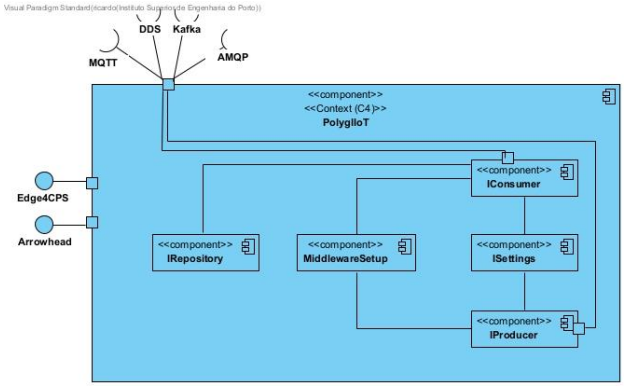


*Figure 12 – Logical View, level 2*

IConsumer component is the main concept for every implemented protocol as a consumer and the same applies to IProducer. Finally, IRepository is the component that helps IConsumer handle the Exactly Once transmissions. It acts as a database persisted in memory to prevent duplication of messages.

#### 4.1.2.3 Level 3

The third level of the logical view is depicted in Figure 13. Here, the production and consumption components for each communication protocol have direct communication with the external components, namely, MQTT, DDS, Kafka and AMQP for the broker consuming and producing. Each communication protocol has its own custom producer and consumer (as

known as internal producer and consumer) that is responsible for producing/consuming data to/from brokers. DDS, RabbitMQ, MQTT and Kafka have their own settings for QoS mapping.

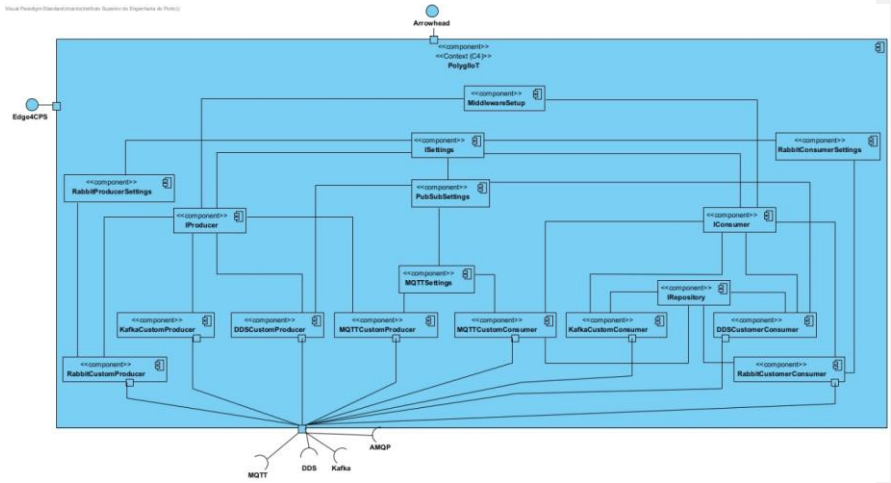


*Figure 13 – Logical View, level 3*

### 4.1.3 Physical View

As known as deployment view depicts how the system is physically distributed and the type of communication between different nodes. Level 2 is the only one represented.

#### 4.1.3.1 Level 2

The PolyglIoT's physical view is depicted in Figure 14. Each container/node represents a different and, physically, separated machine. The associations have an attached label which represents the communication protocol.

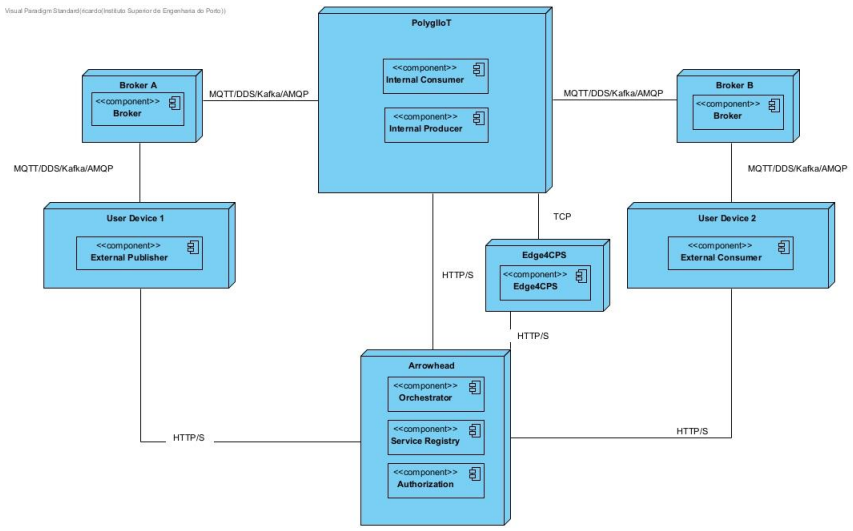


*Figure 14 – Physical View, level 2*

PolyglIoT runs on a dedicated machine and contains the internal producer(s) and consumer(s), it communicates through Kafka, MQTT, DDS or AMQP with nodes “Broker A” and “Broker B” for these internal components consumption and production. Both brokers communicate with the internal and external producers and consumers. Arrowhead communicates through HTTP/S with both user devices for translation requests, broker data requests and publisher service register requests (only publishers). Edge4CPS communicates with PolyglIoT through TCP when it receives a request from Arrowhead to instantiate it.

It is noteworthy to say that this example represents the most separated (and realistic) way. Brokers might be inside PolyglIoT node, or even be apart from PolyglIoT and both “Broker A” and “Broker B”, inside the same node.

### 4.1.4 Implementation View

This view represents how different modules, classes, components, and their interactions come together to form the software solutions. Level 1 is the only representation.

#### 4.1.4.1 Level 1

PolyglIoT is only composed of its system, everything else is external to it (Figure 15).

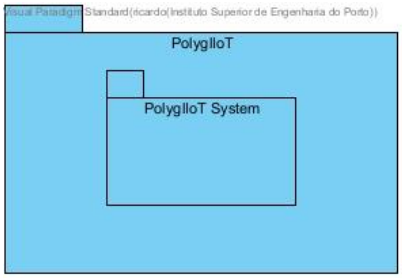


*Figure 15 – Implementation View, level 1*

### 4.1.5 Use cases

The use case diagram is represented by Figure 16. It provides the visual image of the use cases identified. It is noteworthy that the Arrowhead box includes only UC3 and UC4, because UC1 and UC2 are referred to the no integration scenario.

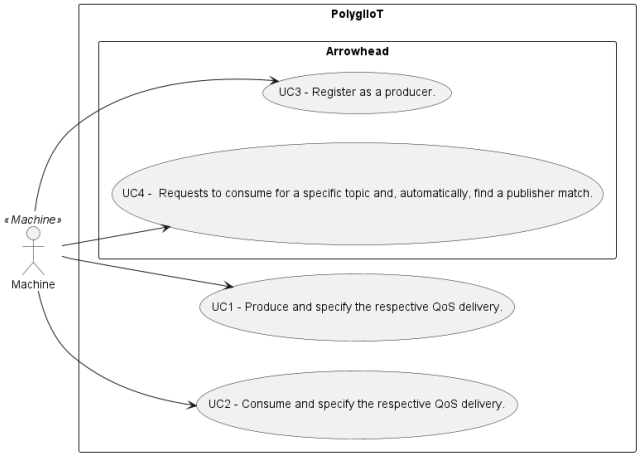


*Figure 16 – Use Case Diagram*

From Table 5 to 8, the identified use cases are going to be meticulously explained. What must be true before the use case is invoked, what states that the system can be after the use case runs and necessary data to be applied.

*Table 5 – UC1: Produce messages and specify the respective QoS delivery*

| | |
|---|---|
| *Use Case 1* | *Produce messages and specify the respective QoS delivery.* |

| | |
|---|---|
| *Description* | The system intends to produce messages to a broker. |
| *Actor(s)* | System |
| *Preconditions* | 1. Have access to PolyglIoT<br>2. Have access to the broker |
| *Postconditions* | 1. External publisher establishes connection with broker and produces data |
| *Necessary data* | • Broker address and port<br>• Protocol names involved |

*Table 6 – UC2: Consume messages and specify the respective QoS delivery*

| *Use Case 2* | *Consume messages and specify the respective QoS delivery.* |
|---|---|
| *Description* | The system intends to consume messages from a broker. |
| *Actor(s)* | System |
| *Preconditions* | 1. Have access to PolyglIoT<br>2. Have access to the broker |
| *Postconditions* | 1. External consumer establishes connection with broker and consumes data |
| *Necessary data* | • Broker address and port<br>• Protocol names involved |

*Table 7 – UC3: Register production request as a service*

| *Use Case 3* | *Register production request as a service.* |
|---|---|

| | |
|---|---|
| *Description* | The system intends to produce messages and register itself, in Arrowhead, as a producer. |
| *Actor(s)* | System |
| *Preconditions* | 1. Have access to PolyglIoT<br>2. Have access to Arrowhead<br>3. Have access to the broker |
| *Postconditions* | 1. External producer is defined as an Arrowhead's producer |
| *Necessary data* | • Arrowhead endpoints<br>• Valid authorization |

*Table 8 – UC4: Request consume and, automatically, find a publisher match*

| *Use Case 4* | *Request consume and, automatically, find a publisher match.* |
|---|---|
| *Description* | The system intends to consume messages and find, automatically, a producer match. |
| *Actor(s)* | System |
| *Preconditions* | 1. Have access to PolyglIoT<br>2. Have access to Arrowhead<br>3. Have access to the broker |
| *Postconditions* | 1. External consumer consumes data from the matched producer |
| *Necessary data* | • Arrowhead endpoints<br>• Valid authorization |

### 4.1.6 Process View

Process view presents sequence diagrams for specific use cases mentioned in the Analysis chapter. It provides insights into the dynamic aspects of the system. The following sequence diagrams follow Unified Modeling Language (UML) notation.

