Multiprotocol Middleware Translator for IoT

ISEP

2022 / 2023

1200911 Pedro Silva Costa

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Multiprotocol Middleware Translator for IoT

ISEP

2022 / 2023

1200911 Pedro Silva Costa

Degree in Informatics Engineering

July 2023

ISEP Advisor: **Luís Miguel Moreira Lino Ferreira**

“We’re working towards a goal that’s nonexistent, but we just keep believing.”

H. Williams

Acknowledgments

I would like to firstly give my thanks to Professor Nuno Miguel Vieira Morgado, for mentioning to me how my future internship advisor, Professor Luis Miguel Moreira Lino Ferreira, was looking for students to work in the Ferrovia 4.0 Project, and how he thought I was a good fit for it.

Of course, I would also like to thank Professor Luis Ferreira himself, for the opportunity, help, and patience he gave me during this project.

I also have to thank two of my research colleagues from ISEP, Tiago Carlos Caló Fonseca and Bernardo Magalhães Amaral Cabral for all the fun moments and work we shared, but especially their availability and efforts in supporting me with the project and in easing me into the team.

And of course, my parents and my past self for providing me with the opportunity to be here.

Abstract

Internet of Things (IoT) is a continuously growing market, consisting of a network of devices that communicate with each other in real-time, commonly used in areas like Home Automation and Autonomous Driving.

These devices utilize special communication protocols to communicate with each other, such as MQTT and Kafka. However, there is no protocol that best fits every situation, and so developers usually must pick one or a select few that best fit their requirements.

The development of this project aimed to develop a solution for this problem, where developers could use our application instead of having to compromise with an undesirable protocol.

The resulting Middleware application is capable of redirecting messages between the MQTT, Kafka and AMQP protocols, including support with Arrowhead for acquiring the address of brokers, configurability of clients based on their Java libraries, support for all three Message Delivery Semantics levels regardless of protocol, and an architecture capable of supporting new protocols.

Keywords (Theme): IoT, Middleware, Translator

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

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
COAP	Constrained Application Protocol
FR	Funcional Requirement
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
MQTT	Message Queuing Telemetry Transport
NFR	Non-Functional Requirement
OPC-UA	Open Platform Communications Unified Architecture
QoS	Quality of Service
REST	Representational state transfer
SSL	Secure Sockets Layer
XMPP	Extensible Messaging and Presence Protocol

1 Introduction

This Section contains a brief introduction to the main concepts of this report, such as the problem this project was designed to solve, and the technologies used for it.

1.1 Project Context

IoT or "Internet of Things" can be described as a network of physical devices that communicate with each other in near real-time with no need for human intervention. These systems are commonly used in areas from Home Automation to Healthcare, among many others. The IoT market is expected to continue steadily growing, from a market size of 457$ billions in 2020, to 11680$ billions in 2030 (Al-Sarawi et. al., 2020).

These devices communicate with each other, many times by periodically transmitting sensor readings where simple device-to-device messaging protocol, like REST, are not scalable enough. This is what the IoT Messaging Protocols are for. To provide a messaging platform for multiple devices of the same system to communicate with each other, be them producing or consuming messages. However, there are different open or proprietary solutions, and no universal protocol every device can communicate in.

The MQTT protocol is one of the most used for the communication between IoT devices and the cloud/edge, due to its small footprint and overhead. There are also many cases were the REST and COAP technologies are being used. But to communicate between edge and cloud modules, e.g., for data analysis and visualizations, these technologies are not adequate and require more powerful solutions, like Kafka and AMQP.

As explained by (Naik, 2017), choosing an effective protocol for all IoT applications can be quite a challenging task, since it depends on the IoT system and its messaging requirements. These protocols usually all have a situation they fit best in, but none of them could be the best fit for every IoT system.

For example, the authors in (Nam & Choi, 2022) chose both MQTT for light-weight messaging and Kafka for the heavier images/videos, the authors in (Happ et. al., 2017) utilized AMQP, XMPP and ZeroMQ, while the authors in (Drahoš et. al., 2018) used OPC-UA in an industrial context.

Therefore, in some systems it may be more efficient and a requirement of the existing infrastructure to use multiple protocols at once.

This project was supported by the Ferrovia 4.0 Project (Ferrovia, 2020), in which it was identified a need for message translation for the messages being sent from the sensors deployed, along the railway and in the train, using the MQTT protocol, to communicate with a gateway/edge device in a wagon and the Main Data Center expecting to receive them through the Kafka protocol.

1.1.1 Objectives

Based on the Ferrovia 4.0 requirements and a vision for a Middleware product, the following objectives for this project were decided on:

1. **Develop an application capable of connecting devices who communicate in different messaging protocols.**

In this application, users should be able to easily connect two devices of different messaging protocols. Assuming that the user has a producer and a consumer of different protocols, the messages produced by the producer should successfully arrive at the consumer, provided that the Middleware was correctly configured.

2. **Support broker address configuration with Arrowhead**

Through the usage of the Arrowhead Framework, the process of choosing and finding which brokers to connect to should be automatic. Any changes can be altered directly in the Arrowhead System, without the need to change the address in every single device.

3. **Develop a simple and clear interface that can easily support additional translation rules, as well as new protocols.**

By working with simple interfaces, any user experienced with the protocols at hand should be able to modify the application to allow for additional behavior before receiving a message, a new property, or even the implementation of an entirely new protocol, without having to worry about compatibility with the others.

4. **Support advanced producer/consumer configuration.**

The libraries of the protocols the application intends to support offer several configuration options, such as setting a topic, connection timeout, queue, among many others. The application should provide an easy way for the user to define these, as well as extra configurations, like Message Delivery Semantics (QoS).

1.1.2 Approach

To answer these problems, an application was developed capable of supporting, for now, message translation between the MQTT, Kafka and AMQP protocols, but it has been designed in a way that supporting any other protocol in the future should be simple. With the Middleware, users can easily support additional protocols in their system, through internal Java producers and consumers that can be configured beyond what their libraries allow.

To develop this application, study focused on the messaging protocols it needed to support and their common features. Research was made on the Arrowhead Framework and how it could be used, as well as what could be learned from applications built with a similar purpose. The application in Java was developed to fix all those issues, in a way that it could support the different protocols regardless of different implementations and slightly different logic.

1.1.3 Contributions from this project

The application that resulted from this work is an open source (Middleware Repo, 2023) solution, which provides a way for any user aware of the advantages of using multiple protocols to be able to use them as an alternative to either a single protocol, or proprietary software. Figure 1 illustrates the role of the Middleware in a system with more than 2 protocols.

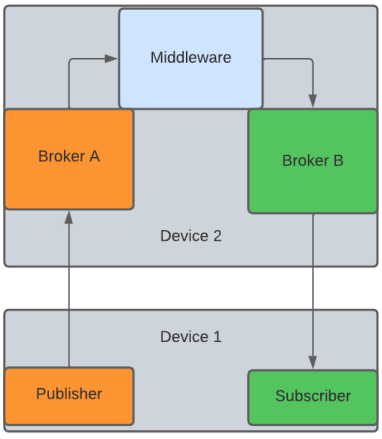

Figure 1 Middleware Use-Case

Users who understand the concepts of IoT messaging can extend the already built application architecture to support any protocol they'd want, instantly having a way to redirect messages between that protocol and those already implemented, without the need of any additional work, other than code for its producers and consumers.

Furthermore, the code for already built producers and consumers can be altered to support additional translation logic the user may want, like, for example, attaching a prefix to every received message.

Through Arrowhead, the application uses a valuable IoT framework perfect for syncing multiple devices and providing them with a common service. Changing an address for the clients to connect to, be it the user's producers and consumers or the middleware itself, does not require changing every single property file, it only needs a simple request to the Arrowhead's Service Registry.

1.1.4 Project Planning

The following Gantt Diagram presents the development process from the beginning of the project to the completion of report writing.

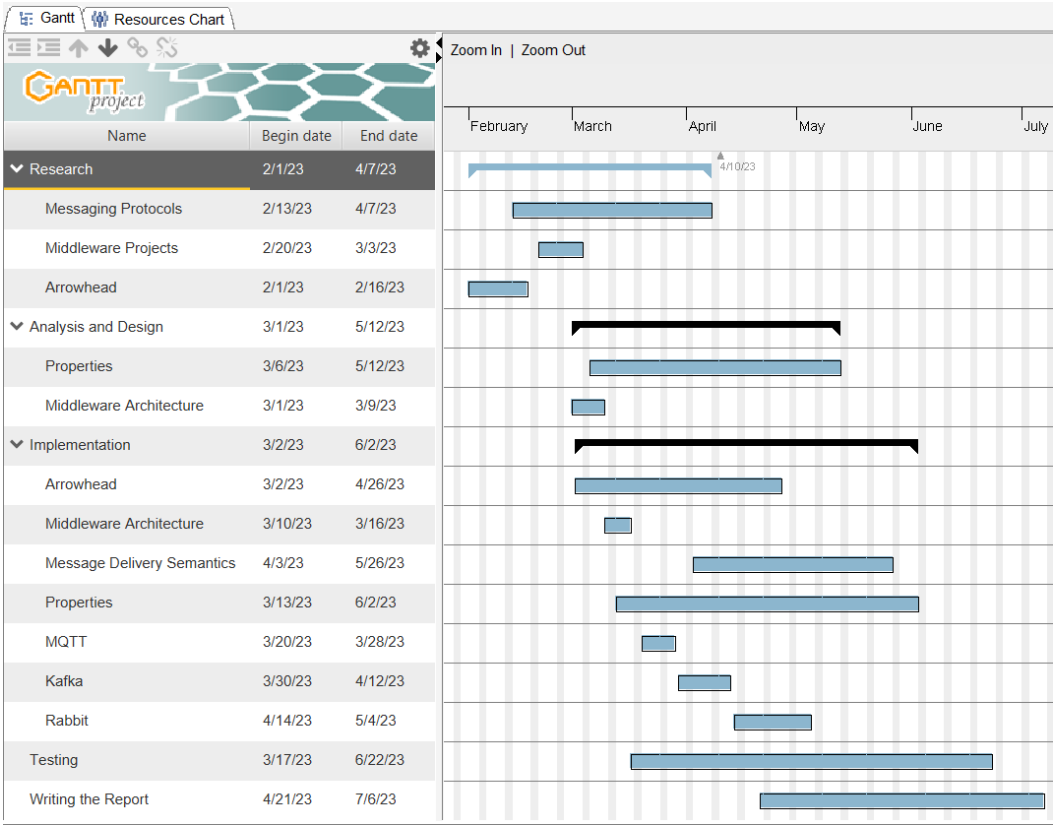

Figure 2 Gantt Diagram

As evidenced by Figure 2, the project first started with a "Research Phase" where we looked to have a better understanding of the Arrowhead Framework and the components of it we were going to use, such as the Service Registry and the Orchestrator. As well as the Messaging Protocols of IoT like MQTT, Kafka and AMQP, and other Middleware Projects with a purpose similar to this one.

Roughly after the Research period, we performed an Analysis and Design of the Middleware in general, especially its Properties file, due to its importance and the fact that it is the main point of interaction with the user. After the Design, we proceeded with the Implementation phase, attempting to utilize Arrowhead, setting up the planned Middleware Architecture, supporting each of the planned Messaging Protocols, and the Properties file, which took the most time due to the alternatives we had to consider, along with Arrowhead, due to some complications in utilizing the framework due to some outdated libraries, unclear instructions and running issues.

After Implementation, we spent the rest of the time mostly finalizing tests, especially System Tests, since the application was mostly done, wrote this report, and perfected the System itself by cleaning up methods, declaring Constants, etc.

1.2 Report Structure

Beyond the introduction, this report contains 6 other chapters.

Chapter 2 – State of the Art: This chapter features research on the IoT protocols the application supports and their particularities, a quick overview of Arrowhead, and other projects with a similar functionality of this Middleware.

Chapter 3 – Analysis: This chapter holds a preliminary analysis of the problem at hand, resulting in functional and non-functional requirements, User Stories, and a domain model.

Chapter 4 – Design: This chapter will present a detailed overview of the components that are part of the Middleware, and the ones the Middleware interacts with, as well as discussing some of the design choices and the processes associated with the User Stories in Chapter 3.

Chapter 5 – Implementation: This chapter contains a deep-dive description of how the Middleware was developed and how it functions internally, particularly how its classes interact with each other, the procedures executed on setup, and how it communicates with the exterior.

Chapter 6 – Tests: This chapter contains an overview of the tests that were developed to test the system and attempt to prove everything was running as expected.

Chapter 7 – Conclusion: The Conclusion chapter consists of a discussion of potential future developments for the application, and an appreciation of the work produced compared to the initial goals.

2 State of the Art

After this chapter, the reader will have a deeper understanding regarding some of the most used messaging protocols and how this application compares to others.

2.1 Existing technologies

IoT systems can communicate using multiple messaging protocols. For this first version of the protocol translator, support was included for Message Queueing Telemetry Transport (MQTT), Kafka, and (Advanced Message Queuing Protocol) AMQP protocols, since these are all required by the Ferrovia 4.0 project and are quite distinct between them and best appliable in different situations.

OPC-UA was also pondered for support due to its widespread usage in industrial applications but was ultimately considered unnecessary since the OPC-UA PubSub configuration supports the use of MQTT and AMQP to broadcast messages, making it possible to just consider the devices communicating in OPC-UA PubSub as MQTT or AMQP (OPC-UA Pub/Sub, 2023).

