Creating a message channel management system between messaging channels

Dimitrios Kalathas
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
Email: el18016@mail.ntua.gr

Dimitrios Kalemis
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
Email: el18152@mail.ntua.gr

Michael Kolliniatis
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
Email: el18901@mail.ntua.gr

***Abstract*—Over the years many developers and architects struggled through multiple integration vendors' product documentation just to realize later that many of the underlying concepts were quite similar. Due to this, Gregor Hohpe and Bobby Woolf created a book titled "Enterprise Integration Patterns" in which they compiled 65 integration patterns from various integration efforts. A straightforward "cookbook" method cannot be used to handle the complicated problem of enterprise integration. Instead, patterns can offer direction by capturing the kind of knowledge that ordinarily exists exclusively in the minds of architects: they are acknowledged answers to recurrent issues within a certain environment. Patterns are detailed enough to give designers and architects practical advice while remaining abstract enough to apply to the majority of integration technologies. Moreover, patterns give developers a vocabulary to effectively define their solution. Nowadays, it has become much easier to integrate and build these messaging patterns thanks to a variety of open-source frameworks.**

Keywords—Enterprise Integration Patterns, MQTT, AMQP, RabbitMQ, Eclipse Mosquitto, Apache Camel

I. Introduction

This project was part of the Analysis and Design of Information Systems course at NTUA and it is concerned with the development of a message channel management system. More specifically, we developed a system that creates and manages multiple data streams in real time, between heterogeneous systems. Concerning the created data streams, our team had to create data that would simulate the behaviour of sensor devices. The system through an appropriate REST API enables the user to specify a source adaptor (connection to the system that holds the messages) and destination adaptor (connection to the system receiving the messages) and implements a communication channel between the two. The system does not translate/process the messages, but only takes care of their transfer from the source (s) to the destination (d). For the implementation, RabbitMQ broker is used for the source and Mosquitto broker is used for the destination.

II. Source Code

Github link: https://github.com/jimk0099/Analysis-and-Design-of-Information-Systems

III. Diagrams

Fig. 1 shows the EIP diagram of the project and Fig. 2 shows the REST API endpoints diagram.

IV. Technology Stack

- Apache Camel
  Used as integration framework
- RabbitMQ
  Used as AMQP broker for the source
- Eclipse Mosquitto
  Used as MQTT broker for the destination
- SpringBoot
  Used as framework to develop the application
- Python3
  Used to create dummy data that represent the data from sensors that detect temperature, humid and atmospheric pressure and send them to the source adaptor
- Java 19
  Used to implement the majority of the system’s components and use cases

V. Tools

- Postman
  Used to test our API endpoints and write functional tests
- Spring initializr
  Used to generate a Spring Boot project
- GitHub
  Used for version control and better collaboration
- Visual Paradigm
  Used to model business information systems and manage development processes

VI. INSTALLATION

We are working with Linux Ubuntu 22.04, so the following installation steps refer to that system. We also use the dollar symbol ($) for normal user and the hash symbol (#) for the system administrator:

A. Clone Project

- Git clone the project from the repository
- Navigate to the folder src/main/java/com/jss
- Execute the CamelApplication.java file

B. RabbitMQ

- open a new terminal
- run *$ sudo apt-get update* and then *$ sudo apt-get upgrade* to update the system
- install the main dependency package for RabbitMQ by executing *$ sudo apt-get install erlang*
- install the RabbitMQ server with the command *$ sudo apt-get install rabbitmq-server*
- start RabbitMQ service with the command *$ sudo systemctl enable rabbitmq-server*
- and finally to start the RabbitMQ server we run *$ sudo systemctl start rabbitmq-server*

To check whether the broker is installed and runs successfully, we can run the command *$ sudo systemctl status rabbitmq-server*. We also need to enable the RabbitMQ management plugin to be able to monitor the broker via an HTTP-based API and access the browser-based UI, so we need to run the following command *$ sudo rabbitmq-plugins enable rabbitmq_management*. By default, RabbitMQ will listen on port 5672 on all available interfaces and the UI will listen on port 15672.

C. Eclipse Mosquitto

- open a new terminal
- run *$ sudo apt-get update* to update the system
- install mosquitto broker by executing *$ sudo apt-get install mosquitto*
- install mosquitto clients with the command *$ sudo apt-get install mosquitto-clients*

To check if the broker is installed and runs successfully, we can run the command *$ sudo systemctl status mosquitto* and ensure that the package is loaded and active. To stop the mosquitto service we may run *$ sudo systemctl stop mosquitto* and to start the service we can run *$ sudo systemctl start mosquitto.* The default configuration of Eclipse Mosquitto listens on port 1883.