#### 4.1.6.1 Level 1

The use case “UC1 - Produce and specify the respective QoS delivery” is represented in Figure 17.

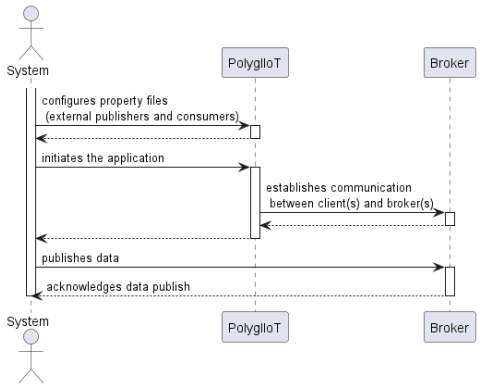


Figure 17 – Process View, level 1, UC1 (At Least Once + Exactly Once)

It is particularly important to refer that this diagram represents acknowledgements, however At Most Once QoS level will not have those, like it is represented in Figure 18.

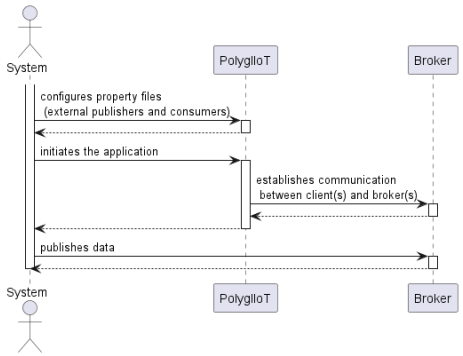


*Figure 18 – Process View, level 1, UC1 (At Most Once)*

Figure 19 represents “UC2 - Consume and specify the respective QoS delivery”. “Broker A” represents the shared broker between the system (external consumer) and PolygIIoT and “Broker B” is the broker where PolygIIoT’s internal consumers are consuming data.

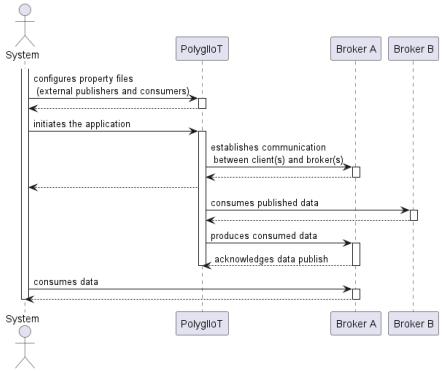


*Figure 19 – Process View, level 1, UC2*

Figure 20 represents “UC3 - Register as a producer.”. The System interacts with Arrowhead framework and requests its production registry.

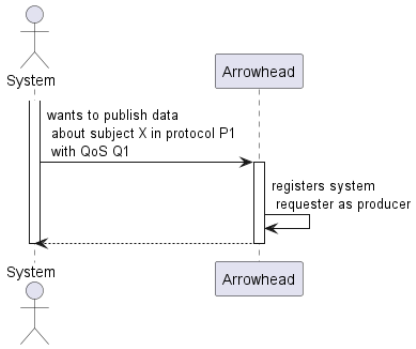


Figure 20 – Process View, level 1, UC3

Finally, Figure 21 shows “UC4 - Requests to consume for a specific topic and, automatically, find a publisher match.”. The system starts by interacting with Arrowhead, asking to find a publisher match for this subject. After finding it, Arrowhead requests Edge4CPS to instantiate PolygIIoT with the respective configurations. Finally, PolygIIoT executes and starts publishing the received data from the shared broker with the matched external publisher. The user can consume from the broker.

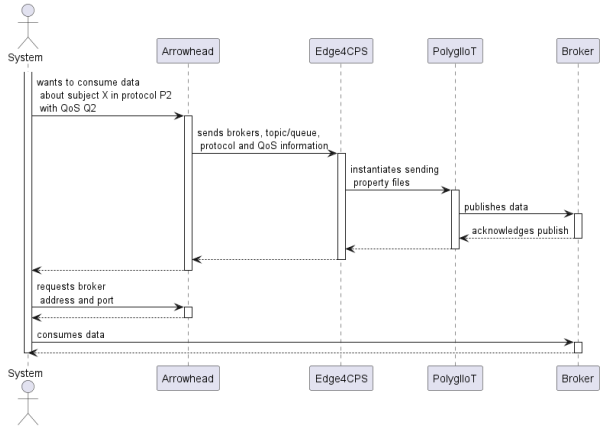


Figure 21 – Process View, level 1, UC4

#### 4.1.6.2 Level 2

UC1 is represented in Figure 22. PolygIIoT accesses configuration files to retrieve information related to broker addresses and ports, external producers and consumers, the respective topics/queues, communication protocols, QoS levels and associations between producers and consumers. Then producers and consumers are validated by ISettings and instantiated by IProducer and IConsumer, respectively.

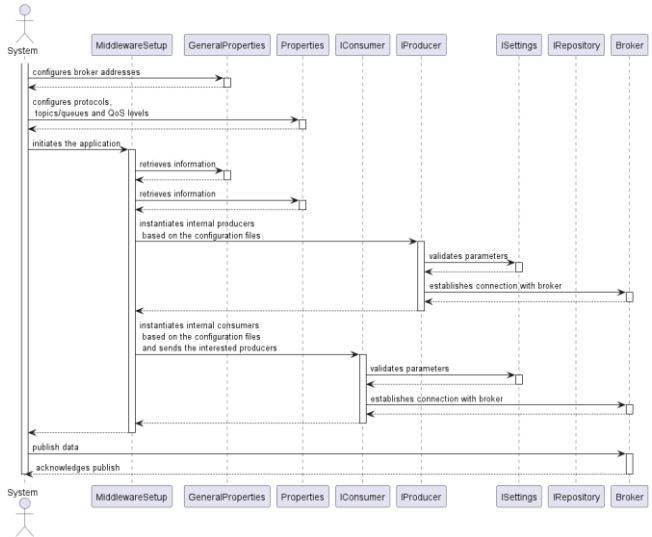


Figure 22 – Process View, level 2, UC1

Figure 23 represents UC2. For every consumer created, there is a new thread responsible for consuming from the shared brokers with the external producers and when the data is published, this thread spreads the consumed information across all the linked internal producers. Those are going to publish data in the respective broker. Finally, the external consumer consumes the data from the shared broker with the internal producers.

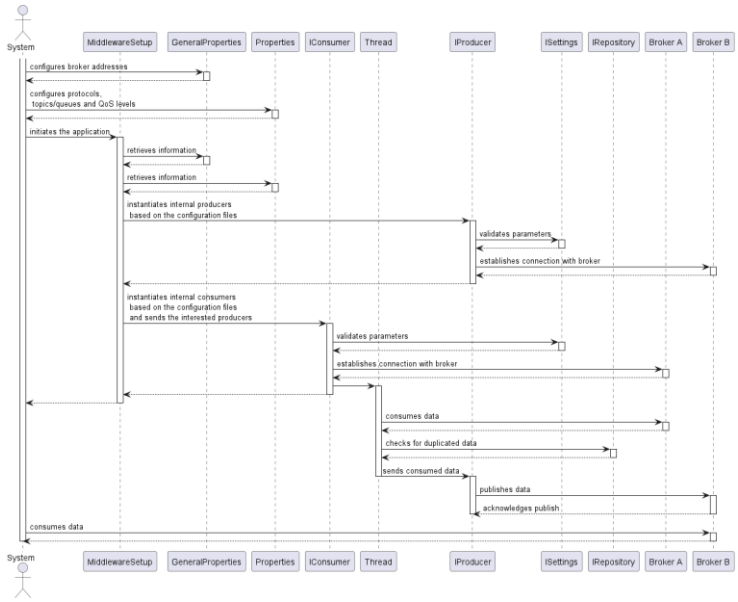


*Figure 23 – Process View, level 2, UC2*

Note that only Exactly Once QoS communications proceed to “check for duplicated data”, the remaining QoS’s do not verify.

Figure 24 presents UC3. The user communicates directly with the Service Registry. The Service Registry registers this service with the “Producer” definition.

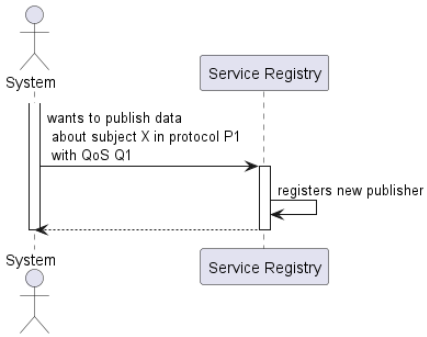


Figure 24 – Process View, level 2, UC3

Figure 25 depicts UC4. Whenever an user wants to consume data from a specific topic, using Arrowhead, it needs to request Orchestrator. This request must be authorized by Authorization and then Orchestrator asks Service Registry to query for a service with "Publisher" definition and containing the same topic as this consumer. After successfully matching these clients, it sends Edge4CPS a request containing all the necessary information. Then, Edge4CPS instantiates PolyglIoT with the specific data. Finally, PolyglIoT's internal publisher produces the consumed messages from the external publisher and the user (external consumer) can consume it in the desired protocol.