2.1.1 Messaging Protocols

Messaging Protocols are the backbone of IoT applications. They standardise data transmission with the intent of establishing communication between devices (Tukade & Banakar, 2018).

Some IoT Protocols, like OPC-UA or HTTP rely on a direct connection between the producer (usually a sensor transmitting its data) and the consumer (an application interested on the data from the sensor) of a message. This results in a high complexity between connections or any attempts of scaling and adding new producers/consumers, requiring the handling of individual direct connections and addressing.

Other protocols like MQTT, Kafka or AMQP, do not require a direct connection between every client, instead connecting everything to a broker, or a cluster of them, allowing for consumers to specify which type of messages they would like to receive, and redirecting it to them once the Producers have sent them. Although this introduces a single point of failure and a potential performance bottleneck, it also simplifies the connection between devices, requiring only for clients to know the broker's address and a stream identifier, like a topic or queue, to establish a connection with another client, allowing for multiple consumers to receive messages from multiple producers with minimal setup and the possibility for advanced routing.

2.1.1.1 MQTT

MQTT is a broker-based publish/subscribe messaging protocol. Its messages minimize transport overhead, and its clients require minimal resources (MQTT Page, 2023), drastically reducing network traffic and improving its efficiency, making it a popular choice in less powerful network or devices.

Furthermore, brokers in MQTT 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 in the event of a consumer, receive all queued messages they may have missed.

In MQTT, messages are sent to topics in a broker by “publishers”, which are then redirected to “consumers” on the other end. By default, topics can be created by publishers as they receive their first message, however, messages will be lost if there are no subscribers on the other end, or once a subscriber acknowledges receiving that message.

Eclipse Mosquitto is an implementation of an MQTT-based broker developed by the Eclipse Foundation (MQTT Page, 2023). Its Java library allows developers to easily implement new producers and consumers and configure their connections with the broker.

2.1.1.2 Kafka

Kafka is a scalable, durable, and fault-tolerant protocol based on the publish-subscribe system, built to handle large amounts of data.

Topics in Kafka are organized into partitions, which provide load balancing by splitting topics into multiple smaller divisions, which can be handled by other less busy brokers. Fault tolerance is also ensured by replicating each of these partitions across the cluster, meaning that in the event of one broker failing, the others can still handle the clients communicating with that topic and ensure no message is lost.

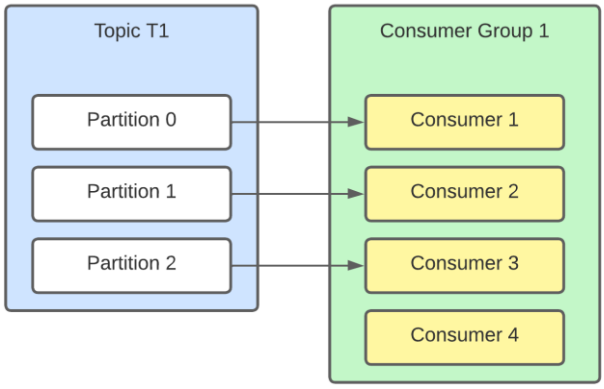

Figure 3 Kafka's Consumer Groups

Through consumer groups, Kafka also provides horizontal scaling for message processing. In them, one consumer takes the role of assigning each of the consumers with a subset of the partitions the group is subscribed to. This, however, implies that there will be idle consumers if there aren't enough partitions (as exemplified by Figure 3), but allows for a more efficient load balancing of message handling from the consumer side, since messages will be evenly distributed for each partition.

Although Kafka can store its messages, Kafka Consumers know not to process them again by storing their current offset for each partition they are consuming from. Once a Consumer reads a message from offset 10, for example, it will update its offset and request messages starting from offset 11.

Offsets are shared among the consumer group, so that when a consumer eventually fails, the consumer that will be left processing its partitions will be able to continue its progress.

While in MQTT the broker redirects messages to its subscribers, Kafka uses instead a "pull-based" system. Consumers in Kafka choose how many and how frequently they'd like to receive their messages, by sending a "poll" request to the broker, returning with messages for their partition starting from their offset up to the number specified.

2.1.1.3 AMQP

Unlike the Protocols discussed so far, the AMQP protocol uses an "Exchange/Queue" type of messaging, instead of topics. Messages are published to Exchanges, which then, decided by its Routing Key, are redirected to Queues from which consumers can read directly. These queues can store messages indefinitely, until a consumer has finally read them. If the exchange cannot route a message to any queue, the message will be lost.

Exchanges can be bound to Queues with a direct Routing Key. For example, binding the exchange “exchange 1” with the routing key “ISEP” to “queue 1”, would have it so messages sent to “exchange 1” would need that exact routing key to reach the queue.

AMQP allows for advanced queue mapping with wildcards in the routing key. The character “*” implies a word, the character “#” one or more words. Words are any pattern of characters separated by a dot (.).

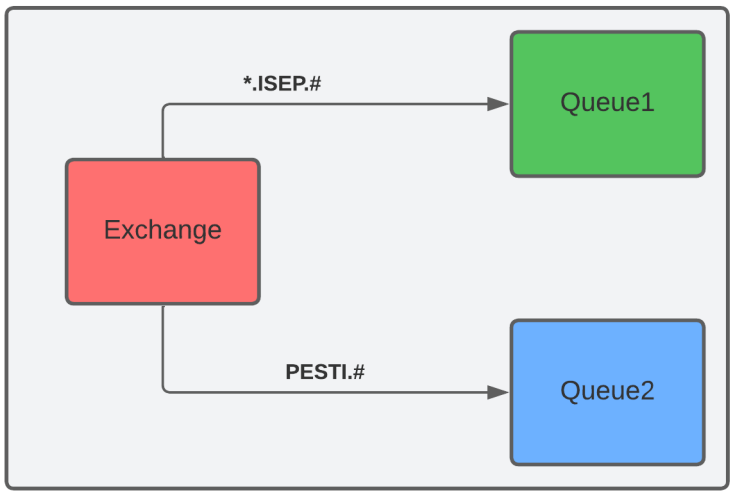

Figure 4 AMQP queue mapping example

Taking Figure 4 as an example, a message produced with a Routing Key of “ISEP.INFORMATIC” would be unrouteable, unless it contained a word before “ISEP” to match with the “*” character. In that case, the message would be routed to Queue1. On the other hand, any message with a routing key starting with “PESTI” on this exchange would be routed to Queue2.

RabbitMQ

RabbitMQ is an implementation of an AMQP-based broker developed by VMware (Dossot, 2014). RabbitMQ has a Java library to easily implement new producers and consumers and configure their connections with the broker.

RabbitMQ offers a free management API for monitoring and easy queue manipulation. Through it, applications can connect and obtain information, like, for example, available queues, number of connected consumers, or number of messages waiting to be processed in a queue. Figure 5 shows an example of 2 graphs of the dashboard, which display the status of all messages in all the broker’s queues, and the most recent rate of messages arriving at the broker.

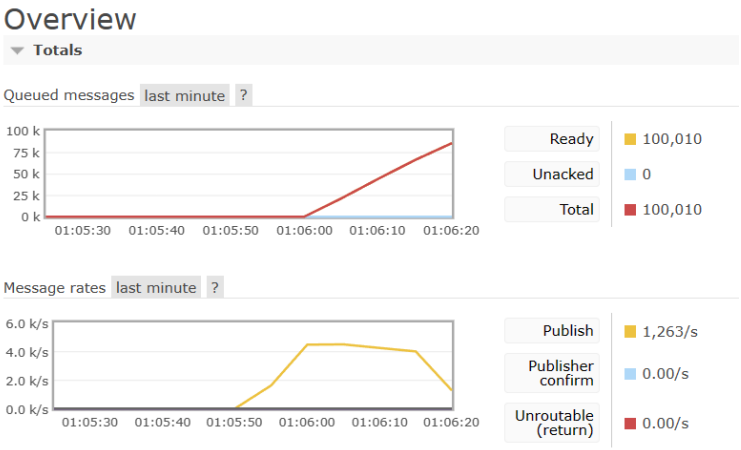

Figure 5 RabbitMQ Management API Snippet

2.1.2 Arrowhead Framework

Eclipse Arrowhead is a service-oriented framework designed for facilitating the communication, security, and discovery of IoT devices (Varga et. al., 2017).

Its Core Services (Orchestrator, Service Registry, and Authorization), answer common IoT challenges like having systems who provide a service be known to services who want to consume them, having service consumers discover available services to consume, and having the system provider verify if the consumer is authorized to consume from a service provider. The identity of its systems is verified through SSL Certificate Trust Chains.

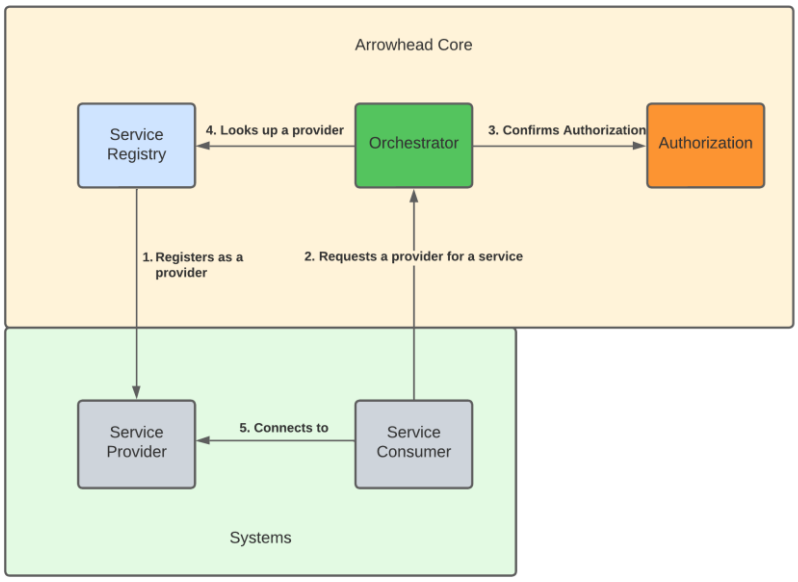

Figure 6 Arrowhead Core Interactions

The interactions between their Core Services can be described by Figure 6.

In the Service Registry, service providers can register themselves in the system, specifying the name of the service they provide, and where service consumers can connect to consume their service, through address and port.

In the Authorization, a System Operator can define what services certain service consumers have access to.

Service consumers utilize the Orchestrator to request a service provider for a specific service they wish to consume. The Orchestrator will first confirm with the Authorization service that the requesting consumer is authorized to consume the service. If so, it will query the Service Registry for a previously registered provider for that service and return to the consumer information about the provider so they may connect.

2.1.3 Message Delivery Semantics

Message Delivery Semantics in Kafka/AMQP, or QoS in MQTT is a way of specifying the reliability of message delivery. These semantics can be defined in 3 levels:

0. Messages are delivered “At Most Once”
1. Messages are delivered “At Least Once”
2. Messages are delivered “Exactly Once”

Each of the three chosen protocols have similar capabilities of these semantics, except for “Exactly Once”.

2.1.3.1 MQTT

This information was extracted from (MQTT QoS, 2023).

At Most Once

In “At Most Once”, the sender delivers the message without expecting an acknowledgment of message delivery. This means the message will only be sent once, and either the receiver processes it then, or the message is lost.

At Least Once

In “At Least Once”, the sender publishes the message, stores it, and waits for an acknowledgement from the receiver. If the sender does not receive an acknowledgement in a reasonable timeframe, the sender will attempt to resend the message, allowing in some cases for the message to be duplicated. This process is illustrated by Figure 7.

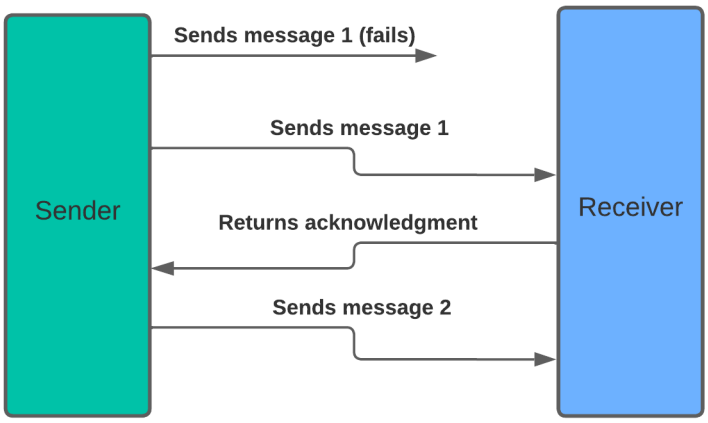

Figure 7 At Least Once in MQTT

In “Exactly Once”, like in “At Least Once”, the sender publishes the message, stores it, and resends if no acknowledgment is received.

Once an acknowledgment is received, the sender will respond with a PUBREL packet, and wait for a PUBCOMP packet from the receiver. The receiver will send that packet once it’s finished processing the message. The receiver will also store a packet identifier for the message it’s receiving until it does so, preventing it from processing the message again if the original sender thinks the original message was lost and decides to resend it again.

2.1.3.2 Kafka

Producers

Kafka’s “At Most Once” and “At Least Once” behavior for their producers is identical to the processes described above for MQTT.

Regarding “Exactly Once”, Producers can request acknowledgement from the broker that messages were received and successfully replicated. If the Producer does not receive an acknowledgment in time and resends the message, it will do so with idempotency, which the broker will interpret to overwrite and not duplicate the message.