D. Spring Boot

In order to create a Spring Boot stand-alone application, we used the spring initializr tool [https://start.spring.io/]. We created a Maven project in Java 19 with the Spring Web dependency so that we could easily build a RESTful app. More dependencies were added in the pom.xml such as Lombok and camel that were used to implement the system [1].

E. Apache Camel

To install the Apache Camel libraries we simply add them to the pom.xml file that can be found in the root folder of our project and the reload the project using the maven management tool. The Apache Camel version used is 3.19.0 and it has useful libraries like camel-rabbitmq and camel-paho that helped with the implementation of EIP and integration between the heterogeneous systems.

F. Python3 and Java 19

Python3 and Java 19 should be preinstalled on our Linux system. To ensure that we have successfully installed these prerequisites we may run the commands *$ python3 –version* and *$ java –version* at the terminal.

VII. REST API

A. API

An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between a provider of information and a user of that information, outlining the content that the consumer (the call) and the producer (the producer) are obligated to deliver (the response). For instance, the API design for a weather service may need the user to provide their zip code and the producer to respond with two parts, the first of which would be the high temperature and the second, the low. To put it another way, an API enables you to express your needs to a computer or system so that it can comprehend and carry out your request when you wish to engage with it to retrieve information or carry out a function [2].

B. REST

REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways. A RESTful API transmits a representation of the resource's state to the requester or endpoint when a client request is made through it. One of various formats, including JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text, is used to send this information or representation via HTTP. Despite its name, JSON is the most widely used file format because it can be read by both humans and machines and is language-independent [2].

VIII. RabbitMQ Broker

RabbitMQ is an open-source message-broker program (also known as message-oriented middleware) and it was the first which implemented the Advanced Message Queuing Protocol (AMQP). Since then, RabbitMQ has been expanded with a plug-in architecture to support the Streaming Text Oriented Messaging Protocol (STOMP), MQ Telemetry Transport (MQTT), and other protocols. The Open Telecom Platform framework for clustering and failover serves as the foundation for the RabbitMQ server, which is written in Erlang. For all popular programming languages, client libraries are available to interface with the broker. The Mozilla Public License is used to release the source code [3][4].

IX. Mosquitto Broker

Eclipse Mosquitto is an open source message broker that complies with the EPL/EDL license and supports MQTT versions 5.0, 3.1.1, and 3.1. Mosquito is portable and adaptable for usage on many types of equipment, from large servers to single board computers with modest power requirements. Using a publish/subscribe approach, the MQTT protocol offers a simple way to conduct messaging. Because of this, it can be used for Internet of Things messaging with low power sensors or mobile devices like phones, embedded computers, or microcontrollers. Together with the widely used command line MQTT clients mosquitto pub and mosquitto sub, the Mosquitto project also offers a C library for building MQTT clients [5].

X. AMQP Protocol

AMQP 0-9-1 (Advanced Message Queuing Protocol) is a messaging protocol that enables conforming client applications to communicate with conforming messaging middleware brokers. AMQP is a binary, application layer protocol, designed to efficiently support a wide variety of messaging applications and communication patterns. It provides flow controlled, message-oriented communication with message-delivery guarantees such as at-most-once (where each message is delivered once or never), at-least-once (where each message is certain to be delivered, but may do so multiple times) and exactly-once (where the message will always certainly arrive and do so only once), and authentication and/or encryption based on SASL and/or TLS. It assumes an underlying reliable transport layer protocol such as Transmission Control Protocol (TCP) [6].

XI. MQTT Protocol

MQTT is a lightweight publish-subscribe machine-to-machine network protocol for message queue/message queuing services. It was originally an acronym for MQ Telemetry Transport. It is intended for connections with distant sites that have resources-constrained or network bandwidth-constrained devices, such as in the Internet of Things (IoT). It must function via a transport protocol, commonly TCP/IP, that offers arranged, lossless, bidirectional connections. The ISO/IEC 20922 recommendation is an open OASIS standard [7].

XII. Apache Camel

Apache Camel is a Java based integration framework that we used to integrate the API with the other two messaging Channels [0]. This open source framework is based on known Enterprise Integration Patterns (EIP) and that is the main reason we chose to use it [8]. Our application's main part integrates what is depicted in Figure 1.

RESTful API: The application features a REST API that not only allows the user to create a message channel, but also perform management tasks. The various endpoints of the api are explained later in detail.