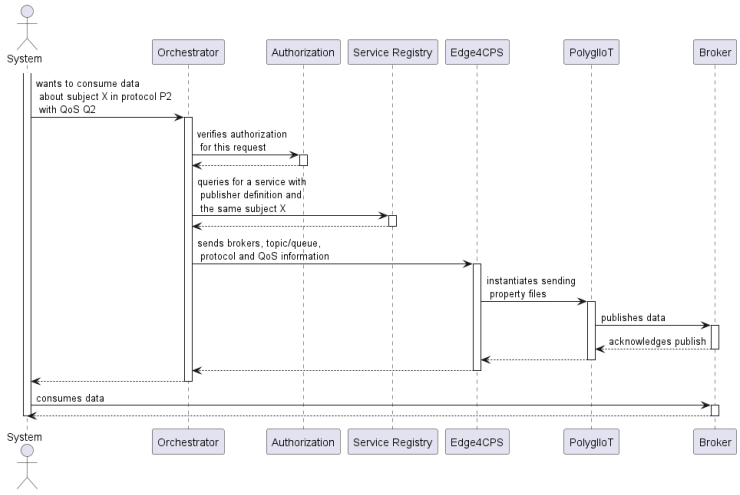


Figure 25 – Process View, level 2, UC4

#### 4.1.6.3 Level 3

This level is represented by UC1 (Figure 26) and UC2 (Figure 27). The remaining use cases (UC3 and UC4) do not require further elaboration, as they belong to the Arrowhead framework and are therefore beyond the scope of this report.

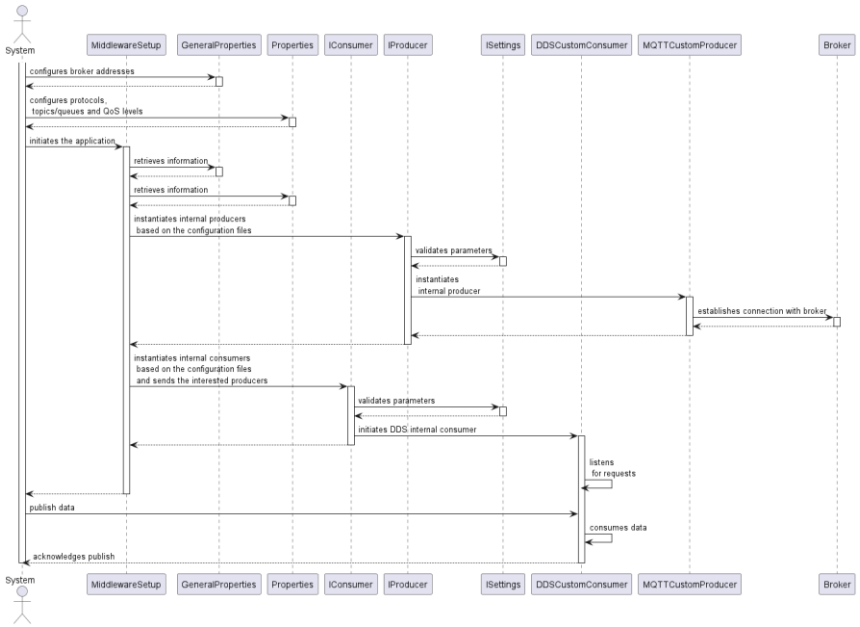


*Figure 26 – Process View, level 3, UC1*

Figure 26 illustrates an external producer, producing DDS messages. Figure 27 depicts the internal consumer that consumes these DDS messages, redirects them to the associated MQTT internal producer, and subsequently transmits them to the broker shared with the external consumer. It is noteworthy that these use cases could be with any combination of protocols (for example, Kafka as an internal consumer and RabbitMQ as an internal producer). Note also that only the internal producer is connected to “Broker” because DDS does not use it.

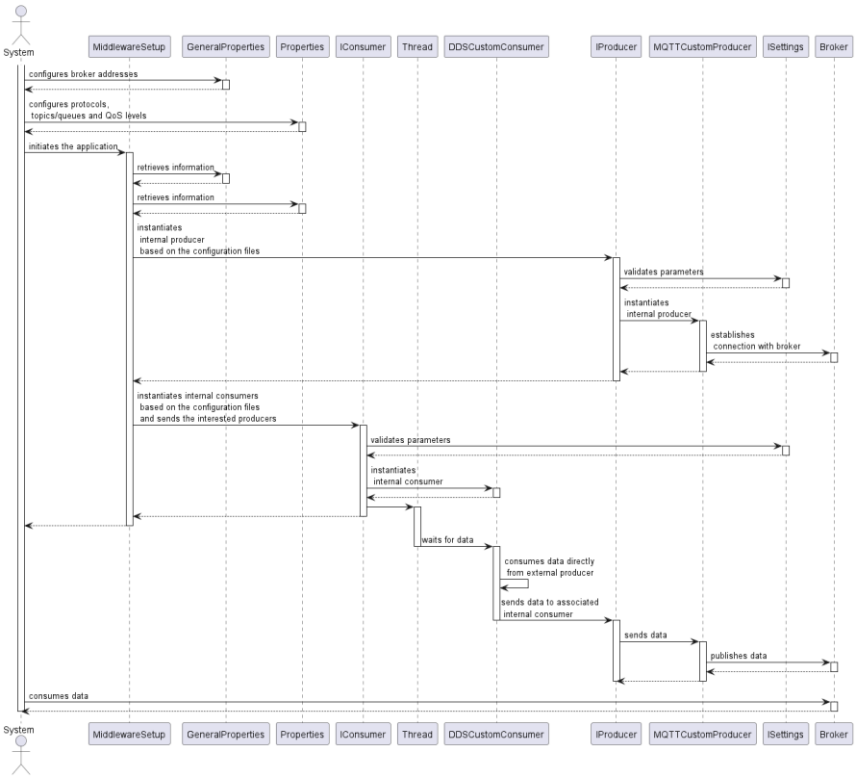


*Figure 27 – Process View, level 3, UC2*

This use case does not represent any access to IRepository (that helps handling Exactly Once packets), because DDS already eliminates duplicates.

# 5 Implementation

In this chapter, the presentation shows the design implementation, offering a detailed exploration of how the design concepts are realized and put into practice.

## 5.1 Technologies

To evolve this project, it made sense to continue using Java, since the project was previously developed, solely, in Java. MQTT, Kafka and RabbitMQ provide Java libraries that make it easier to implement, however DDS was trickier. DDS is a communication protocol that contains many implementations and most of those are in different programming languages. Nevertheless, OpenDDS was a suitable solution to this problem because it provides Java Native Interface.

The Java Native Interface (JNI) is a framework that allows Java code running in the Java Virtual Machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform) and libraries written in other languages like C and C++. JNI is part of the Java Development Kit (JDK) and provides a powerful way to leverage existing native libraries and to perform operations that are not possible with standard Java libraries alone (Oracle, “Java Native Interface Overview”).

Table 9 depicts every software used and the respective version.

Table 9 Software versions

| Software | Version |
|---|---|
| Java | 11.0.16 |
| Eclipse Paho MQTT | 1.2.5 |
| Apache Kafka | 3.2.3 |
| RabbitMQ | 5.9.0 |
| OpenDDS | 3.28 |
| Orchestrator (Arrowhead) | 4.6.0 |
| Authorization (Arrowhead) | 4.6.0 |
| Service Registry (Arrowhead) | 4.6.0 |

| | |
|---|---|
| Edge4CPS | 1.0 |

## 5.2 PolygIIoT Prerequisites

To be able to execute PolygIIoT successfully, there are several softwares that need to be installed and configured, as such as:

- OpenDDS: it is recommended to follow strictly the official OpenDDS's GitHub repository documentation (OpenDDS, "The OpenDDS Project"), because it might be challenging to set OpenDDS with JNI;
- Arrowhead: is it recommended to install the three core services (Orchestrator, Authorization and Service Registry) through Arrowhead tools (AITIA, "Arrowhead Tools"). Note that Arrowhead requires Java and MySQL installed;
- Edge4CPS: it is recommended to follow B. Cabral (2023) to install and configure this software properly.

## 5.3 Implementation description

This section elaborates on the implementation of the solution proposed in the preceding chapter. It offers comprehensive explanations and evidence of solutions.

### 5.3.1 Application Settings

There are two fundamental configuration files: general properties and properties. Properties file (Code Snippet 1) consists of a JSON file that allows the user to specify producers, consumers, QoS, topics/queues and associations between producers and consumers.

```
{
  "Producers": [{
    "internal.id": "device01", "protocol" : "kafka",
    "additional.props": { "topic": "Test1",
      "connection.timeout": 20,
      "qos": 0,
      "client.id": "middleware-producer"
    }
  }],
  "Consumers": [{
      "internal.id": "device02", "protocol" : "mqtt",
      "additional.props": {
        "topic": "Test2",
        "client.id" : "middleware-consumer",
        "qos": 0
      }
    }
    ],

  "Streams": [{
    "from.producers" : "device01",
    "to.consumers" : "device02"
    }]
}
```

*Code Snippet 1 - Properties configuration file example*

The example depicted in Code Snippet 1, shows that a Kafka producer is going to produce to topic “Test1” with QoS 0, and a MQTT consumer is going to consume from topic “Test2” also with QoS 0.