Consumers

In “At Most Once”, Kafka Consumers will read the set of messages, update their offset, and then process the message. This means that if the Consumer crashed after updating its offset but before processing the message, the consumer that would take over processing would not receive the un-processed messages and would instead continue with the next offset.

For “At Least Once”, Kafka Consumers will read the set of messages, process them, and then update their offset. This way, if the consumer were to crash after processing the message but before updating the offset, the consumer that would take over would assume the messages were not processed yet and request the same offset again.

Kafka’s Consumers require the use of an additional Kafka component to support “Exactly Once”, such as the Kafka Transactions API (for 3+ brokers), Kafka Connect, or Kafka Streams.

2.1.3.3 AMQP

“At Most Once” is identical to all other protocols. Messages are sent by the sender without expecting an acknowledgement.

Through Consumer Acknowledgments and Publisher Confirms (Rabbit QoS, 2023), RabbitMQ offers an implementation of “At Least Once” like in MQTT and Kafka, where the sender of the message will wait for an acknowledgement from the receiver before sending the next message. If no acknowledgement is received, it will attempt to resend the message, allowing for the message to be duplicated.

However, this component is not enough to support an implementation of “Exactly Once”, as even if the receiver finally sends an acknowledgement that the message was received, it could have, for example, happened while the sender had already attempted to resend the message thinking the message was lost, allowing the receiver to duplicate its processing. This situation is illustrated by Figure 8.

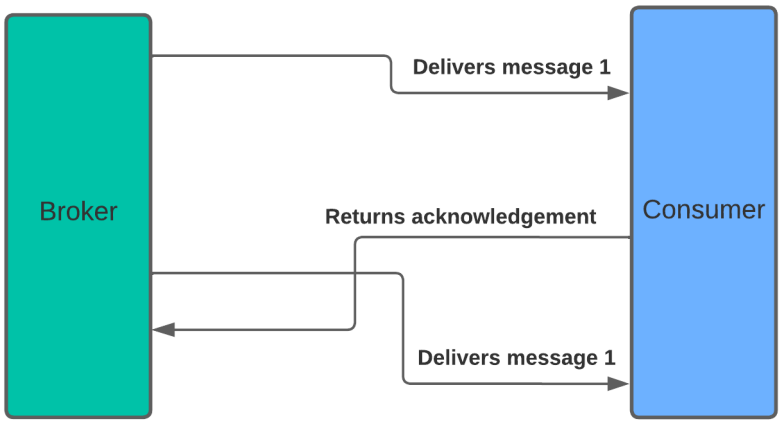

Figure 8 AMQP Message duplication

2.2 Related projects

Software like Microsoft's IoT hub already contain some protocol translation mechanisms and a sturdy architecture with safety in mind, but it wouldn't be possible to use their application to, for example, automatically redirect any MQTT message to a Kafka broker, or to quickly alter translation logic or support a new protocol by directly changing the code, as it was not built as a message broker or data streaming platform.

These next sub-sections, will provide a more in-depth overview of some projects we could have taken advantage of for this Middleware, and some of their downsides.

2.2.1 Kafka Connect

Kafka Connect (Connect Documentation, 2023) is capable of translating data from and to Kafka, providing a system that shares Kafka's security and scalability, as well as a REST interface for managing connectors.

The code available at (Kafka Connect MQTT, 2023) provides an open-source implementation of an MQTT Connector to both redirect messages to Kafka from MQTT and from MQTT to Kafka. Analyzing its configuration options, however, exposes some of the issues in using Kafka Connect itself, mainly how it considers every Consumer/Producer (Sink/Source) connection as a separate instance, implying that applications deployed in Connect are stand-alone, and do not benefit from a centralized custom application to control them, for purposes of advanced configurability and routing, or for utilizing Arrowhead Services for the discovery of brokers, even though they do increase reliability and scalability through Kafka Connect itself.

It also means that to translate messages between protocols that aren't Kafka, must use Kafka as an intermediary anyways.

2.2.2 Stream Processing Frameworks

Software like Flink (MQTT Flink Connector, 2023), Samza (Samza Connectors, 2023), or Spark Streaming (MQTT Spark Connector, 2023) are all Stream Processing Frameworks developed by Apache, providing an already implemented Kafka integration with both the role of a Consumer and a Producer, and a highly scalable and reliable framework for deploying eventual custom-made applications by the User, like custom connectors.

These Frameworks are commonly used in Use-Cases of Event-Driven Applications, Data Analytics, or Data Pipeline, similar to our case (Apache Flink Use Cases, 2023).

Similarly to Kafka Connect however, there is no centralized program that could control them, reutilize instances of Producers and Consumers, or connect to brokers based on addresses fetched through Arrowhead, and deploying one anyways would be no different that deploying the application through another platform.

3 Analysis

This chapter contains a preliminary analysis of the problem at hand, analyzing its requirements and the developed model.

3.1 Problem domain

As represented in Figure 9, the Broker(s) act as the intermediary between the User's Producer(s)/Consumer(s) and the ones internally created by the Middleware to connect them. Any Stream provided by this Middleware will likely have a broker of one protocol connecting the User's Producer and the Middleware's Consumer, and another connecting the Middleware's Producer and the User's Consumer.

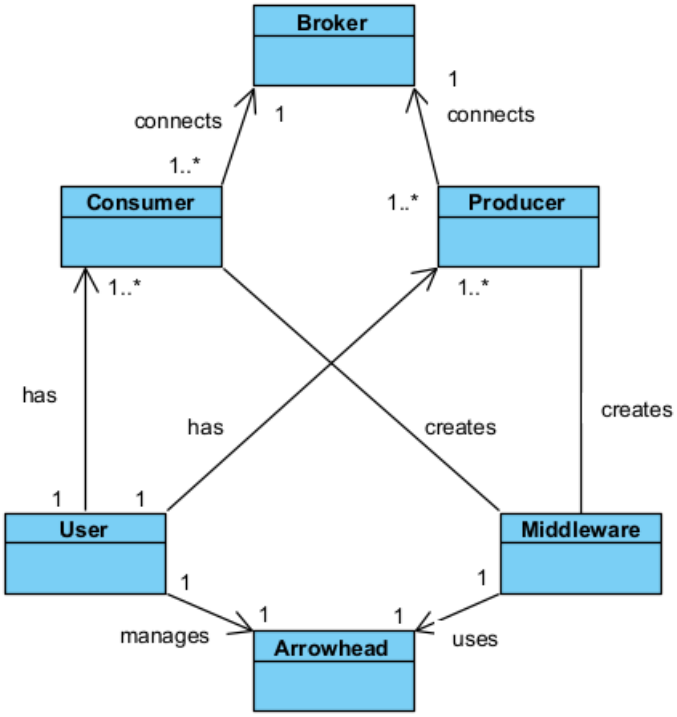

Figure 9 Domain Model

Before creating its clients, the Middleware will optionally attempt to use Arrowhead's Orchestrator to acquire the Broker's address, provided it had been previously registered at the Arrowhead Service Registry. This registration must be performed by the User, communicating with the Service Registry to create a service provider for each protocol. The application will only use the address, port and the name of the service provided, so only these

attributes are essential. The name of the registered service should be identical to the pattern the Middleware uses to identify that protocol, in, for example, its properties file.

Table 1 Glossary

Arrowhead	IoT framework which provides 2 essential core services for the dynamic definition of the brokers address for the application to register to: The **Service Registry**, where the **User** can register its **Brokers** (and their addresses) as a service provider, and the **Orchestrator**, where the **Middleware** and other **User Producers/Consumers** can request a service provider (Broker address previously registered)
Broker	Implementing a protocol, acts as the intermediary between Producers and Consumers. It is responsible for receiving and storing messages sent by Producers, and delivering them to the Consumers
Client	A running instance of a Producer or Consumer.
Consumer	A device/application (client) associated with a certain protocol who reads messages from a broker, previously sent by its producers
Producer	A device/application (client) associated with a certain protocol which sends messages to its broker

3.2 Requirements

3.2.1 Functional Requirements

FR 1. The system should support translation between multiple messaging protocols, at least:

- MQTT
- Kafka
- AMQP

The message's contents should be preserved during translation.

FR 2. The application should load the broker's connection information from its properties file and from Arrowhead.

FR 3. The user should be able to specify a Message Delivery Guarantee level for both the Producer and Consumer, regardless of whether it's directly supported by the protocol.

FR 4. The application's producer should optionally use the internal consumer's topic, removing part of the topic's name if necessary.

FR 5. The application's producers and consumers should support advanced configurability as supported by their Java libraries.

3.2.2 Non-Functional Requirements

3.2.2.1 Usability

NFR1. The properties file should be designed from the client's perspective, reading a list of user devices (producers and consumers), and between which to redirect messages.

3.2.2.2 Performance

NFR2. The application must translate messages efficiently, taking advantage of the available CPU capabilities, like multicores.

3.2.2.3 Scalability

NFR3. The System should be capable of scaling using multiple message broker in different computers.

3.2.2.4 Interoperability

Considering that this project's purpose goal is to connect multiple devices who communicate in different languages, Interoperability is inherent to the application.

3.2.2.5 Portability

NFR4. The application should be able to run across multiple platforms, particularly containers. This requirement is associated with the Ferrovia 4.0 Project, where Kubernetes is utilized to deploy applications.

3.3 Actors

In this application, there is a single Actor: The User.

It is the User's job to set up its own devices, the publishers and consumers, before assigning them a translator in the application.

They are also responsible for making sure all the brokers of the protocols in use are running and accessible, as well as setting up services in the Arrowhead framework, if they decide to use it.

They must also configure any security detail, like specifying the credentials for the middleware's MQTT clients to connect or providing an Arrowhead certificate necessary for proving the identity of the Middleware system.

3.4 User Stories

After analysing the system's requirements, we arrived on the following User Stories:

Table 2 User Stories

User Stories	Acceptance Test
As a User I want to translate messages between different protocols	The application must be able to redirect messages from the User's Producer to the User's Consumer running on different protocols.
As a User I want to specify the address of my brokers	The application must connect its Producers/Consumers to the brokers defined by the User in either the Properties file through the Arrowhead Framework.
As a User I want for the translation to support different levels of Message Delivery Guarantees.	The application's Producers/Consumers should implement different levels of Message Delivery Guarantees, regardless of the protocol implementation
As a User I want messages to be redirected based on the original topic I produced them to.	The application's Producers should be able to use the Consumers topic the message originated from, instead of their own. They should also allow for pattern removal in topics (for example, "toMqtt/temperature" becomes "temperature")
As a User I want to configure the application's Producers/Consumers as much as their libraries allow too.	The application's Producers/Consumers should be able to interpret the User's configuration to customize their connection with their broker based on the settings their libraries allow.

4 Design

This chapter contains the decisions based on the requirements and analysis made in the chapter above.

4.1 Architectural Design

In this section, we will discuss the design of the application featuring diagrams built with the C4 model (Vázquez-Ingelmo et. al., 2020).

4.1.1 Level 1

4.1.1.1 Implementation

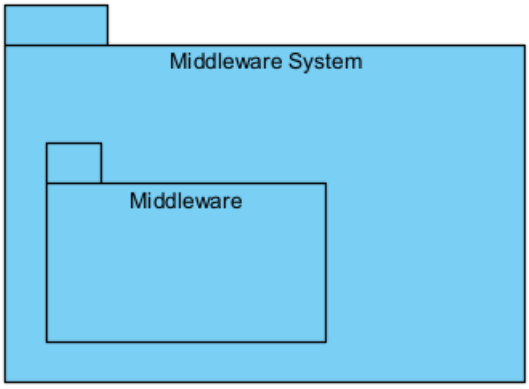

Figure 10 Level 1 Implementation Diagram

The Middleware System is only composed of a single Middleware Component. Everything else is external to it.

4.1.1.2 Logical View

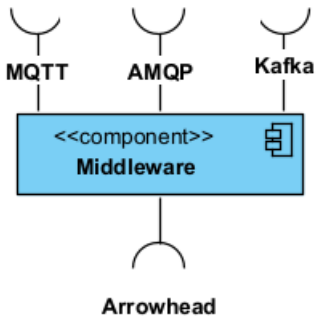

Figure 11 Level 1 Logical View

- **Middleware**: Represents the main component, containing all the essential instances of the producers and consumers, who utilize the supported brokers to offer a translation service between them.
- **MQTT, AMQP, Kafka**: This connection is used by the Middleware's producers/consumers to connect and produce/consume messages to/from.
- **Arrowhead**: This connection is used by the Middleware before creating its producers and consumers, to obtain, if necessary, the address where the supported brokers are available at.