Making a Connection: The user is able to make a connection from a RabbitMQ endpoint to a Mosquitto endpoint, giving custom hosts, ports, topics and credentials. This connection can be created at runtime and the user may track any updates on the RabbitMQ user interface at http://localhost:15672/ with the default credentials which are “guest” as username and “guest” as password. When the user makes a custom connection using the appropriate API call he/she should be able to see the new connection under the Connections tab in the UI. At that time a new messaging channel should be created as well.

Connecting a Sensor: The python scripts located at /src/main/python simulate the behavior of a sensor that sends data regarding the environment (temperature, humid, atmospheric pressure). These scripts create a connection with the source adaptor (in our case RabbitMQ), generate the dummy data at a rate that can be customized by the user and sends them to a RabbitMQ exchange.

Connecting an MQTT receiver: Mosquitto broker comes with a simple MQTT version 5/3.1.1 client called “mosquitto_sub” that will subscribe to topics and print the messages it receives. In order for the mosquitto client to receive the messages, we open a new terminal and run the following command *$ mosquitto_sub -t message-topic* with the message-topic be the one defined in the connection.

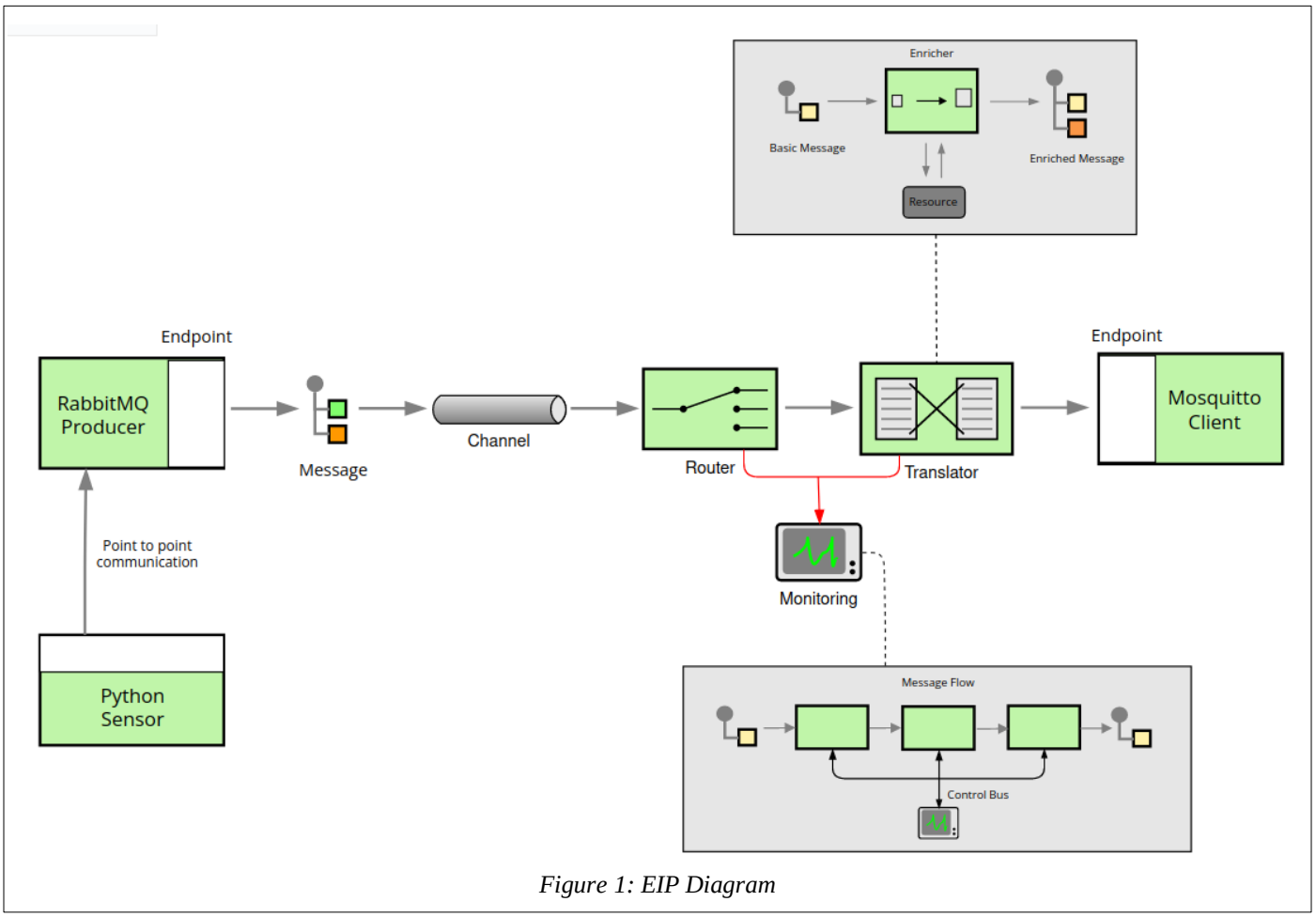

Figure 1: EIP Diagram

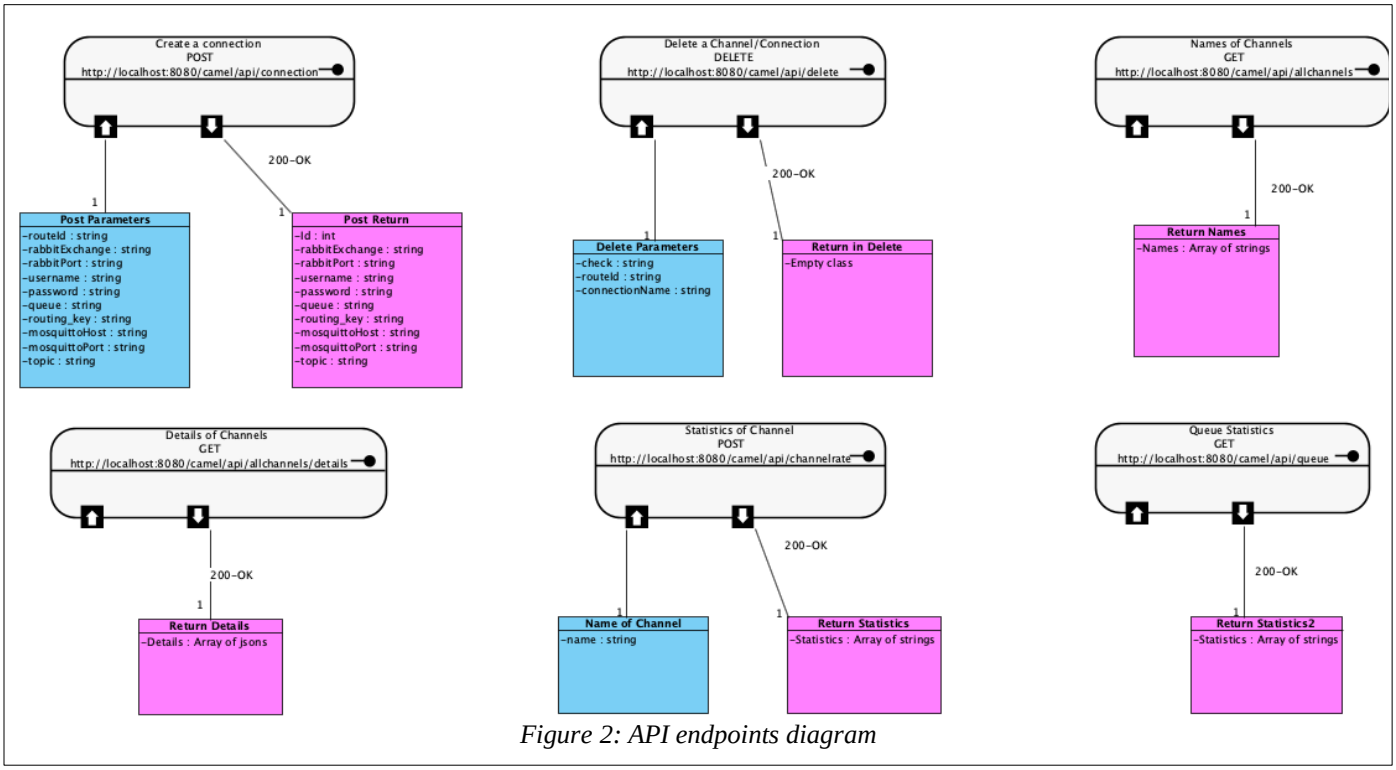

Figure 2: API endpoints diagram

XIII. Our REST API

Six endpoints make up our RESTful API, all of which are used to support the project's use cases as shown in Fig. 2 . Like we've said before, the REST API enables users to construct communication channels and carry out management operations. In more detail:

(i) Post request at the endpoint */connection*: The user can establish at runtime a fresh link between a RabbitMQ source adaptor and a Mosquitto destination adapter by using this endpoint. The system will build the connection in accordance with the user's choices if the user posts a json file with the connection information. More specifically, the host name and port for both theRabbitMQ and Mosquitto can be adjusted by the user. Also, the topic name for the destination system and the desired exchange, queue, and credentials for the source application can all be set. The user can set the routeId at the Apache Camel level to make management jobs easier.