- Internal ID – Indicates a unique identifier for PolygIIoT. Must be unique.
- Protocol – Must be inserted a communication protocol. Values are “mqtt”, “kafka”, “dds” and “rabbit”.
- Additional Props – Here, topic, QoS and the client ID are specified:
    - Topic – Subject that the producers and consumers are going to communicate. Must be a non-empty string value.
    - QoS – Delivery level that messages from this producer/consumer are going to be produced/consumed. It must be an integer in range from 0 to 2, where At Most Once is mapped to 0, At Least Once to 1 and Exactly Once to 2.
    - Client ID – Name to identify the client. Must be a non-empty string value.
    - Connection Timeout – Maximum amount of time to establish connection to the broker.
- Streams – Association between consumers and producers. Must contain valid Internal IDs.

General properties file consists of a JSON file that allows to establish the brokers addresses, ports, and arrowhead integration.

```
"default_brokers" :
  {
    "mqtt" : {
      "address" : "1.1.1.1",
      "port" : 1883
    },

    "kafka" : {
      "address" : "1.1.1.1",
      "port" : 9092
    },

    "rabbit": {
      "address" : "2.2.2.2",
      "port" : 15672
    }
  }
}
```

*Code Snippet 2 - General properties configuration file example*

The JSON structure must always be the same, differing only the addresses and ports (Code Snippet 2). Note that DDS does not need to be specified because it establishes communication directly between producer and consumer.

### 5.3.2 IProducer and IConsumer

These classes are the key to all abstraction implemented in this application. Every implemented communication protocol must extend one of these classes. IProducer contains an abstract method called “produce”, meaning that every class that extends IProducer needs to develop its own implementation according to its logic (Code Snippet 3).

IConsumer class contains, in contrast to IProducer, an already implemented method called OnMessageReceived. This method is invoked every time that is consumed a message from a broker. There is a multithreading handling for this situation, for each producer linked to a specific consumer, there is going to be created a new thread that is responsible for producing a specific message, improving simultaneously translations.

```
public void OnMessageReceived(String topic, String message) {

    List<Thread> producersThreads = new ArrayList<>();

    for (IProducer producer : producerList) {
        producersThreads.add(new Thread(new ProducerThread(producer, topic, message)));
        producersThreads.get(producersThreads.size() - 1).start();
    }
    try {
        for(Thread producer : producersThreads)
            producer.join();
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Code Snippet 3 - *IConsumer method OnMessageReceived*

### 5.3.3 Setup

Firstly, internal consumers and producers, and the respective classes are put on a consumer and a producer map data structure (Code Snippet 4), respectively.

```
private void createMaps() {
    logger.info("Loading initial configs");

    consumerMap.put("mqtt","eu.arrowhead.application.skeleton.consumer.classes.mqtt.MqttCustomConsumer");
    consumerMap.put("teste","eu.arrowhead.application.skeleton.consumer.classes.testServices.PeriodicConsumer");
    consumerMap.put("kafka","eu.arrowhead.application.skeleton.consumer.classes.kafka.KafkaCustomConsumer");
    consumerMap.put("rabbit","eu.arrowhead.application.skeleton.consumer.classes.rabbit.RabbitCustomConsumer");
    consumerMap.put("dds", "eu.arrowhead.application.skeleton.consumer.classes.dds.DDSCustomConsumer");

    producerMap.put("mqtt","eu.arrowhead.application.skeleton.consumer.classes.mqtt.MqttCustomProducer");
    producerMap.put("teste","eu.arrowhead.application.skeleton.consumer.classes.testServices.ConsoleProducer");
    producerMap.put("kafka","eu.arrowhead.application.skeleton.consumer.classes.kafka.KafkaCustomProducer");
    producerMap.put("rabbit","eu.arrowhead.application.skeleton.consumer.classes.rabbit.RabbitCustomProducer");
    producerMap.put("dds", "eu.arrowhead.application.skeleton.consumer.classes.dds.DDSCustomProducer");
}
```

Code Snippet 4 - *Producer and consumer map creation*

Then, data from general properties (broker addresses, ports and arrowhead integration) is retrieved to configure internal producers and consumers, that then, are created iterating over

the general configuration file (Code Snippets 5 and 6). This type of implementation allows scaling due to its abstraction, it does not need to explicitly mention the class because it is already saved into consumer map and producer map.

```
private IConsumer createConsumer(ConnectionDetails cd, List<IProducer> producer, String name, Map<String, String> settings) {
    Class<?> c;
    try {
        c = Class.forName(consumerMap.get(name));
        Constructor<?> cons = c.getConstructor(ConnectionDetails.class, List.class, Map.class);
        return (IConsumer) cons.newInstance(cd,producer,settings);
    } catch (ClassNotFoundException | InvocationTargetException | NoSuchMethodException | InstantiationException |
            IllegalAccessException e) {
        throw new RuntimeException(e);
    }
}
```

Code Snippet 5 - *Consumer creation*

```
private IProducer createProducer(ConnectionDetails cd, String name, Map<String,String> settings) {
    Class<?> c;
    try {
        c = Class.forName(producerMap.get(name));
        Constructor<?> cons = c.getConstructor(ConnectionDetails.class, Map.class);
        return (IProducer) cons.newInstance(cd, settings);
    } catch (ClassNotFoundException | InvocationTargetException | NoSuchMethodException | InstantiationException |
            IllegalAccessException e) {
        throw new RuntimeException(e);
    }
}
```

Code Snippet 6 - *Production creation*

After these creations, they are associated regarding “Streams” from the same general configuration file. Finally, for each internal consumer, a thread is going to be created and responsible for consuming data and reproduce it for interested internal producers.

### 5.3.4 QoS Mapping

The following topics approach, in detail, how each QoS was implemented and assured for each supported protocol.

#### 5.3.4.1 MQTT

The QoS delivery semantics were inspired on MQTT QoS levels, meaning that MQTT already has this QoS mapping built in. However, visualizing Code Snippet 7, that represents the producer view, there are some additional configurations like retained and clean session.

```
MqttMessage mqttMessage = new MqttMessage(message.getBytes());
MqttConnectOptions conn = new MqttConnectOptions();
int qos = settings.getQos();

if(qos == 0){
    configureParameters(mqttMessage, conn,  cleanSession: true,  retained: false);
}else if(qos == 1){
    configureParameters(mqttMessage, conn,  cleanSession: true,  retained: true);
}else{
    configureParameters(mqttMessage, conn,  cleanSession: false,  retained: true);
}

mqttMessage.setQos(qos);
client.publish(publishToTopic, mqttMessage);
```

*Code Snippet 7 - MQTT Producer QoS assuring*

The consumer, after receiving a message, retrieves the consumed message's QoS and if it matches Exactly Once, is queried in the memory persisted database and if there isn't any match, registers it.

#### 5.3.4.2 Kafka

Observing Code Snippet 8 is possible to see that producer configurations are directly inserted regarding the specified QoS.

```
switch (settings.getQos()) {
    case 0:
        config.put(ProducerConfig.ACKS_CONFIG, "0");
        break;
    case 1:
        config.put(ProducerConfig.ACKS_CONFIG,"all");
        break;
    case 2:
        config.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG, "true");
        config.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTION, "5");
        break;
}
```

*Code Snippet 8 - Kafka Producer QoS assuring*

For the consumer, it also has the Exactly Once helper and applies the same properties.

#### 5.3.4.3 RabbitMQ

For At Most Once level, a message is built and published. Nevertheless, for At Least Once and Exactly Once levels, an auxiliary data structure was necessary, namely, a concurrent navigable map. This structure represents a thread-safe and navigable map (Oracle, "*Java SE Documentation*", C). For every produced message, this map registers a new entry with the message identifier and the confirmation state, since both levels imply acknowledging, that is when the messages are removed from the map (if they were acknowledged) or reproduced (if they were not acknowledged). It is relevant to say that this structure is useful due to its thread-safety, since acknowledging is asynchronous. Code Snippet 9 depicts

acknowledgement. Notice that “multiple” condition indicates if multiple (a batch of) messages were received.

```
ConfirmCallback confirmCallback = (sequenceNumber, multiple) -> {
    if(multiple){
        int amount = 0;
        Iterator<Map.Entry<Long, String>> it = outstandingConfirms.descendingMap().entrySet().iterator();
        while(it.hasNext() || outstandingConfirms.containsKey(sequenceNumber)){
            if(it.next().getKey() <= sequenceNumber) {
                it.remove();
                amount++;
            }
        }
        numberOfMessages += amount;
    }else{
        outstandingConfirms.remove(sequenceNumber);
        numberOfMessages++;
    }
};
```

Code Snippet 9 - *RabbitMQ QoS assuring* for acknowledgement

Code Snippet 10 represents the not acknowledged messages and it is the same logic. Multiple indicates if there are a batch of messages not acknowledged and if so, all those messages are reproduced.