4.1.1.3 Use-cases

Based on the User Stories defined in the previous chapter, we arrived on the following Use-Cases Diagram:

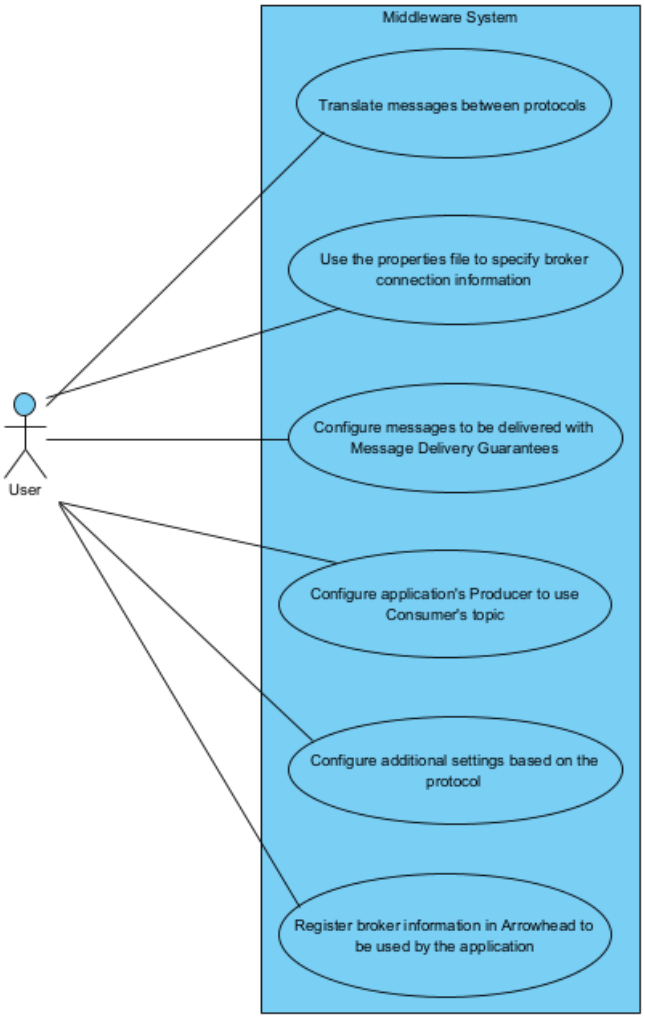

Figure 12 Use-cases Diagram

4.1.1.4 Process

Table 3 Use Case 6 - Connect to the User's Brokers with Arrowhead

Use Case 6	***Register broker information in Arrowhead to be used by the application***
Description	The user intends for the Middleware's Producers and Consumers to connect to the broker's address, obtained through the Arrowhead infrastructure.
Actor(s)	User
Preconditions	1. Have access to the Middleware. 2. Have access to the Arrowhead Service Registry and Orchestrator. 3. Have access to at least one broker
Postconditions	1. Publishers and Consumers from the Middleware connect to the brokers defined in the Arrowhead infrastructure.
Necessary data	• Broker address • Arrowhead Service Registry address • Name of protocols involved

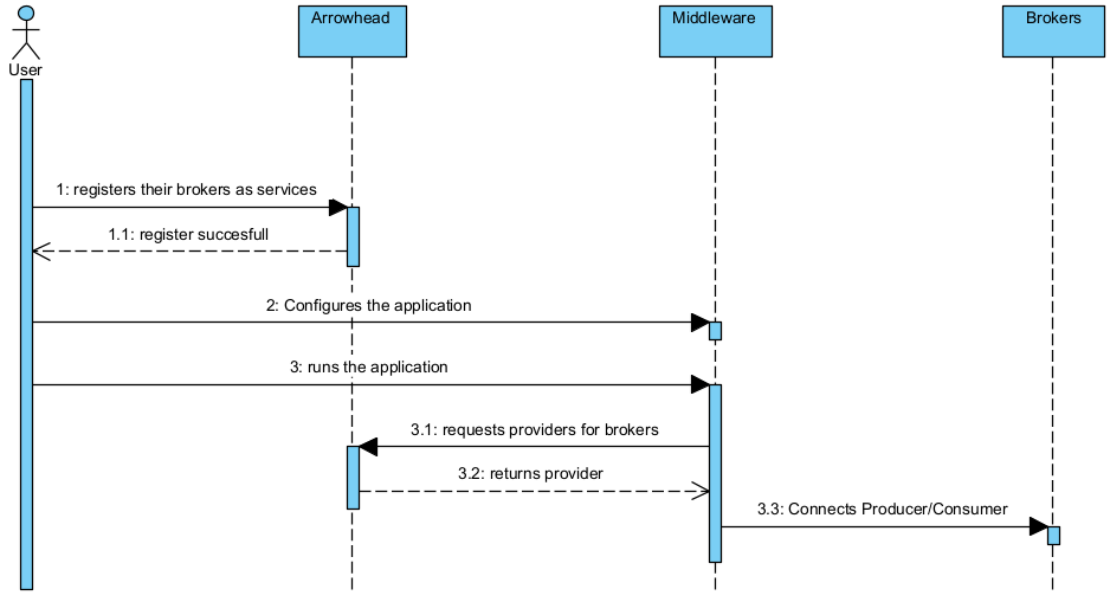

Figure 13 Use-case 6 Level 1 Sequence Diagram

1. The User registers their brokers and information, most notably, their address and port, at the Arrowhead's Service Registry.
2. The User configures through the Application's properties that the Middleware must use Arrowhead, and sets up devices that use the broker they just registered in Arrowhead.
3. The User runs the Middleware application.
4. The Middleware uses Arrowhead to request a provider for a service defined with the name of the protocol it requires.
5. The Middleware receives its address and port and passes it into its Producers and Consumers of that protocol, who will use it to establish their connection.

4.1.1.5 Deployment

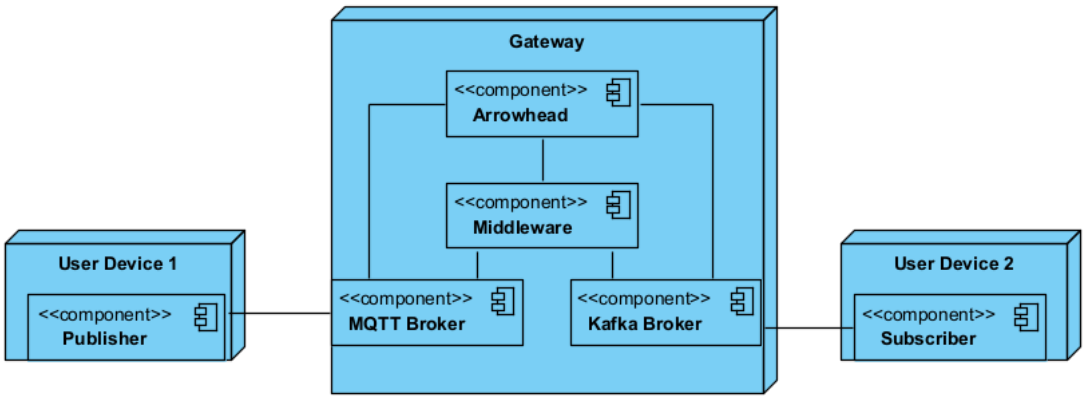

Figure 14 Level 1 Deployment Diagram

It's worth nothing that every relevant component for this application (the middleware itself, the brokers, Arrowhead, and the external publishers and subscribers) can be deployed wherever the user pleases. For this example, we are assuming a use-case with "User Device 1" wanting to publish its data to "User Device 2", from MQTT to Kafka, while brokers, the middleware instance and the arrowhead services all used the same machine.

Brokers

Brokers can be deployed as a running application or inside a container, like through Docker Hub. They can be configured in any way the User would prefer, including security with username and password, as long as the User later provides credentials to the application so its Producers and Consumers can also connect.

Other than MQTT, whose topics have no additional settings and can be auto created by its clients, it's recommended that topics and queues be created before running the application. Kafka Producers CAN be configured to auto-create topics, but Consumers cannot, and auto-creation implies less configurability, for, for example, number of topic partitions.

If the queues are not created before the RabbitMQ clients run, Producers will end up sending un-routable messages that will ultimately be discarded.

Arrowhead

The application easily supports the Arrowhead framework, as it was built with its skeleton available at (Arrowhead Skeleton Git, 2023). Brokers must be registered as services in Arrowhead before running the application, with the same name as the one the application uses for them.

The application also needs to know the address where the Service Registry Core service is available, as well have access to a certificate if necessary.

4.1.2 Level 2

4.1.2.1 Logical View

This Level 2 Logical View presents an overview inside the Middleware component represented in the Level 1 diagram.

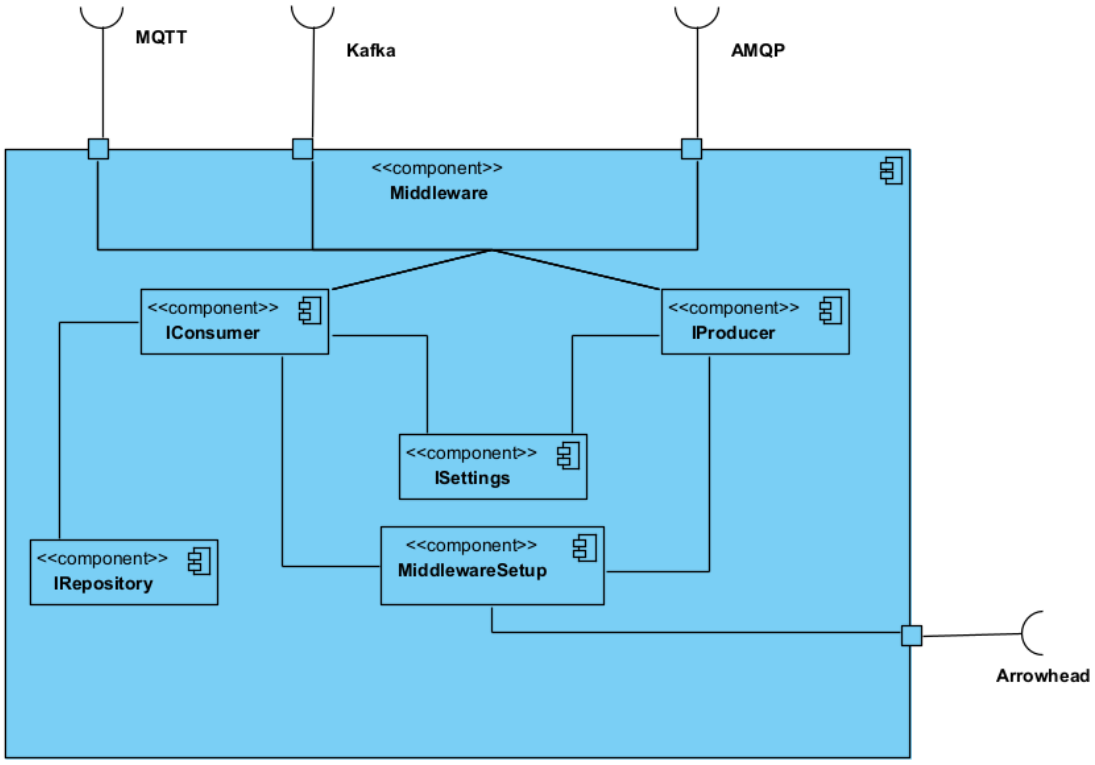

Figure 15 Level 2 Logical View

IConsumer and IProducer are the abstract classes every Consumer/Producer of the application must extend. Through their implementation, the Middleware can utilize them and their function of Consuming/Producing messages without having to distinguish between protocols.

Any of those implementations contains a reference to an ISettings class, which contains the level of QoS the client must work with. Any other necessary setting can be interpreted from the Consumer/Producer's implementation based on the settings defined in the properties file.

The Middleware Setup class is used to iterate over the properties file and create its respective Consumer and Producers, pass into them the appropriate settings, and give consumers a

reference to their linked Producers so they can communicate whenever a new message is received.

The **Repository** class is a necessity for Consumers of protocols who do not directly support "Exactly Once" messaging. They can store previously processed message identifiers, preventing the Consumer from processing them again.

4.1.2.2 Process

Table 4 Use Case 1 – Translate messages between different protocols

Use Case 1	***Translate messages between different protocols***
Description	The user intends to communicate between a producer and a consumer, regardless of the protocol
Actor(s)	User
Preconditions	1. Have access to the Middleware. 2. Available Brokers are running for the protocols involved, specified in the properties or through Arrowhead. 3. The User's Publisher and Consumer are connected to the brokers mentioned in 2.
Postconditions	1. Messages from the users Publisher are succesfully arriving at the Consumer
Necessary data	• Broker address • Publisher and Consumer (topic/queue) • Name of protocols involved

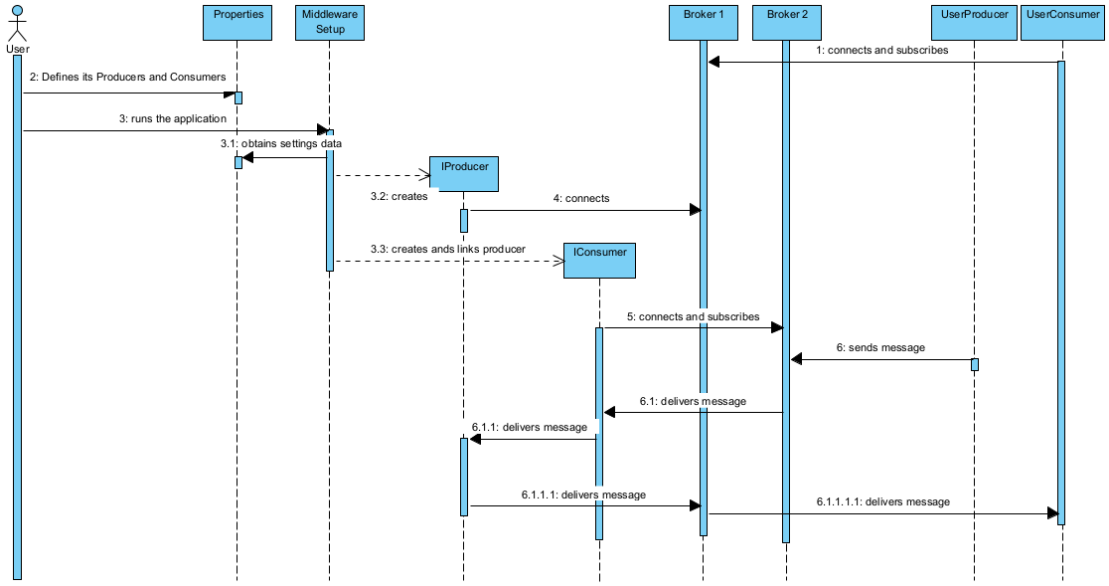

Figure 16 Use Case 1 Level 2 Sequence Diagram

1. User Devices connect/subscribe to the respective brokers.
2. User defines in Properties their own Producers and Consumers
3. User runs the application.
4. Application uses the devices defined in Properties to create Producers and Consumers and pass a reference of the Producer to its Consumer, so it can warn them on a new message.
5. The Producers and Consumers connect to their respective brokers based on implementation.
6. The User's Producer sends a message to Broker 2, who will deliver the message to its subscribers.
7. The message is received by the Middleware's Consumer, who returns the message to the Producer, who delivers the message to Broker 1.
8. The User's Consumer, subscribed to Broker 1, receives the message.