```
ConfirmCallback nackedConfirmCallback = (sequenceNumber, multiple) -> {
        if(multiple){
            ConcurrentNavigableMap<Long, String> nAcked = outstandingConfirms.headMap(sequenceNumber,  inclusive: true);
            nAcked.forEach((id, message) -> produceMessage(settings.getQueue(), message));
        }else{
            produceMessage(settings.getQueue(), outstandingConfirms.get(sequenceNumber));
        }
    };
```

*Code Snippet 10 - RabbitMQ QoS assuring for not acknowledgement*

#### 5.3.4.4 OpenDDS

DDS cannot perform an At Least Once QoS delivery because it automatically eliminates duplicates. It is noteworthy that “Duration_T” attributes are seconds and nanoseconds, respectively, meaning that QoS 0’s (At Most Once) deadline is 0.5 seconds (Code Snippet 11).

```
if(qos == 0){
    dw_qos.reliability.kind = ReliabilityQosPolicyKind.from_int(ReliabilityQosPolicyKind._BEST_EFFORT_RELIABILITY_QOS);
    dw_qos.deadline.period = new Duration_t(0, 500000000);
    dw_qos.destination_order.kind = DestinationOrderQosPolicyKind.from_int(DestinationOrderQosPolicyKind
            ._BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS);
    reliable = false;
}else{
    dw_qos.reliability.kind = ReliabilityQosPolicyKind.from_int(ReliabilityQosPolicyKind._RELIABLE_RELIABILITY_QOS);
    dw_qos.deadline.period = new Duration_t(1, 0);
    dw_qos.destination_order.kind = DestinationOrderQosPolicyKind.from_int(DestinationOrderQosPolicyKind
            ._BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS);
}
```

*Code Snippet 11* - OpenDDS QoS assuring

When reliability performs in best effort mode, DDS performs At Most Once QoS. However, when reliability performs in reliable mode, DDS performs Exactly Once QoS.

### 5.3.5 Performance and Resource metrics

Since these metrics are going to be used by every producer, it does not make sense to replicate code. The best solution is to have utility code and the same instance shared among all threads, however this might lead to race conditions due to multiple threads accessing and modifying the same variables. There are two main methods called “initializeCounting()” (Code Snippet 12) and “pointReached(long id, Logger log)” (Code Snippet 14).

```
public static long initializeCounting(){
    long myID = incrementIdentifier();

    List<Integer> tcl = new ArrayList<>();
    List<Integer> tpc = new ArrayList<>();
    List<String> hu = new ArrayList<>();
    List<String> nhu = new ArrayList<>();
    List<Integer> ap = new ArrayList<>();
    List<Double> sla = new ArrayList<>();

    insertNewList(myID, threadCount, tcl);
    insertNewList(myID, threadPeakCount, tpc);
    insertNewList(myID, heapUsage, hu);
    insertNewList(myID, nonHeapUsage, nhu);
    insertNewList(myID, availableProcessors, ap);
    insertNewList(myID, sysLoadAvg, sla);

    threads(tcl, tpc);
    memory(hu, nhu);

    insertNewTime(myID);

    return myID;
}
```

*Code Snippet 12 - Initialize Couting method*

Code Snippet 12 depicts the beginning of this process. It creates lists that are going to store information about threads, memory and CPU. It also saves the starting time (for messages per second effects). Variable “myID” is an unique identifier for each instance that invokes this method. Note that “insertNewList()” (Code Snippet 13) does not apply fine-grained synchronization because it would remove the abstract and generic way of inserting new lists into the main map.

```
private static <E> void insertNewList(long id, Map<Long, ? super List<E>> map, List<E> list){
    synchronized (generalMapLock) {
        map.put(id, list);
    }
}
```

Code Snippet 13 - *List insertion with* generic synchronization

The method “pointReached(long id, Logger log)” (Code Snippet 14) is called whenever it is supposed to present the results. This block of code applies fine-grained synchronization because there are two different objects locking different types of resources leading to more efficient multithreading execution. It also deletes unnecessary data, because it won’t be used

anymore, since every new initialize counting assigns a new identifier. Note that variable “sb” is a “StringBuffer” class that is thread-safe, which explains why it is not being covered by the “synchronized” exclusivity.

```
public static void pointReached(long id, Logger log){

    long execTime = System.currentTimeMillis() - accessInitialTime(id);
    sb.append("Messages per second + ").
            append(100000f / (execTime / 1000f)).
            append(" ||| Execution time: ").
            append(execTime / 1000f);

    synchronized (terminalOutputLock) {
        if(log != null)
            log.info(sb.toString());
        else
            System.out.println(sb);
    }

    synchronized (generalMapLock) {
        cpu(accessAvailableProcessors(id), accessSysLoadAvg(id));
        cpuInfo(accessAvailableProcessors(id), accessSysLoadAvg(id), log);
        memoryInfo(accessHeapUsage(id), accessNonHeapUsage(id), log);
        threadsInfo(accessThreadCount(id), accessThreadPeakCount(id), log);
    }

    deleteUnnecessaryData(id);
}
```

*Code Snippet 14 - Point Reached method*

### 5.3.6 DDS Producer and Consumer

To create a successful DDS producer or consumer, there are several steps that need to be assured. Firstly, there is a configuration file (Code Snippet 15), typically designed by “tcp.ini” (when TCP is used as transport mechanism, it might be also UDP) and it defines diverse settings that control how DDS entities communicate over TCP/IP.

```
[common]
DCPSGlobalTransportConfig=$file
DCPSDefaultDiscovery=DEFAULT_RTPS

[transport/the_transport]
transport_type=tcp
local_address=192.168.1.154:12345
```

*Code Snippet 15 – OpenDDS “tcp.ini” file*

In section “common”, the property “DCPSDefaultDiscovery” specifies the default discovery mechanism to be used by OpenDDS, in this case Real-Time Publish-Subscribe (RTPS) protocol, which is the standard for interoperability. Property “DCPSGlobalTransportConfig” indicates that the transport configuration settings will be loaded from this file. The “$file” placeholder specifies that the current file contains the transport configurations to be used globally.

In section “transport/the_transport”, the property “local_address” indicates the address and port of the publisher or consumer. It is noteworthy to say that it is not allowed to have multiple

producers or consumers using the same IP address and port. The property “transport_type” sets the transport mechanism.

Code Snippet 16 represents an excerpt of code that will passively wait (OpenDDS, “Conditions and Listeners”) until the current thread finds a match or timeout occurs. The variable “timeout” was defined as infinite, meaning that it is mandatory that, before producing, a match with at least one consumer needs to exist.

```
ConditionSeqHolder cond = new ConditionSeqHolder(new Condition[]{});
if (ws.wait(cond, timeout) != RETCODE_OK.value) {
    System.err.println("ERROR: wait() failed.");
    return;
}
```

*Code Snippet 16 – Producer match waiting*

OpenDDS implements, implicitly, asynchronous publishing. Asynchronous publishing refers to the mechanism where data is written by an internal thread rather than directly in the user thread (RTI Core Libraries, “PUBLISH_MODE QoSPolicy (DDS Extension)”). This means that instead of waiting for the message to be delivered to all consumers, it continues to produce, and a separate internal thread is waiting for the acknowledgement. Besides this efficient built-in mechanism, a batch producing strategy was also implemented (Code Snippet 17) to refine the production velocity.