Table 5 Use Case 2 - Connect to the User's Brokers with properties

Use Case 2	***Use the properties file to specify broker connection formation***
Description	The user intends for the Middleware's Producers and Consumers to connect to the broker's address, specified in a properties file.
Actor(s)	User
Preconditions	1. Have access to the Middleware. 2. Have access to the Arrowhead Service Registry and Orchestrator. 3. Have access to at least one broker
Postconditions	1. Publishers and Consumers from the Middleware can connect to the brokers as specified by the User
Necessary data	• Broker address • Arrowhead Service Registry address • Name of protocols involved

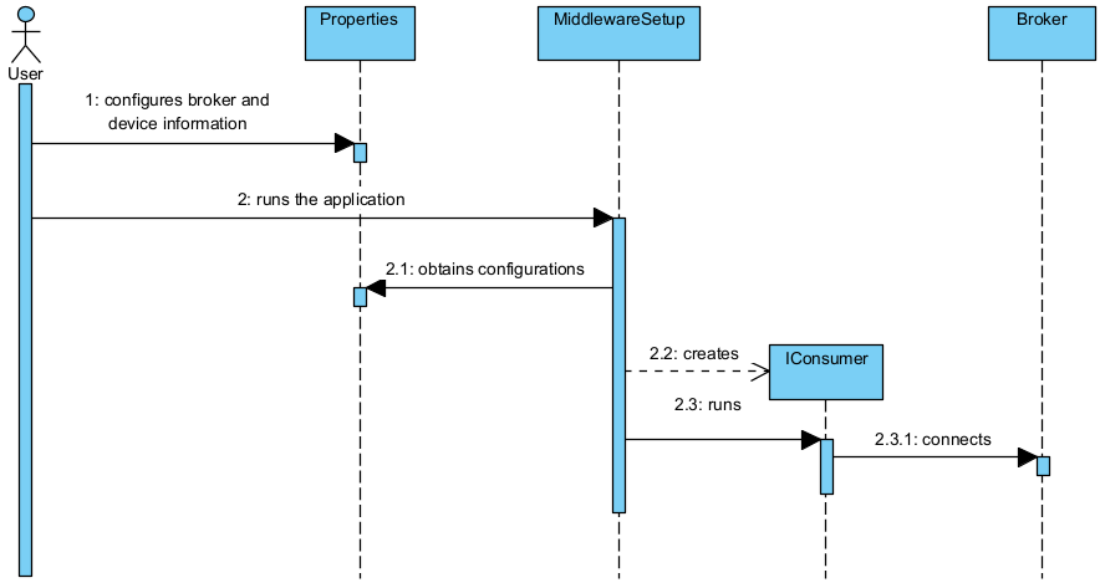

Figure 17 Use Case 2 Level 2 Process Diagram

In this Use Case:

1. The User configures the Properties file with the address and port from its brokers, as well as at least one device that uses that broker's protocol.
2. The User runs the application.
3. The Setup class fetches configuration information from the Properties file, including broker connection information.
4. The Setup class creates its clients.
5. The clients connect to the broker specified in properties.

Table 6 Use Case 3 - Message Delivery Guarantee

Use Case 3	***Message Delivery Guarantee***
Description	The application's Producers and Consumers must be able to support all 3 Message Delivery Guarantees levels (At most once, at least once, exactly once), regardless of whether that level of Guarantee is implemented by their protocol
Actor(s)	User
Preconditions	1. The User's Producer and Consumer are running. 2. The Middleware application is running, and its producer and consumer have been configured to use the correct Message Delivery Guarantee Level.
Postconditions	1. Messages are treated accordingly based on the Message Delivery Guarantee Level
Necessary data	• Broker addresses • Topic/Queue to connect to • Message Delivery Guarantee level

The application's Producers and Consumers must be able to support all 3 Message Delivery Guarantees levels (At most once, at least once, exactly once), regardless of whether it's implemented by their protocol.

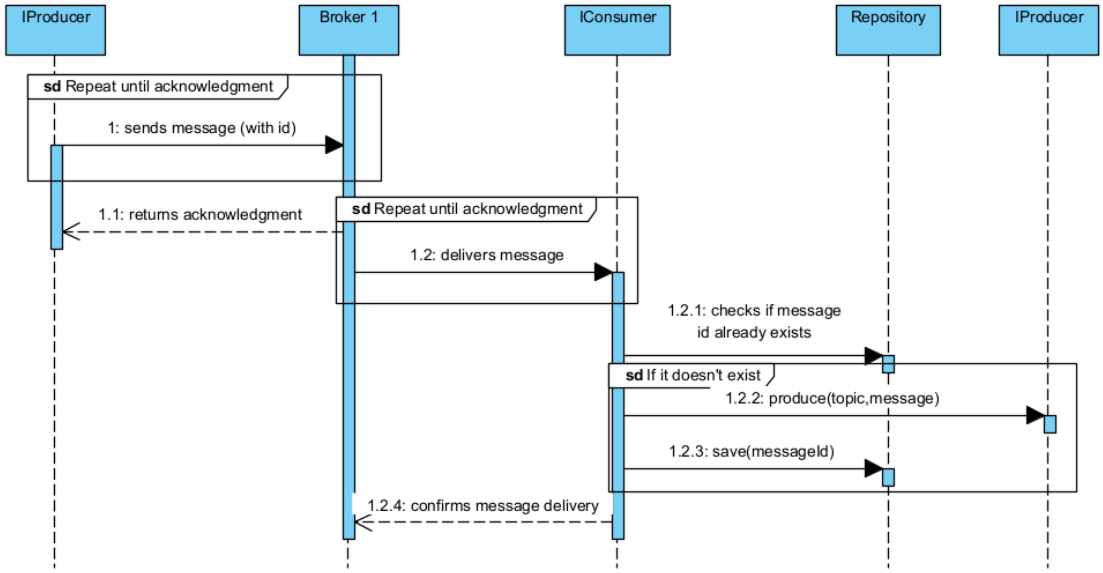

Figure 18 Use Case 3 Level 2 Sequence Diagram (Exactly Once)

This Diagram applies to Kafka and AMQP, as MQTT already has its own implementation of "Exactly Once" (MQTT QoS, 2023).

1. Middleware Producers will always produce their messages with a random identifier (Record key in Kafka, message ID in AMQP).
2. The Broker will return an acknowledgement of the received message. If not, the Producer will attempt to resend the message, until an acknowledgement is received.
3. Consumers, upon receiving the message, will verify if it already has been processed at their own personal repository.
4. If not, the message is processed (instructs the Producer that a new message has been received), and the message ID of the processed message is stored in the repository.
5. Only then will the Consumer confirm delivery. This way, if the Consumer is somehow unable to confirm the delivery, which would cause the Broker to resend the message later, the message would not be processed again, as its identifier would already be present in the repository.

Table 7 Use Case 4 - Use Consumer's Topic

Use Case 4	***Use Consumer's Topic***
Description	The User must be able to specify that it wants the Middleware's internal Producer to use the topic of the Consumer from where the message originated, as well as removing a specified pattern from the topic's name
Actor(s)	User
Preconditions	1. The User's producers and consumers are running. 2. The Middleware application is running, and its producer and consumer have been configured to use the correct Message Delivery Guarantee Level.
Postconditions	1. The Middleware's Publisher publishes messages to whatever topic its Consumer received the message from through the User's Publisher
Necessary data	• Broker addresses • Topic/Queue for the Consumer to connect to (User's Producer)

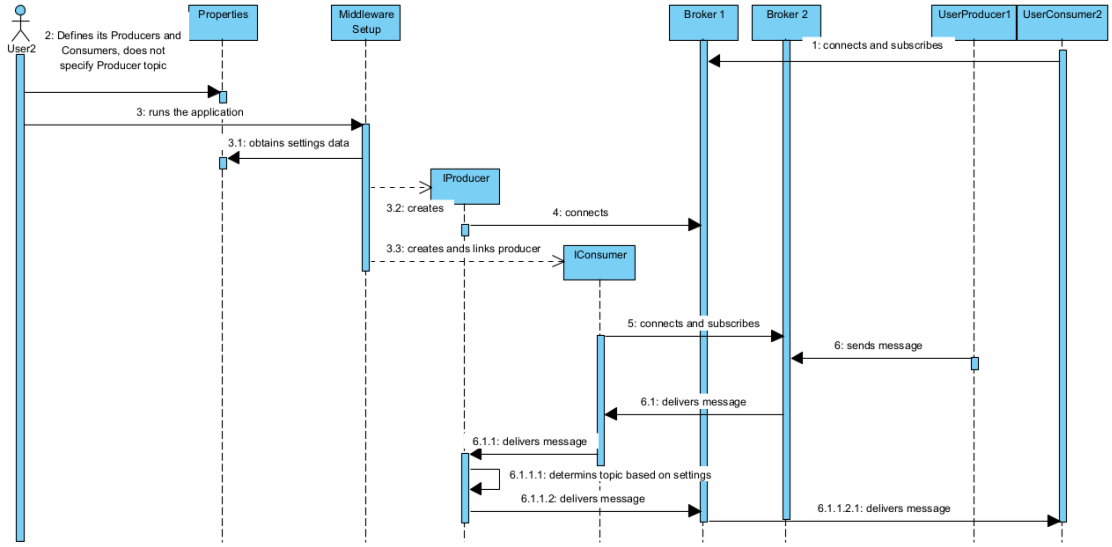

Figure 19 Use Case 4 Level 2 Sequence Diagram

This Use Case has all the exact same steps as Use Case 1. The only difference is that between the Producer receiving the message and delivering it to its broker, it will first determine the topic it must produce to, based on the settings with which it was created, deleting patterns from the topic received by the Consumer if necessary.

Table 8 Use Case 5 Client Configurability

Use Case 5	***Client Configurability***
Description	The User intends to configure through the properties file any setting that is also configurable through the protocols Java library
Actor(s)	User
Preconditions	1. Have access to the Middleware.
Postconditions	1. Publishers and Consumers behave accordingly to the settings they were configured with
Necessary data	• Broker addresses • Topic/Queue to connect to • Settings to change

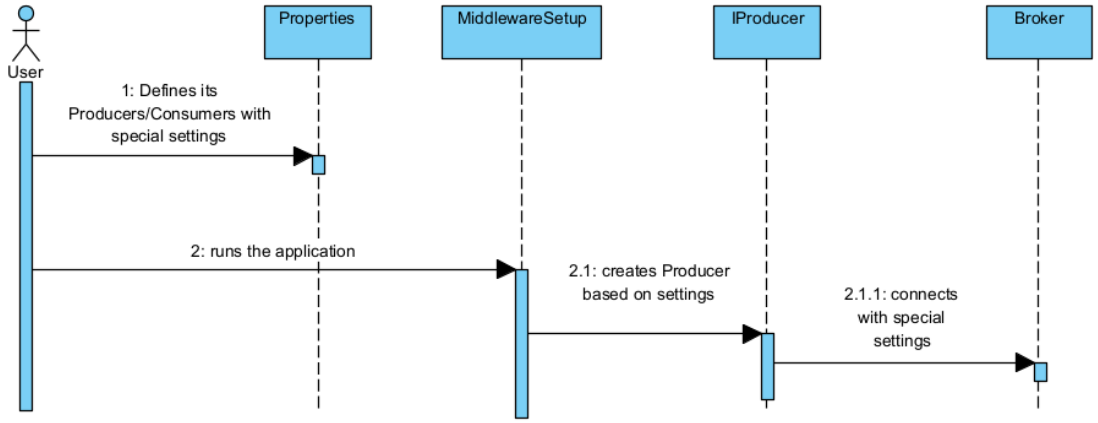

Figure 20 Use Case 5 Level 3 Diagram

1. The User defines streams to connect in the Properties file with special settings (for example, connection timeout for MQTT or value serializer for Kafka)
2. The User runs the application.
3. The Setup class creates its Producers (and Consumers) and passes them their associated settings.
4. The Producers and Consumers parse these settings and connect to the broker/handle messages in a different way, as defined by the libraries they use.

4.1.3 Level 3

4.1.3.1 Logical View

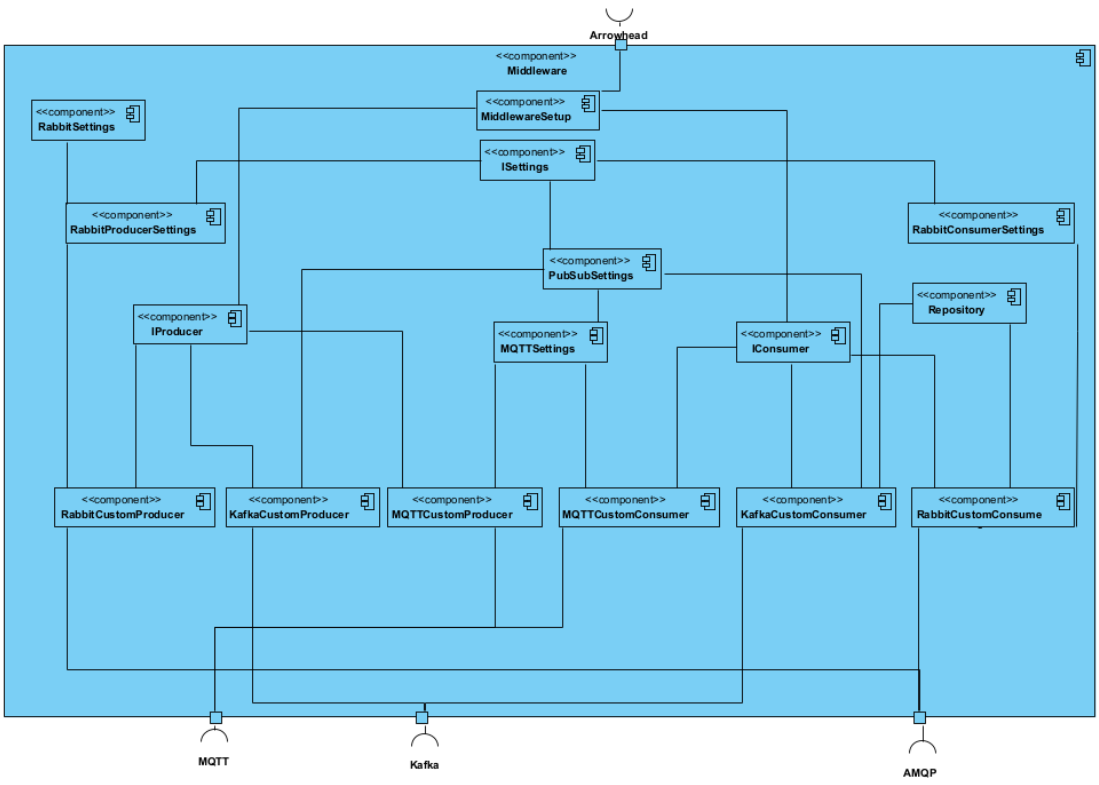

Figure 21 Level 3 Logical View

The IProducer and IConsumer abstract classes must be implemented by any Producer/Consumer supported in the Middleware. They contain methods designed to facilitate the interaction between the multiple protocols.

The **ISettings** class contains an attribute necessary for every Producer and Consumer: QoS. The **PubSubSettings**, for every Producer and Consumer of a topic-based protocol: topic and client.id

Some protocols require special settings to connect, like the **MQTT Settings** class, the **RabbitProducer/ConsumerSetting**, containing information of exchange and routing key or queue, the **RabbitSettings** containing details about the connection itself, and the KafkaSettings which are already provided by its library.

Rabbit/Kafka/MQTT Custom Producers are implementations of the IProducer class. They're responsible for producing messages to their respective brokers, to connect to the User's Consumers.

Rabbit/Kafka/MQTT Custom Consumers are implementations of the IConsumer class. They're responsible for receiving messages from their respective brokers, which were sent by the User's Producers.

The **MiddlewareSetup** class is responsible for parsing through the properties file and creating the Producers and Consumers accordingly, and utilizing the Arrowhead library.

5 Implementation

In this chapter, we discuss how the application was implemented, and the way its classes interact with each other and the outside.

5.1 Technologies used

To develop this application, we chose to use Java, since not only does it offer a good performance compared to other languages (Dhalla, 2020), which is a considerable requirement for an application designed for message distribution, but is also the language the Arrowhead Framework and the Application Library designed to interact with it are in.

All 3 supported protocols (MQTT, Kafka and AMQP) offer a Java library for easy configuration and use of their Producers and Consumers.

Table 9 Software Versions

Software	**Version**
Java	11.0.16
Arrowhead Application	4.4.0.2
Eclipse MQTT	1.2.5
Apache Kafka	3.2.3
Rabbit MQ (AMQP)	5.9.0

5.2 Implementation Description

This section contains an in-depth overview of the application's software and the interactions between classes, all according to the Design made in the previous chapter.

5.2.1 Application Settings

In the properties file, the user may define whether to use Arrowhead, and the default address and port to connect to the broker of each protocol (in case "arrowhead_enabled" is set to false, or Arrowhead orchestration fails), as well as declare all its external devices, be them producers or consumers. Code Snippet 1 gives an example of a configuration file, where the User has expressed a translation from their MQTT Producer to their Kafka Consumer.

```
{
  "Producers": [{
    "internal.id": "device01",
    "protocol" : "mqtt",
    "additional.props": {
      "topic": "test",
      "connection.timeout": 20,
      "qos": 0,
      "client.id": "middleware-consumer"
    }
  }],
  "Consumers": [{
    "internal.id": "device04",
    "protocol" : "kafka",
    "additional.props": {
      "topic": "test",
      "client.id" : "middleware-producer",
      "qos": 0
    }
  }],
  "Streams": [{
    "from.producers" : "device01",
    "to.consumers" : "device04"
  }]
```

Code Snippet 1 Properties Example

Internal ID – This field defines a unique identifier for the device so the Middleware can know which devices are supposed to be connected. The Internal ID is only used inside the Middleware, it has no connection to the ID of the Producer or Consumer itself.

Protocol - Refers to the messaging protocol this device is producing/consuming to/from.

Additional Props - These concern the properties of the equivalent consumer/producer the application will create to communicate with the user's device, and not necessarily the device itself (the user's device may, for example, produce messages with a QoS of 2, but the user may declare for its Middleware consumer equivalent to receive messages with a QoS of 0). In Figure 4, for example, the application will have to create an MQTT consumer to connect to a broker, to receive the messages from the user's MQTT producer. This consumer will only treat messages with a QoS of 0, regardless of the device's QoS, connect to the topic "test" (which the user should also be using), and set its connection timeout to 20.

MQTT options – (MQTT Connect Options, 2023)

Kafka options – (Kafka Producer Config, 2023) (Kafka Consumer Config, 2023)

AMQP options – (RabbitMQ Connection Factory, 2023)

Streams - With all the equivalent Consumers/Producers created, the application will assign to each internal Consumer a list of Producers, as specified in this entry.

5.2.2 IProducer and IConsumer classes

To answer the challenge of a simple design that could connect consumers and producers regardless of the protocol, we created the "IProducer" and "IConsumer" abstract classes. Every supported protocol has at least one implementation of each.

IProducer

The abstract method `produce(topic,message)` must be implemented by each Producer according to their own protocol logic. This method being called means the Consumer linked to it just received a message from a certain topic (1st parameter) with a certain content (2nd parameter). The result should have the producer sending a new message to its broker.

IConsumer

The OnMessageReceived(topic,message) is already implemented:

```
public void OnMessageReceived(String topic, String message) {
        for (IProducer producer : producerList) {
            producer.produce(topic,message);
        }
    }
```

Code Snippet 2 On Message Received

It has the purpose of notifying each Producer linked to this consumer that a new message was received at topic/queue "topic" with contents "message". `IConsumer` implementations only need to make sure this method is called every time a new message is received.

Possible exceptions thrown during the execution of the produce method are handled by the `Producer` class and appropriately logged. This way, any error during the production of one message from a certain producer does not compromise the other producers.

This class also requires an implementation of the "`run()`" method for the implemented Consumer to subscribe to its topic/queues. This is so the Consumer may be created first, but not get its thread stuck while waiting on messages.

5.2.3 Setup

Before starting the translation process, the `MiddlewareSetup` class must first iterate over the client's devices and create its own consumers and producers accordingly.

1. **Map initialization**

Firstly, the application will initialize 2 maps: One for producers, and one for consumers. Their keys represent the associated protocol (for example, Kafka), while their values hold a String representing the path from the source root to the respective class. This will be used later to easily create new instances for them without elaborate switch cases, while also allowing to quickly switch out one consumer for another, if necessary.

2. **Broker Connection**

The application will then check if the Arrowhead mode was enabled. If not, it will initialize a map with the name of the protocol as keys and the address and port to connect to as values.

3. **Create Producers and Consumers**

The system will next iterate over the list of User Consumers/Producers as defined in properties and create a Producer/Consumer for each.

The Internal ID will be used as a key to store the resulting Producer/Consumer in the Map of Producers/Consumers.

The application will check if an address and port already exist for the Protocol of the producer/consumer. If not, that means step 2 was not executed and Arrowhead Mode is enabled, and so it will send a request to the Arrowhead's Orchestrator for a service of the same name as the protocol. If it obtains an invalid response (for example, an invalid authorization, service not present), the default address and port for that protocol will be used instead.

The obtained address and port will be stored in the map to avoid additional requests for clients using the same protocol.

The set of values defined under the "`additional.props`" setting will be transformed into a Map of type `<String,String>`. The Constructor of each Producer and Consumer will receive this Map and process it accordingly.

While Kafka already provided a "`ProducerConfig`" and "`ConsumerConfig`" classes that took as a constructor a `Map<String,Object>` much like the one we use to pass these additional settings to the Consumers and Producers, both MQTT and RabbitMQ had extensive

configuration options for which was necessary a new class, able to individually define each value for each attribute based on the received map.

To instantiate the Producers and Consumers, we considered 2 options:

```
private IProducer createProducer(ConnectionDetails cd, String name,
Map<String,String> settings) {
        Class<?> c = null;
        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 3 Creating Classes Option 1

```
private IProducer createProducer2(ConnectionDetails cd, String name,
Map<String,String> settings) {
        IProducer producer = null;
        switch (name) {
            case "mqtt":
                producer = new MqttCustomProducer(cd,settings);
                break;
            case "kafka":
                producer = new KafkaCustomProducer(cd,settings);
                break;
            case "rabbit":
                producer = new RabbitCustomProducer(cd,settings);
        }
        return producer;
    }
```

Code Snippet 4 Creating Classes Option 2

Option 1 searches for the constructor for a class with the path as defined in the Producer/Consumer map, then instantiates it with the needed parameters and returns it so the Setup phase may continue.

Option 2 instead uses a switch case. Assigning each protocol name their respective Producer/Consumer class.

While option 2 seems simpler and easier to understand, it also isn't as abstract as option 1, as it requires the user to specifically reference their desired class in an extensive switch case, for both Producers and Consumers.

Option 1 needs only to be loaded with the Producer and Consumer Maps before, which is easier, simpler, and overall user-friendly to configure (which is one of the objectives of this application). Once the protocol is registered in both maps, the application will know how to behave once it encounters a device with the same protocol name.

4. **Connecting Consumers to Producers**

Now that every instance of Consumer and Producer was created, we can connect them. Through the "`Streams`" entry of the properties, we pick each Consumer from the `ConsumerMap` using its "`internal.id`" and add every Producer to its private Producer list with the same logic.

Once every Stream has been created, the Consumer Map will be iterated to call the "`run()`" method on a new thread each. The consumers are now waiting for messages to redirect, and the Setup phase is complete.

5.2.4 Message Delivery Semantics Implementation

While MQTT has its own implementation for Delivery Semantics, which they call "QoS levels", Kafka only supports "Exactly-Once" messaging through its "Transactions API" (for more than 3 brokers), and RabbitMQ has no implementation whatsoever.

To still support it, we decided to start from the "At-Least-Once" already implemented for both protocols. This guarantees that the message will eventually be received, we just need to make sure it's not processed multiple times.

Through the "key" value of messages in Kafka, and the "messageId" of messages in AMQP, we can store messages that have already been processed, and only process messages with an identifier not stored yet.

To store messages that have already been processed, we developed the "`IRepository`" interface, which requires the implementation of the "`registerNewMessage(messageID)`" and "`messageExists(messageID)`" methods and implemented a version of this class with a List<String> as the storage.

```
List<String> database = new ArrayList<>();

    @Override
    public boolean registerNewMessage(String messageId) {
        if (database.size() > Constants.MAX_UNIQUE_MESSAGES) {
            database.remove(0);
        }
        return database.add(messageId);
    }

    @Override
    public boolean messageExists(String messageId) {
        return database.contains(messageId);
    }
```

Code Snippet 5 IRepository Implementation

This way, the AMQP and Kafka consumers can check if a message has already been processed with "`messageExists(messageId)`", and, if it returns false, process it, and immediately store that message's ID.