```
private void addToBatch(Message message, MessageDataWriter mdw, int handle){
    messageBatch.add(message);
    int batchSize = 100;
    int batchTimeout = 2000;
    if(messageBatch.size() >= batchSize || (System.currentTimeMillis() - lastSentTime) >= batchTimeout){
        sendBatch(mdw, handle);
        lastSentTime = System.currentTimeMillis();
        messageBatch.clear();
    }
}
```

*Code Snippet 17 – Batch strategy*

The “batchSize” variable defines the max number of messages that a batch can send, “batchTimeout” sets the maximum number of milliseconds that a batch might accumulate. Messages to be produced are stored in the batch (list) and if it reaches a hundred messages or more, or two thousand milliseconds, the batch is produced.

# 6 Unit and Performance Tests

Several kinds of tests were made to evaluate the system performance, efficiency, and functionality. System performance tests consist of a stand-alone testbed that evaluates the production speed for each protocol as producer/consumer. Multi-translation tests evaluate the system's capacity of performing simultaneously translations. Unit tests, assure that the system is running as expected and imposed by the diverse requirements.

## 6.1 Tests Characteristics

For system performance and multi-translation tests, was setup a standalone testbed, comprised of three PCs (let it be A, B and C) connected, via wireless, to a dedicated network (300MB/s WiFi).

Computer A acted as an external producer, and it shared the broker with the PolyglIoT's internal consumer. Computer B, acted as a broker and PolyglIoT. Finally, computer C acted as an external consumer, and it shared the same broker with internal producer.

The following list, represents the above-mentioned computers characteristics:

- External Producer (PC A) - HP ProBook 6460b, Linux Mint 21.3 Virginia, Intel Celeron B840 dual core, 2GB RAM.
- Broker & PolyglIoT (PC B) - Toshiba PORTEGE Z30-C, Linux Mint 21.3 Virginia, i7-6500U quad core, 16GB RAM.
- External Consumer (PC C) - Lenovo IdeaPad S145-15API, Ubuntu 22.04.4 LTS (Jammy Jellyfish), AMD Ryzen 3 3200U, 6GB RAM.

### 6.1.1 System Performance Tests

The external producer was responsible for producing data for the shared broker with the PolyglIoT's internal consumer. There was only one external producer and one external consumer (Figure 28) and every combination of protocol and QoS was tested.

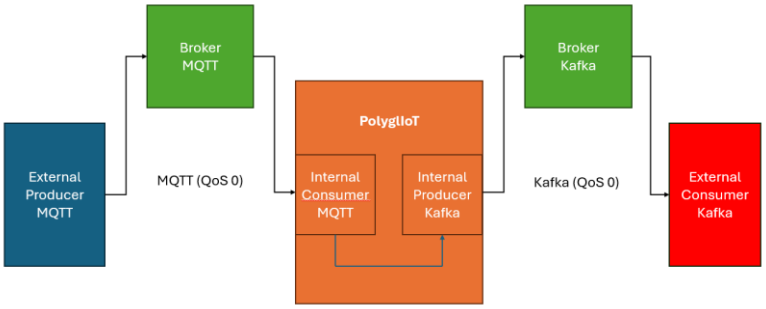


*Figure 28 – System Performance Test, MQTT to Kafka example*

The tests were performed by sending 10 batches of 100k messages in a single message stream (using a single topic), each with a payload of 5 bytes, to which it must be added the intrinsic additional information from each protocol. For every 100k messages successfully sent, the number of messages per second and the execution time was recorded on the internal producer side. It is noteworthy that, for reliable QoS levels (At Least Once and Exactly Once), the count was made solely when the acknowledgement packet was received. On the least reliable QoS level (At Most Once), the count was made right after production.

### 6.1.2 Multi-Translations Tests

The setup is the same as the system performance tests, however this test aims to assess the system's simultaneous translation capacity (Figure 29). It tested every combination of protocol and QoS level.

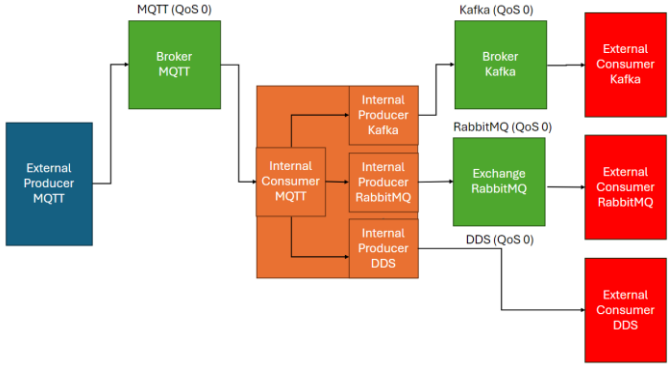


*Figure 29 – Multi-Translation Test, MQTT to Kafka, RabbitMQ and DDS example*

It sent 10 batches of 10K messages in a single message stream, each with a payload of 5 bytes.

### 6.1.3 Unit Tests

This type of testing involves a technique that tests individual components or modules of a software application that are tested in isolation. The primary goal of unit testing is to validate that each unit of the software performs as expected (AWS, “What is Unit Testing?”).

Code Snippet 18 demonstrates that invalid QoS is not acceptable, specifically, if the value is null.

```
@Test
public void nullQoSThrowsQoSException(){

    Map<String, String> settings = new HashMap<>();

    settings.put("topic","test");

    Assertions.assertThrows(InvalidQoSException.class,
            () -> new DDSCustomProducer(ddsConnection, settings));

}
```

*Code Snippet 18 – Invalid QoS (null) throws an exception unit test*

Code Snippet 19 is analogous to the preceding one, yet it demonstrates that the QoS value must be an integer within the range of -3 to +1.

```
@Test
public void invalidQoSThrowsQoSException(){

    Map<String, String> settings = new HashMap<>();

    settings.put("topic","test");
    settings.put("qos", "-1");

    Assertions.assertThrows(InvalidQoSException.class,
            () -> new DDSCustomProducer(ddsConnection, settings));

    settings.put("topic","test");
    settings.put("qos","3");

    Assertions.assertThrows(InvalidQoSException.class,
            () -> new DDSCustomProducer(ddsConnection, settings));
}
```

*Code Snippet 19 – Invalid QoS (out of range) throws an exception unit test*

### 6.1.4 Integration Test

The integration test is a process whereby components of the software are gradually combined and then tested as a unified group. Although these components typically function correctly on

their own, they may encounter issues when integrated with other components (Katalon, “What is Integration Testing? Definition, How-To, Examples”).

Code Snippet 20 depicts an integration test involving brokers, DDS internal consumer, DDS external producer, MQTT internal producer, MQTT external consumer, Kafka internal producer, Kafka external consumer, RabbitMQ internal producer and RabbitMQ external consumer. This test sets up one internal consumer (DDS) and three internal producers (MQTT, RabbitMQ and Kafka) associated with the internal consumer. Additionally, there are the respective external producer (DDS) and external consumers (MQTT, RabbitMQ and Kafka).

```
public void multiTranslationDDS() throws InterruptedException, RuntimeException, IOException {

    ConnectionDetails mqttConnection;
    ConnectionDetails rabbitConnection;
    ConnectionDetails kafkaConnection;

    rabbitConnection = midSetup.loadBroker("rabbit");
    mqttConnection = midSetup.loadBroker("mqtt");
    kafkaConnection = midSetup.loadBroker("kafka");

    mqttInternalProducer = new MqttCustomProducer(mqttConnection, settings);
    kafkaInternalProducer = new KafkaCustomProducer(kafkaConnection, settings);
    rabbitInternalProducer = new RabbitCustomProducer(rabbitConnection, settings);

    rabbitExternalConsumer = new RabbitCustomConsumer(rabbitConnection,  producer: null, settings);
    kafkaExternalConsumer = new KafkaCustomConsumer(kafkaConnection,  producer: null, settings);
    mqttExternalConsumer = new MqttCustomConsumer(mqttConnection,  producer: null, settings);

    List<IProducer> producerList = new ArrayList<>();
    producerList.add(rabbitInternalProducer);
    producerList.add(kafkaInternalProducer);
    producerList.add(mqttInternalProducer);

    ddsInternalConsumer = new DDSCustomConsumer(ddsConnection, producerList, settings);

    new Thread(ddsInternalConsumer).start();

    List<String> results = new ArrayList<>();
    results.add(kafkaExternalConsumer.getLastMessage());
    results.add(rabbitExternalConsumer.getLastMessage());
    results.add(mqttExternalConsumer.getLastMessage());

    final String finalMessage = "Test";

    Thread.sleep( millis: 10000);

    for(String s : results)
        Assertions.assertEquals(s, finalMessage);
}
```

*Code Snippet 20 – Multi-translation with DDS as external producer integration test*

The external producer initiates the flow by producing a message, “Test”, which is then processed by the internal consumer and distributed across the internal producers. The internal producers will produce for the respective broker. To validate the translation process, a comparison is made between the external consumers last received message, which must be

"Test". Note that there is a "thread sleep" due to the matching between DDS's internal consumer and external producer that takes a few seconds.

## 6.2 Solution Evaluation

Here, are going to be presented and discussed the results of the above-mentioned and explained tests.

### 6.2.1 System Performance Tests Results

The obtained results are depicted in the following tables (from Table 10 to 12). Table 10 represents the number of messages that can be processed per second, for the At Most Once QoS level which implies a minimum effort by the producers, and it is a fire are forget mechanism. These values are the produced messages per second with a 95% confidence interval. Horizontally, it is possible to visualize the same producer, producing for different consumers. Vertically, it is the opposite, meaning that it is possible to visualize the same consumer, consuming from different producers.

*Table 10 - At Most Once Performance Results*

| QoS "At most Once" | | Consumer | | | |
|---|---|---|---|---|---|
| | | Kafka | MQTT | RabbitMQ | OpenDDS |
| Prod. | Kafka | | 17538 ± 674 | 18371 ± 74 | 9321 ± 683 |
| | MQTT | 11204 ± 38 | | 10984 ± 19 | 6879 ± 292 |
| | RabbitMQ | 42361 ± 1262 | 12357 ± 1670 | | 9750 ± 793 |
| | OpenDDS | 2703 ± 11,63 | 2548 ± 33 | 3137 ± 18 | |

Not surprisingly, DDS is heavily penalized due to the usage of JNI. In most cases, the results are asymmetric, meaning that producing with protocol X and consuming with Y isn't the same as producing with Y and consuming with X. For example, when Kafka acts as producer and MQTT as consumer, it produces 17538 messages per second, however, when MQTT acts as producer and Kafka as consumer, it produces 11204 messages per second (36,12% decrease). These differences are mostly due to the different queuing mechanism implementations from each communication protocol. Indeed, was found that the average end-to-end delay, which is measured from the time a message is produced until it reaches the consumer varies from 24 µS (in the RabbitMQ/Kafka combination) to 390 µS (in the case of the DDS/MQTT combination).

Table 11 presents the At Least Once results, which implies more reliability in the message transmission, assuring that at least once message is delivered. It is noteworthy that DDS cannot produce this QoS level due to its nature of, automatically removing duplicates.

*Table 11 - At Least Once Performance Results*

| QoS "At least Once" | | Consumer | | | |
|---|---|---|---|---|---|
| | | Kafka | MQTT | RabbitMQ | OpenDDS |
| Prod. | Kafka | | 3428 ± 29 | 4260 ± 40 | 1532 ± 22 |
| | MQTT | 10886 ± 72 | | 11391 ± 70 | 7555 ± 304 |
| | RabbitMQ | 31103 ± 834 | 7047 ± 217 | | 6106 ± 277 |
| | OpenDDS | | | | |

In some cases, it is possible to observe a sharp reduction in the number of transmitted messages due to the protocol settings required to assure this QoS level. Curiously, this decrease is more accented when Kafka acts as a producer reaching a decrease of 80% in the number of transmitted messages. For DDS, it was considered the usage of its most reliable QoS setting.

Finally, Table 12 presents the results of the Exactly Once QoS level, which means that a reliable and complex mechanism needs to be used assuring that there are no message duplications.

*Table 12 – Exactly Once Performance Results*

| QoS "Exactly Once" | | Consumer | | | |
|---|---|---|---|---|---|
| | | Kafka | MQTT | RabbitMQ | OpenDDS |
| Prod. | Kafka | | 2378 ± 32 | 2196 ± 23 | 2565 ± 147 |
| | MQTT | 11209 ± 10 | | 6106 ± 277 | 7704 ± 303 |
| | RabbitMQ | 28713 ± 541 | 4838 ± 172 | | 6159 ± 314 |
| | OpenDDS | 2519 ± 10 | 1947 ± 12 | 3084 ± 16 | |

Again, an increase on the QoS level, generally, decreases the data delivery rate, reaching an 86% decrease in the more visible case of the Kafka producer when compared to QoS At Most Once and 30% if compared to QoS At Least Once. (Oracle, "*Java SE Documentation*", A). These results partially confirm the finding of W. Y. Liang .et. al (2023) and T. M. Tukade .et .al (2018), regarding the better performance in message dispatching of Kafka when compared to MQTT and RabbitMQ clients. However, it is also the most affected protocol by the QoS changes

### 6.2.2 Multi-Translation Tests Results

The following tables (from 13 to 16) represent the results of the multi-translation tests of each one of the supported protocols. According to (Oracle, "*Java SE Documentation*", A), heap memory is the area of memory used for dynamic memory allocation for Java objects and JRE classes during runtime. Non-heap memory refers to other areas of memory used by the JVM for different purposes beyond object storage. Threads refers to the highest number of live threads recorded since the Java Virtual Machine (JVM) started. The system load average (SLA) represents the average number of runnable processes (those using the CPU or waiting for the

CPU) and processes waiting for disk I/O to complete. This value is averaged over a specific period, typically 1, 5, and 15 minutes.

In the heap and non-heap section, there are some abbreviations with the specified meaning according to (Oracle, "*Java SE Documentation*", B):

- "i" – initial memory, amount of memory that the JVM initially requests.
- "u" – used memory, amount of memory that is currently being used by the JVM.
- "c" – committed memory, amount of memory that is guaranteed to be available for use by the JVM.
- "m" – maximum memory, upper limit of memory that can be used for memory management.
- "SLA" – system load average.
- "TC" – thread count.

Note that initial and maximum memory might be undefined and if so, the representation is -1 (negative one).

There are some values that remain the same all over the tests, namely, the initial and maximum in both heap and non-heap memories.

*Table 13 – OpenDDS Multi-Translation Test Results*

| | | Simultaneously Internal Producers | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | MQTT + RabbitMQ + Kafka | | | | | | | | | |
| | | Threads | Memory | | | | | | | | CPU |
| | | TC | Heap | | | | Non Heap | | | | SLA |
| | | | i | u | c | m | i | u | c | m | |
| Internal Consumer OpenDDS | QoS "At most Once" | 18 | 250 | 95 | 389 | 3978 | 5 | 51 | 55 | -1 | 1,358 |
| | QoS "At least Once" | | | | | | | | | | |
| | QoS "Exactly Once" | 21 | 250 | 78 | 397 | 3978 | 5 | 51 | 55 | -1 | 2,327 |

Observing Table 13, DDS acts as an internal consumer and Kafka, MQTT and RabbitMQ act as internal producers across the diverse QoS levels. Elevating the QoS reliability implies directly on the thread counting and SLA values. At Most Once has the lowest SLA and thread counting, indicating that it is the least resource intensive as expected. Exactly Once has the highest SLA increasing 58% from At Most Once and more threads. Surprisingly, At Most Once uses more heap memory than Exactly Once, however both have, practically, the same amount of committed memory.

*Table 14 – Kafka Multi-Translation Test Results*

| | | Simultaneously Internal Producers | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | MQTT + RabbitMQ + OpenDDS | | | | | | | | | |
| | | Threads | Memory | | | | | | | | CPU |
| | | TC | Heap | | | | Non Heap | | | | SLA |
| | | | i | u | c | m | i | u | c | m | |
| Internal Consumer Kafka | QoS "At most Once" | 20 | 250 | 61 | 385 | 3978 | 5 | 53 | 57 | -1 | 1,5125 |
| | QoS "At least Once" | 19 | 250 | 95 | 363 | 3978 | 5 | 53 | 56 | -1 | 1,65 |
| | QoS "Exactly Once" | 20 | 250 | 96 | 375 | 3978 | 5 | 53 | 56 | -1 | 1,61 |

Observing Table 14, Kafka acts as an internal consumer and OpenDDS, MQTT and RabbitMQ act as internal producers across the diverse QoS levels. Despite At Most Once having the most committed memory, uses the least. This means that its resource usage is low.

*Table 15 – MQTT Multi-Translation Test Results*

| | | Simultaneously Internal Producers | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | OpenDDS + RabbitMQ + Kafka | | | | | | | | | |
| | | Threads | Memory | | | | | | | | CPU |
| | | TC | Heap | | | | Non Heap | | | | SLA |
| | | | i | u | c | m | i | u | c | m | |
| Internal Consumer MQTT | QoS "At most Once" | 20 | 250 | 69 | 470 | 3978 | 5 | 51 | 54 | -1 | 2,3915 |
| | QoS "At least Once" | 19 | 250 | 56 | 349 | 3978 | 5 | 53 | 57 | -1 | 2,61 |
| | QoS "Exactly Once" | 20 | 250 | 60 | 399 | 3978 | 5 | 52 | 56 | -1 | 3,06 |

Table 15 depicts MQTT as an internal consumer and OpenDDS, RabbitMQ and Kafka act as internal producers across the diverse QoS levels. This Exactly Once has the higher SLA meaning that OpenDDS, RabbitMQ and Kafka, as internal producers, is the most resource-consumption set. Surprisingly, like DDS, the At Most Once QoS level uses more memory than the remaining levels.

*Table 16 – RabbitMQ Multi-Translation Test Results*

| | | Simultaneously Internal Producers | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | OpenDDS + MQTT+ Kafka | | | | | | | | | |
| | | Threads | Memory | | | | | | | | CPU |
| | | TC | Heap | | | | Non Heap | | | | SLA |
| | | | i | u | c | m | i | u | c | m | |
| Internal Consumer RabbitMQ | QoS "At most Once" | 18 | 250 | 81 | 362 | 3978 | 5 | 51 | 54 | -1 | 2,559 |
| | QoS "At least Once" | 21 | 250 | 96 | 362 | 3978 | 5 | 52 | 55 | -1 | 1,79 |
| | QoS "Exactly Once" | 22 | 250 | 95 | 345 | 3978 | 5 | 52 | 55 | -1 | 1,71 |

Additionally, Table 16 represents RabbitMQ as an internal consumer and OpenDDS, MQTT and Kafka act as internal producers across the diverse QoS levels. Every QoS level uses high quantities of heap memory, despite not reflecting directly in SLA values.

It can be inferred from the data presented in the Multi-Translation Tests that At Most Once is consuming a significant amount of memory, particularly, on internal consumers as OpenDDS and MQTT. This is due to the high volume of data being fast consumed by both internal consumers. It is notable that non-heap memory does not fluctuate significantly, with usage consistently falling between 51 and 53 across all tables. The data presented in each table indicates that the heap memory usage remains below 100, which suggests that the resources are being managed efficiently and that the performance is satisfactory.