```
if (settings.getQos()==2) {
if (!qosRepository.messageExists(record.key())) {
newMessage(record);
              qosRepository.registerNewMessage(record.key());
       }
} else {
       newMessage(record);
}
```

Code Snippet 6 Kafka "Exactly Once" implementation

It's worth noting that Kafka or AMQP messages do not have their unique identifier (key/message.id) unless they are specifically produced with one. Producers in Middleware can generate an identifier for each message, but the Application User must also publish messages with an ID so the Middleware Consumers can guarantee "Exactly-Once".

5.2.5 Using Arrowhead

The application was built by extending the Arrowhead Application Skeleton, available at (Arrowhead Skeleton Git, 2023)

Arrowhead was used to connect producer and consumers with the message-oriented brokers, but it is still possible to run the system without Arrowhead support, if the "arrowhead_enabled" property in the configurations file is set to true.

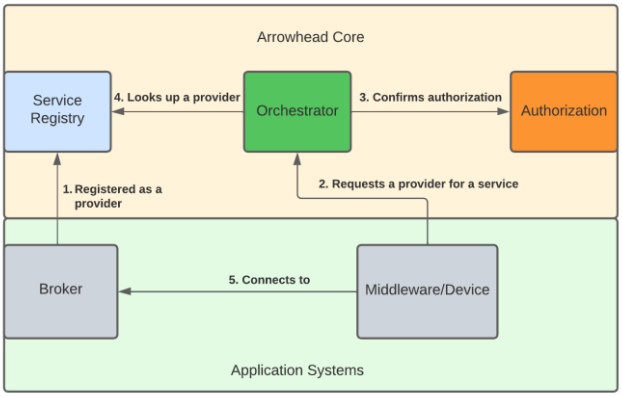

Figure 22 Arrowhead Orchestration

Based on Figure 22, our application, the Middleware, plays the role of a "Service Consumer" in the Orchestration Process, acquiring connection details for each of the protocols specified by the user in the properties file, requesting providers for the service with the same name as the protocol.

Before setting it to true, however, the user must first register a new provider for the protocol they're using in the "Service Registry" (Step 1 in Figure 22).

To register a new service in the Arrowhead System, the user must send a request to the Service Registry as described in (Arrowhead Git, 2023). The only relevant parameters our application will receive after the Orchestration request are the address, port, and service definition. Address and port must be defined as the exact address and port the application's producers and consumers can connect to the broker at. The "service definition" has the same name as the name specified in the "protocol" entry of definitions, and in the Setup phase while associating protocols with their Producer/Consumer pair.

If the user declares, for example, that they are using an MQTT publisher, then during the processing of the configurations file, the application will use the Arrowhead Application Library to send an Orchestration Request to the Orchestrator Core Service, requesting a provider for the "mqtt" (default MQTT name) service. The response will contain details about

the found provider, most notably, the address and port to connect to. Figure 23 is an example of this response obtained in a Postman request.

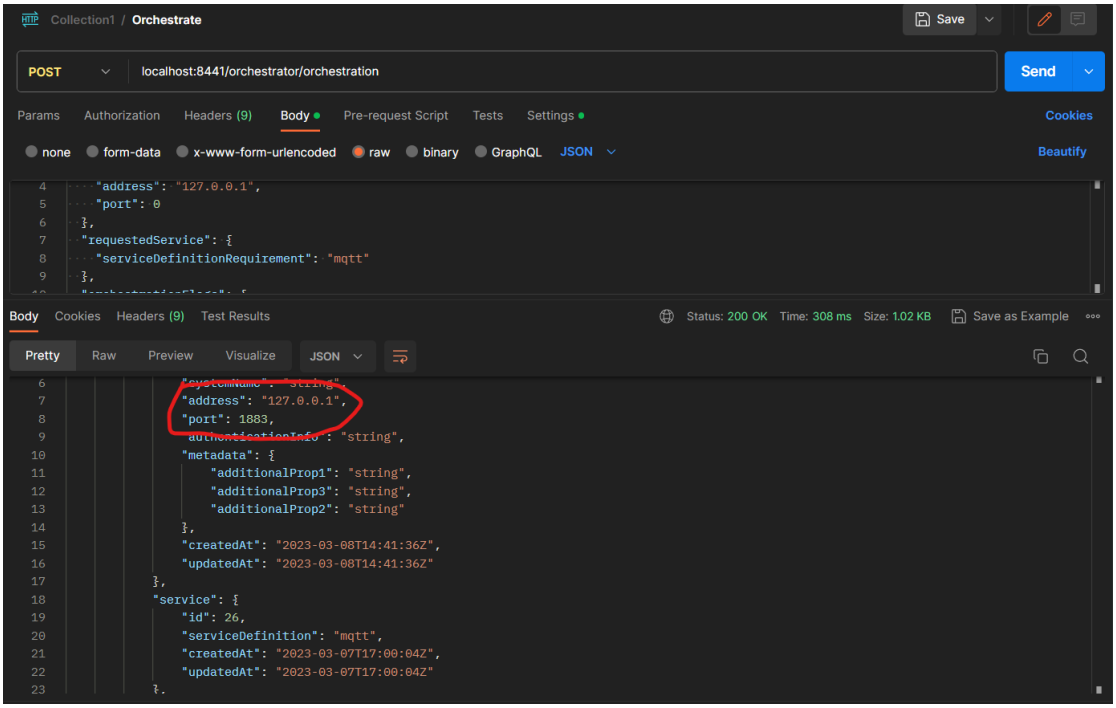

Figure 23 Example Arrowhead Response

The Orchestration response may contain multiple providers. It is up to the user to make sure the protocols they are using contain exactly one provider to connect to, to prevent undesirable outcomes, like the Middleware connecting to a broker the user's devices are not connected to.

If any error occurs with Arrowhead (failed to connect, no providers found), the application will use the default connection for the brokers defined in properties instead.

5.2.6 Additional Client Settings

AMQP

Unlike MQTT and Kafka, producers and consumers in AMQP do not send messages to a topic they both directly interact with. As previously explained, they instead use exchanges and routing keys, or queues, respectively, so we must connect to them in a different way, since these properties are made from the user's perspective.

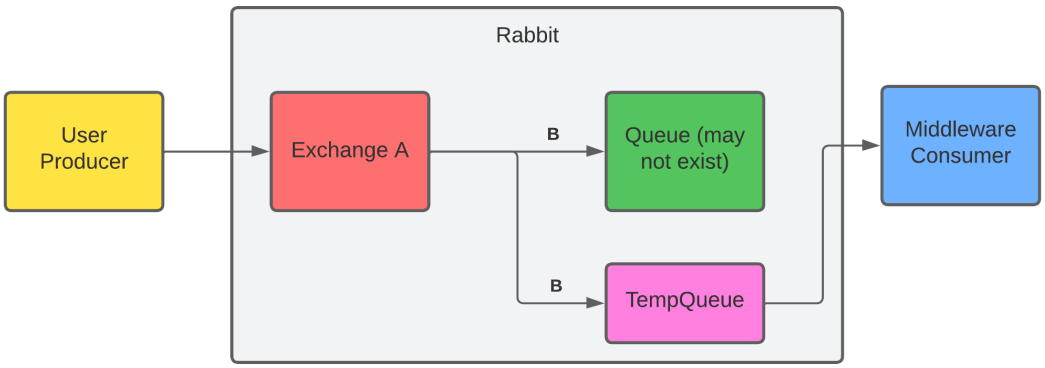

Figure 24 Middleware Consumer AMQP Adjustment

In case of the user having an AMQP producer, producing to Exchange "A" with routing key "B", we can create an equivalent consumer in the application consuming from a temporarily created queue, a queue which may be bound to Exchange "A" with routing key "B", or directly to queue "B" if the default exchange was used. This process is illustrated in figure 24.

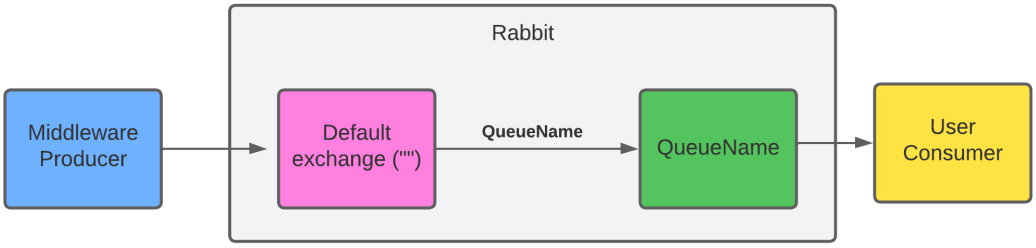

Figure 25 Middleware Producer AMQP Adjustment

If the user has an AMQP consumer, consuming messages from queue "QueueName", we can instead use the default exchange (""), and "QueueName" as the routing key. In AMQP, this default exchange directly maps messages to the queue with a name equivalent to the message's routing key. This process is illustrated in figure 25.

Consumer's Topic

By not directly specifying the name of the Consumer´s topic, if in MQTT/Kafka, or the name of the queue if in AMQP, the internally created Producer will know to produce messages to the topic/queue with the same as the topic/queue the consumer who called its produce method received the message from in the first place.

This is especially useful for MQTT Consumers for example, where Consumers can subscribe to topics using wildcards. Subscribing to "#", for example, will make it so any message that gets produced into that MQTT Broker to whatever topic is received from the Consumer, while the

Producers linked to it will produce messages knowing the topic where the message originally came from.

ID Generation

This property refers to how the Middleware should proceed in generating message identifiers for producing messages in Kafka and AMQP. If not filled in or filled in with the value of “random”, messages will be produced with a random identifier using the UUID library. If the property is set to “messageBody” however, the identifier used will be the content of the message itself.

6 Tests and System Evaluation

In this chapter we will go over the tests that were developed to reliably verify the entire system and if its parts work as expected.

6.1.1 Unit tests

These tests will test individual components of the application, isolated from the rest.

The tests below in particular prove that the "`topicFromConsumer`" method of `IProducer` can correctly transform a topic based on the pattern defined to remove from it.

```
@Test
    void topicCannotBeEmpty() {
        assertThrows(RuntimeException.class,
                () -> producer.topicFromConsumer("Remove/"),
                "Expected method to throw"
        );
    }

    @Test
    void topicIsRemoved() {
assertEquals("temperature",producer.topicFromConsumer("Remove/temperature")
);
    }

    @Test
    void multipleTopicAreRemoved() {
assertEquals("temperature",producer.topicFromConsumer("Remove/temperature/R
emove"));
    }
```

Code Snippet 7 IProducer Unit Tests

6.1.2 Integration tests

These tests will test the expected interaction of multiple components of the application, mainly producers and consumers.

```
void messagesContentsArePreserved(ProducerAndConsumerPair pair) throws
InterruptedException {
        setUp();
        pair.producer.produce("teste","TEST-DATA");
        Thread.sleep(Duration.ofSeconds(3).toMillis());
        assertEquals("TEST-DATA",pair.consumer.getLastMessage());
    }
```

Code Snippet 8 Message Contents Are Preserved Test

The above code checks if message contents are preserved while passing through the producer, broker, and consumer.

```
void producerUsesCorrectTopic(ProducerAndConsumerPair pair) throws
InterruptedException {
        pair.producer.produce("prefix/notTeste","FAKE-DATA");
        Thread.sleep(Duration.ofSeconds(3).toMillis());
        assertNotEquals("FAKE-DATA",pair.consumer.getLastMessage());

        pair.producer.produce("prefix/teste","TEST-DATA");
        Thread.sleep(Duration.ofSeconds(3).toMillis());
        assertEquals("TEST-DATA",pair.consumer.getLastMessage());
    }
```

Code Snippet 9 Producer Uses Correct Topic Test

This test checks if Producers can correctly use the topic name passed on through the "produce" method to produce their message if they were created with settings where the topic/queue name was empty.

```
void allMessagesAreRedirected(ProducerAndConsumerPair pair) throws
InterruptedException {
        int initialMessages = pair.consumer.getNumberOfMessages();
        for (int i = 0; i < 1000; i++) {
            pair.producer.produce("teste", "TEST-DATA-" + i);
        }
        Thread.sleep(Duration.ofSeconds(6).toMillis());
        assertEquals(initialMessages                                    +
1000,pair.consumer.getNumberOfMessages());
    }
```

Code Snippet 10 All Messages are Redirected Test

Test if, with QoS 2, all messages get passed from the producer to the consumer.

```
@ParameterizedTest
@MethodSource("provideExactlyOnce")
void verifyExactlyOnce(ProducerAndConsumerPair pair) throws
InterruptedException {
    for (int i = 0; i < 10; i++) {
        pair.producer.produce("teste","TEST-DATA");
    }
        Thread.sleep(Duration.ofSeconds(4).toMillis());
        assertEquals(1,pair.consumer.getNumberOfMessages());
}
```

Code Snippet 11 Exactly Once Test

This test verifies that messages are being handled correctly by the specified message delivery semantics. This code creates a Producer set to produce messages with a message ID equal to the message's content. The producer will then send 10 messages with the same content, meaning, the same ID, simulating a broker who failed to receive an acknowledgment from the consumer, attempting to resend the message to it multiple times before an answer back.

If the Consumer correctly implements "Exactly-Once" Delivery semantics, it will receive all but only process a single message, since the message ID of the first message will be stored, preventing other messages with the same ID from being processed.

This test is only executed with the implementations of Kafka and AMQP. It couldn't work on MQTT, since MQTT does not allow producers to specify an ID while producing their message, although MQTT does include its own "Exactly-Once" guarantee system.

We also conducted similar tests to verify that for different ID's, all messages were received as expected, and for a QoS level of 0, all messages were received regardless.

6.1.3 System tests

To test that the entire system was working as expected, we created stand-alone publishers for Kafka, MQTT, and AMQP as well as subscribers that would use the time passed between X messages to calculate the average number of messages received per second. To avoid extraordinary results, we executed each test 10 times.

The Middleware uses Arrowhead services to acquire the address for the previously registered brokers.

The Code for the Python scripts was made using the kafka, paho.mqtt and pika libraries.

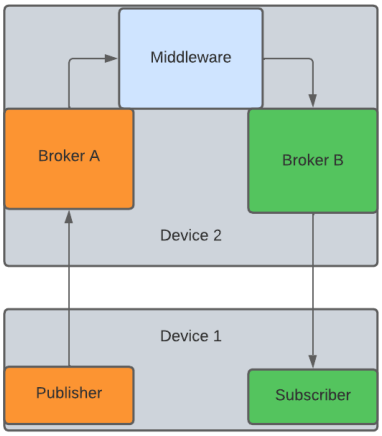

Figure 26 Test Message Flow

Devices and application were set up in an identical way to Figure 26.

The results obtained are shown in Table 10.

Table 10 Performance of Python publishers and subscribers

	MQTT Subscriber (msg/s)	**Kafka Subscriber (msg/s)**	**Rabbit Subscriber (msg/s)**
MQTT Publisher	A: 9950,5 M: 10063,6 m: 9761,8 D: 104,3	A: 10390,9 M: 10920,2 m: 10085,1 D: 225,5	A: 10144,2 M: 10420,4 m: 9757,7 D: 188,9
Kafka Publisher	A: 23865,6 M: 26334,1 m: 22238,6 D: 1166,14	A: 27778,7 M: 30463,9 m: 20479,2 D: 2873,64	A: 23744,2 M: 25619,7 m: 21182,7 D: 1597,9
Rabbit Publisher	A: 5117,83 M: 6657,55 m: 4094,39 D: 844,47	A: 5146,45 M: 6552,37 m: 4551,52 D: 663,655	A: 5665,62 M: 8252,89 m: 4357,02 D: 1229,72

Legend:

- A: Average
- M: Maximum value
- m: Minimum value
- D: Standard Deviation

From this table of results, it's possible to interpret some interesting data, like the one in the figure below:

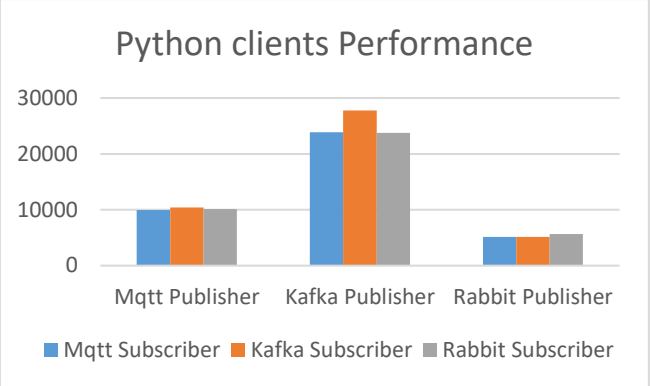

Figure 27 Python Clients Performance

From the graph above, it's easy to conclude that using a different protocol for the Publisher (associated with a User's Consumer) has a much larger impact than using a different protocol for the Subscribers (associated with a User's Producer).

To further evaluate performance, we repeated the same tests but for Java Publishers and Subscribers.

The Middleware's code for its Producers and Consumers was reutilized for the Java clients of these tests.

Table 11 Performance of Java publishers and subscribers

- > |	**MQTT Subscriber**	**Rabbit Subscriber**
MQTT Publisher	A: M: m: D:	A: 14363,8 M: 15205,2 m: 13186,8 D: 556,4
Kafka Publisher	A: 31974,7 M: 34059,8 m: 29075,4 D: 1622,3	A: 30429,6 M: 33113,7 m: 27594,8 D: 1834,2
Rabbit Publisher	A: 24728,2 M: 28383,2 m: 19542,5 D: 2848,7	A: 18380,0 M: 23374,4 m: 13861,5 D: 2827,0

The Kafka subscriber in Java took multiple seconds before receiving his first message, leading to inaccurate results where all the messages were already at the broker before the first arrived. This, for some reason, never happened with the Middleware's Kafka subscriber, despite using the same code and settings.

MQTT -> MQTT was resulting in a very inconsistent number of messages who were delivered.

Despite the undesirable outcomes that occurred while testing with Java Publisher and Subscriber's, it's still possible to observe the Java Clients had a consistently higher performance than the Python Clients for RabbitMQ, as evidenced from the graph below.

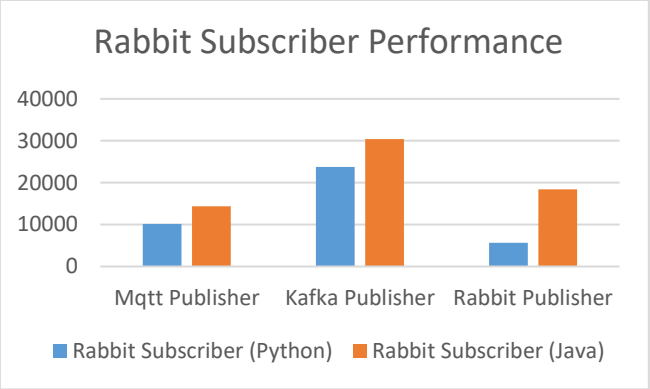

Figure 28 Rabbit Subscriber Performance

QoS tests

To perform these tests, we used the same Python scripts as above, but added an ID generation to the Producer, and specified the Middleware's Producer and Consumers with the different QoS levels.

Table 12 QoS Test Results

	At Most Once	**At Least Once**	**Exactly Once**
Kafka	19193,64	1090,028	388,216

These results imply that having a higher message reliability has a large cost in performance. It's a decrease of 95% and 98% in messages per second for At Least Once and Exactly Once respectively, when compared to At Most Once. However, depending on the system, 400 messages per second could be a worthwhile trade for more reliability.

7 Conclusions

In this chapter, we compare the results obtained in the previous chapter to the objectives initially set, discuss the overall work done and possible future implementations.

7.1 Accomplished goals

1. **Develop an application capable of connecting devices who communicate in different protocols - COMPLETED**

This objective was understood as fully completed through our tests, which had Producers and Consumers in MQTT, Kafka and AMQP communicate in all directions with each other, while maintaining the original message's contents and producing and receiving messages from the expected topic/queues.

2. **Support broker address configuration with Arrowhead - COMPLETED**

All our test have shown that the usage of Arrowhead services to acquire broker addresses has been successfully integrated.

3. **Develop a simple and clear interface that can easily support additional translation rules, as well as new protocols – COMPLETED**

Given that the Application's Architecture could support three protocols, two who worked with topics (MQTT and Kafka), and one who worked with exchanges, routing keys, and queues (AMQP), without losing any functionality associated with the protocols, we consider this objective as completed.

4. **Support advanced producer/consumer configuration - COMPLETED**

Our tests with Message Delivery Guarantees and the usage of the Consumer's topic in the Producer were all successful.

7.2 Limitations and future development

Even though the application reached all initially planned objectives, we were still able to identify some functionalities the system was missing and could benefit from in the future.

Similar to how Kafka Connect handles creation of new connectors through a REST API (Connect Rest API, 2023), our application could also implement an API to enable users to interact with the application less intrusively, creating and deleting new translation streams or changing brokers during runtime.

RabbitMQ brokers offer a Management Plugin accessible by authenticated clients (Rabbit Management, 2023). Through it, the application could obtain useful insights from the queue its RabbitMQ Consumers are currently subscribed to like number of ready messages, for example. From that information, it could determine if more instances of the Consumers should be created to speed up the process and proceed accordingly, taking advantage of the multicore capabilities of the edge platform.

Making use of the application's simple and easily extendible architecture, it could easily support more IoT protocols with a similar design to the ones already implemented (publish/subscribe, one-to-many), like for example XMPP. Moreover, since it requires changing a single line to select a different connector for a protocol, the application could implement translators that do something more than simply redirecting messages.

There are also some possible scalability features to implement, like distributing the Middleware across multiple devices and load balancing based on the work load its Consumers are reporting.

7.3 Final Appreciation

Considering this project managed to achieve every objective as "COMPLETED", going beyond simply translating messages from one place to the other, I am satisfied with the work I did, particularly the efforts In providing additional configurations, like our own Message Delivery Semantics some of our protocols did not even implement, the architecture built for the application that made it so easy to support them, and how with the properties file and just a couple of seconds, I could just create a connection between different protocols, for, for example, my tests.

Despite this, there are some aspects I mentioned in the last subsection I would have liked to have managed to implement, especially the API for easier use. Despite this being an application built to facilitate communication in the IoT, communicating with it is not as simple.

On a personal level, the opportunity to work much more closely with professors, to interact with them and with partners from other universities under the Ferrovia 4.0 Project, and because of it, getting to attend an event of an area I am somewhat interested in was all a very enriching experience.

Although investigative work like this has some challenges I was not initially prepared for, and universally frustrating moments like something simply not working, this experience contributed greatly to my professional and personal development.

References

Al-Sarawi, S., Anbar, M., Abdullah, R., & Al Hawari, A. B. (2020, July). “Internet of things market analysis forecasts, 2020–2030”. In 2020 Fourth World Conference on smart trends in systems, security and sustainability (WorldS4) (pp. 449-453). IEEE.

“Apache Flink Use Cases”, https://github.com/johanvandevenne/kafka-connect-mqtt, last accessed 08 07, 2023.

“Arrowhead Git”, https://github.com/arrowhead-f/core-java-spring, last accessed 08 07, 2023.

“Arrowhead Skeleton Git”, https://github.com/arrowhead-f/client-skeleton-java-spring, last accessed 08 07, 2023.

“Connect Documentation”, https://docs.confluent.io/platform/current/connect/index.html, last accessed 08 07, 2023.

“Connect Rest”, https://docs.confluent.io/platform/current/connect/references/restapi.html, last accessed 08 07, 2023.

Dhalla, H. K. (2020, November). “A Performance Analysis of Native JSON Parsers in Java, Python, MS. NET Core, JavaScript, and PHP”. In 2020 16th International Conference on Network and Service Management (CNSM) (pp. 1-5). IEEE.

Dossot, D. (2014). “RabbitMQ essentials”. Packt Publishing Ltd.

Drahoš, P., Kučera, E., Haffner, O., & Klimo, I. (2018, January). “Trends in industrial communication and OPC UA. In 2018 cybernetics & informatics” (K&I) (pp. 1-5). IEEE.

“Ferrovia”, http://ferrovia40.pt/, last accessed 08 07, 2023.

Happ, D., Karowski, N., Menzel, T., Handziski, V., & Wolisz, A. (2017). “Meeting IoT platform requirements with open pub/sub solutions”. Annals of Telecommunications, 72, 41-52.

“Kafka Connect MQTT”, https://github.com/johanvandevenne/kafka-connect-mqtt, last accessed 08 07, 2023.

“Kafka Consumer Config”, https://docs.confluent.io/platform/current/clients/javadocs/javadoc/org/apache/kafka/clients/consumer/ConsumerConfig.html, last accessed 08 07, 2023.

“Kafka Producer Config”, https://docs.confluent.io/platform/current/clients/javadocs/javadoc/org/apache/kafka/clients/producer/ProducerConfig.html, last accessed 08 07, 2023.

"Kafka QoS", https://docs.confluent.io/kafka/design/delivery-semantics.html, last accessed 08 07, 2023.

"Middleware Repo", https://bitbucket.org/pedroc4/multiprotocol-translator-for-iot/src/master/, last accessed 08 07, 2023.

"MQTT Connect Options", https://www.eclipse.org/paho/files/javadoc/org/eclipse/paho/client/mqttv3/MqttConnectOptions.html, last accessed 08 07, 2023.

"MQTT Flink Connector", https://github.com/ajiniesta/flink-connector-mqtt, last accessed 08 07, 2023.

"MQTT Page", https://mqtt.org/, last accessed 08 07, 2023.

"MQTT QoS", https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/, last accessed 08 07, 2023.

"MQTT Spark Connector", https://bahir.apache.org/docs/spark/current/spark-streaming-mqtt/, last accessed 08 07 2023.

Naik, N. (2017, October). "Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP". In 2017 IEEE international systems engineering symposium (ISSE) (pp. 1-7). IEEE.

Nam, J., Jun, Y., & Choi, M. (2022). "High Performance IoT Cloud Computing Framework Using Pub/Sub Techniques". Applied Sciences, 12(21), 11009.

"OPC-UA Pub/Sub", https://reference.opcfoundation.org/Core/Part14/v104/docs/5, last accessed 08 07 2023.

"RabbitMQ Connection Factory", https://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.ConnectionFactory.html, last accessed 08 07, 2023.

"Rabbit Management", https://www.rabbitmq.com/management.html, last accessed 08 07, 2023.

"Rabbit QoS", https://www.rabbitmq.com/confirms.html, last accessed 08 07, 2023.

"Samza Connectors", https://samza.apache.org/learn/documentation/1.6.0/connectors/overview.html, last accessed 08 07, 2023.

Tukade, T. M., & Banakar, R. (2018). "Data transfer protocols in IoT—An overview. Int. J. Pure Appl. Math", 118(16), 121-138.

Varga, P., Blomstedt, F., Ferreira, L. L., Eliasson, J., Johansson, M., Delsing, J., & de Soria, I. M. (2017). “Making system of systems interoperable–The core components of the arrowhead framework”. *Journal of Network and Computer Applications*, *81*, 85-95.

Vázquez-Ingelmo, A., García-Holgado, A., & García-Peñalvo, F. J. (2020, April). “C4 model in a Software Engineering subject to ease the comprehension of UML and the software”. In 2020 IEEE Global Engineering Education Conference (EDUCON) (pp. 919-924). IEEE.