# 7 Conclusions

This chapter offers a comprehensive overview of the main aspects of this work, underscores the outcomes achieved through the project development, clarifies the limitations, potential future improvements and concludes with a final personal acknowledgment.

## 7.1 Acomplished goals

1. **Enhance the code quality and documentation of the existing implementation by addressing the lack of essential documentation, eliminating code redundancy, and improving readability.**

This point was successfully achieved, code is now following OOP paradigms, good programming practices and contains documentation.

2. **Redefine QoS mapping between the supported protocols due to gaps from the previously implemented protocols and the necessity from the new one (DDS).**

The mapping between the supported protocols was redefined successfully and it is now assured that every QoS level is working properly.

3. **Expand the application to support the DDS protocol, specifically, production and consuming between every supported protocol and DDS.**

DDS is now supported by PolyglIoT successfully. It is possible to produce externally from a DDS producer or to consume externally from a DDS consumer after a translation mechanism.

4. **Integrate PolyglIoT with the Arrowhead framework and Edge4CPS to scale and simplify PolyglIoT's usage.**

This goal was not fully achieved. Its design is complete, however the implementation is not fully tested.

5. **Evaluate and improve the system performance to verify the system's efficiency.**

PolyglIoT's performance was successfully tested and turned out to be interesting and useful to this project's future contributions.

## 7.2 Limitations and future development

PolyglIoT was designed to be simple, and it is not supposed to make it hard to use it. Nevertheless, improvements related to Edge4CPS communication might be required, PolyglIoT depends on a configuration file to setup its configurations which is great for standalone usage, however when integrated with Arrowhead and Edge4CPS might not be the best solution. An adequate approach is to create a REST API, containing producer/consumer creation endpoints, to allow PolyglIoT internal producers/consumers dynamic instantiation, preventing the application restart to update. Moreover, the multi-protocol tests should be subjected to further investigation, as they offer a promising avenue for the retrieval of significant information about PolyglIoT. These tests have the potential to reveal intricate details about the system's performance across different protocols, which could lead to a deeper understanding and improvement of the PolyglIoT. Additionally, a comprehensive mathematical model should be developed and applied to the performance tests. This model would enable a more precise analysis of the data obtained from the tests, facilitating the identification of key performance metrics and potential bottlenecks.

## 7.3 Final Appreciation

Personally, this project achieved satisfactory results. At first sight, it seemed simpler than what it really was, however diving into the QoS mapping problem and Arrowhead plus Edge4CPS integration revealed to be useful and complex. The resulting solution is robust, efficient and accomplishes the proposed goal.

Unfortunately, due to lack of time, Arrowhead and Edge4CPS were not fully implemented because OpenDDS setup and installation takes about an hour, which means that for each test I needed to wait that prolonged period of time.

# References


AITIA, “Arrowhead Tools”, Retrieved June 10, 2024, from https://aitia.ai/products/arrowhead-tools/

A. Cockburn, (2000), “Writing Effective Use Cases.”, Addison-Wesley, Indianapolis, United States of America

A. Feiszli, “Understanding M2M: Machine-to-Machine Communication”, 2023, Retrieved June 13, 2024, from https://www.netmaker.io/resources/understanding-m2m-machine-to-machine-communication

Apache Kafka, “Documentation”, Retrieved May 17, 2024, from https://kafka.apache.org/documentation

A. Vázquez-Ingelmo; A. García-Holgado; F. J. García-Peñalvo, “C4 model in a Software Engineering subject to ease the comprehension of UML and the software”, *2020 IEEE Global Engineering Education Conference (EDUCON)*, Porto, Portugal, 2020, pp. 919-924, doi: 10.1109/EDUCON45650.2020.9125335.

AWS, “What is Unit Testing?”, Retrieved June 14, 2024, from https://aws.amazon.com/what-is/unit-testing/?nc1=h_ls

B. Cabral, (2023). “*A Scalable Clustered Architecture for Cyber-Physical Systems”,* bachelor’s degree, Instituto Superior de Engenharia do Porto, Porto, Portugal

B. Cabral, P. Costa, T. Fonseca, L. L. Ferreira, L. M. Pinho and P. Ribeiro, "A Scalable Clustered Architecture for Cyber-Physical Systems," *2023 IEEE 21st International Conference on Industrial Informatics (INDIN)*, Lemgo, Germany, 2023, pp. 1-6, doi: 10.1109/INDIN51400.2023.10217924.

C. -H. Lee, Y. -W. Chang, C. -C. Chuang and Y. H. Lai, "Interoperability enhancement for Internet of Things protocols based on software-defined network," 2016 IEEE 5th Global Conference on Consumer Electronics, Kyoto, Japan, 2016, pp. 1-2, doi: 10.1109/GCCE.2016.7800510.

Confluent, “Message Delivery Guarantees”, Retrieved June 16, 2024, from https://docs.confluent.io/kafka/design/delivery-semantics.html

DDS Foundation, “Hoes does DDS Work?”, Retrieved May 27, 2024, from https://www.dds-foundation.org/how-dds-works/

DDS Foundation Wiki, “User Experiences”, Retireved June 16, 2024, from https://www.omgwiki.org/ddsf/doku.php?id=ddsf:public:guidebook:03_user:start

Eclipse Arrowhead, “A framework and implementation platform for SoS, IoT and OT integration”. Retrieved June 11, 2024, from https://arrowhead.eu/eclipse-arrowhead-2/

EMXQ Team, “MQTT to Kafka: Benefits, Use Case & A Quick Guide”, 2024, Retrieved June 16, 2024, from https://www.emqx.com/en/blog/mqtt-and-kafka

H. Akhtar, “NFRs: What is Non Functional Requirements (Example & Types)”, 2023. Retrieve June 11, 2024, from https://www.browserstack.com/guide/non-functional-requirements-examples

H. Derhamy, J. Eliasson, J. Delsing, P. P. Pereira and P. Varga, "Translation error handling for multi-protocol SOA systems," 2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA), Luxembourg, Luxembourg, 2015, pp. 1-8, doi: 10.1109/ETFA.2015.7301473

HiveMQ Team, “What is MQTT Quality of Service (QoS) 0,1 & 2? – MQTT Essentials: Part 6”, 2024, Retrieved June 14, 2024, from https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/

IBM, “What is the Internet of Things (IoT)?”, Retrieved June 1, 2024, from https://www.ibm.com/topics/internet-of-things

I. Jawhar, N. Mohamed, & J. Al-Jaroodi, “Networking architectures and protocols for smart city systems”. J Internet Serv Appl 9, 26 (2018). https://doi.org/10.1186/s13174-018-0097-0

Katalon, “What is Integration Testing? Definition, How-To, Examples”, Retrieved June 14, 2024, from https://katalon.com/resources-center/blog/integration-testing

MQTT, “The Standard for IoT Messaging”, Retrieved May 17, 2024, from https://mqtt.org/

M. Glinz, (2007, October). On non-functional requirements. In 15th IEEE international requirements engineering conference (RE 2007) (pp. 21-26). IEEE.

M. Khatri, “Apache Kafka Use Cases: When To Use It? When Not To Use?”, 2023, Retrieved June 16, 2024, from https://www.peerbits.com/blog/apache-kafka-use-cases.html

M. Rouse, “Protocol Converter”, 2017, Retrieved June 16, 2024, from https://www.techopedia.com/definition/9004/protocol-converter

N. Lukman, “CoAP, MQTT, AMQP, XMPP & DDS: Which Protocol Should You Choose for IoT?”, 2021, Retrieved June 16, 2024, from https://www.nexpcb.com/blog/different-data-protocols-which-one-to-choose

N. Naik, "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP" *2017 IEEE International Systems Engineering Symposium (ISSE)*, Vienna, Austria, 2017, pp. 1-7, doi: 10.1109/SysEng.2017.8088251

OpenDDS, “The OpenDDS Project”. Retrieved June 10, 2024, from https://github.com/OpenDDS/OpenDDS/blob/master/java/README

OpenDDS, “Conditions and Listeners”. Retrieved June 10, 2024, from https://opendds.readthedocs.io/en/latest/devguide/conditions_and_listeners.html

Oracle, “*Java SE Documentation*”, A, Retrieved June 10, 2024, from https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-2.html

Oracle, “Java SE Documentation”, B, Retrieved June 10, 2024, from https://docs.oracle.com/en/java/javase/11/docs/api/java.management/java/lang/management/MemoryUsage.html

Oracle, “Java SE Documentation”, C, Retrieved June 10, 2024, from https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentNavigableMap.html

Oracle, “What is IoT?”. Retrieved June 10, 2024, from https://www.oracle.com/internet-of-things/what-is-iot/

Oracle, “Java Native Interface Overview”. Retrieved June 13, 2024, from https://docs.oracle.com/javase/8/docs/technotes/guides/jni/spec/intro.html#java_native_interface_overview

P. H. M. Pereira; G. Cainelli; C. E. Pereira; E. P. Freitas, “An Interoperability Middleware for IIoT”, 2023 IEEE 32nd International Symposium on Industrial Eletronics (ISIE). June, 2023, 1-6. https://doi.org/10.1109/ISIE51358.2023.10227975

P. S. Costa, (2023), “Multiprotocol Middleware Translator for IoT”, bachelor’s degree, Instituto Superior de Engenharia do Porto, Porto, Portugal

QRA Corp, "Functional vs. Non-functional Requirements.", Retrieved June 10, 2024, from https://qracorp.com/functional-vs-non-functional-requirements/.

RabbitMQ, “AMQP 0-9-1 Model Explained”, Retrieved June 16, 2024, from https://www.rabbitmq.com/tutorials/amqp-concepts

RabbitMQ, “RabbitMQ Documentation”, Retrieved May 17, 2024, from https://www.rabbitmq.com/docs/

RTI Core Libraries, “PUBLISH_MODE QoSPolicy (DDS Extension)”, Retrieved June 14, from https://community.rti.com/static/documentation/connext-dds/current/doc/manuals/connext_dds_professional/users_manual/users_manual/PUBLISH_MODE_QosPolicy__DDS_Extension_.htm

T. M. Tukade; R. Banakar, “Data transfer protocols in IoT – An overview”, Int. J. Pure Appl. Math, 2018, 118.16: 121-138.

W. Y. Liang; Y. Yuan; H. J. Lin, “A performance study on the throughput and latency of zenoh, mqtt, kafka, and dds”, arXiv preprint arXiv:2303.09419, 20