(ii) Delete request at the endpoint /*delete:* Any connection that the system manages can be deleted by sending a delete
(iii) API request. The user may upload a json file with the connection's routeId and set the check string to "routeId" if the connection between the two brokers is what he/she wants to remove. On the other hand, a json with the desired connection name should be submitted with the check field having the value "connectionName" if the connection is between a sensor and the source application. In the second case, the system sends a second API delete request to the RabbitMQ Management HTTP API (http://localhost:15672/api) at the endpoint *connections/name* with the parameter name being the one defined by the user.

(iv) Get request at the endpoint */allchannels*: By hitting a get request at this endpoint, the user can receive all the active channels. This endpoint invokes the RabbitMQ Management HTTP API and sends a fresh Get request to its */channels* endpoint. In addition to obtaining a list of the currently active channels, the system also stores them in a file called "connsList1.json" for later use.

(v) Get request at the endpoint */allchannels/details*: This endpoint is implemented similarly to endpoint (iii) and is similar in functionality. The distinction is that we now get more information about each channel in addition to just the name of the ones that are currently active.

(vi) Post request at the endpoint */channelrate*: The user may obtain statistics for a particular messaging channel using this endpoint. The RESTful API of our application should receive a json file containing the channel name as a request, and from there, the system will call the RabbitMQ API at the */channels/name* endpoint with the specified name. The application processes the returned json object and stores it locally in a file called "conns1.json," from which it extracts the total number of messages sent by the channel and the rate at which they were sent. If the channel is inactive, the system sends back a different message telling the user that the channel in concern is not active right now.

(vii) Get request at the endpoint */queue*: All available queue statistics on our system can be obtained using this API. These details are also available in the RabbitMQ UI, but our system analyses the data and provides it in a format that is simple to read. Once more, we obtain the array containing the json object referring to the queues using the RabbitMQ Management HTTP API. If the queue is presently sending messages, we report the outgoing message rate; otherwise, if the queue is currently storing messages, we return both the total number of messages that have been stored and the rate at which they are being stored. In any case, we display the overall deliveries since each queue was established. It is also important to keep in mind that the system keeps track of all these statistics in a local file called "queue.json" that can be accessed in /src/main/other.

XIV. Scenario

To demonstrate the use cases of our program, we create a custom scenario that includes screenshots of both the RabbitMQ UI and postman API calls.

(i) Using a post request and the connection characteristics, we establish a connection. The new connection is established, and its operating status is displayed in the RabbitMQ user interface.

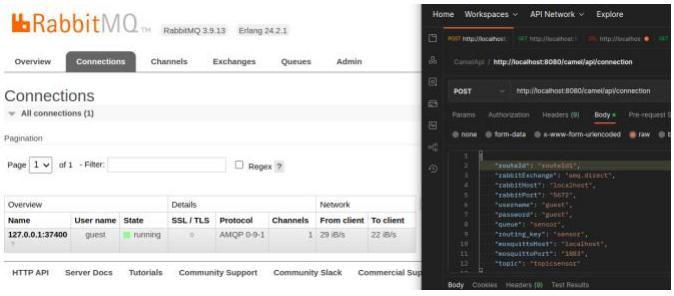

Figure 3

A messaging channel is created at the same time and is accessible via the Channels tab of the RabbitMQ service.

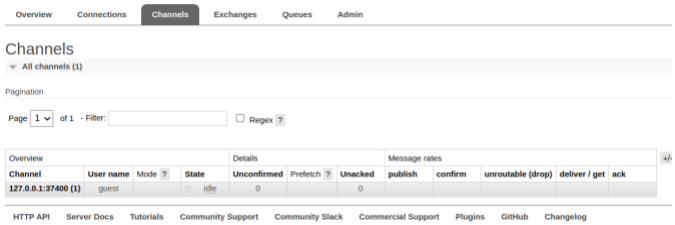

Figure 4

(ii) Here, we see the delete endpoint in action. In the first screenshot, a channel between RabbitMQ and Eclipse Mosquitto is deleted with the routeId as the post object.

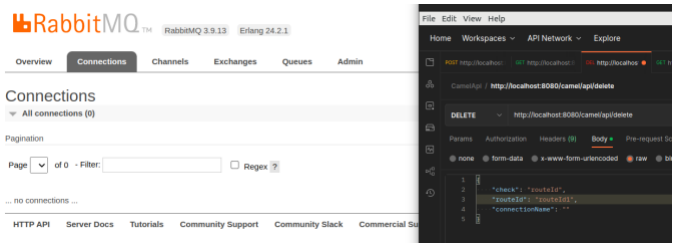

Figure 5

The delete endpoint is demonstrated here.
The routeId is used as the post object to remove a channel between RabbitMQ and Eclipse Mosquitto in the first screenshot.

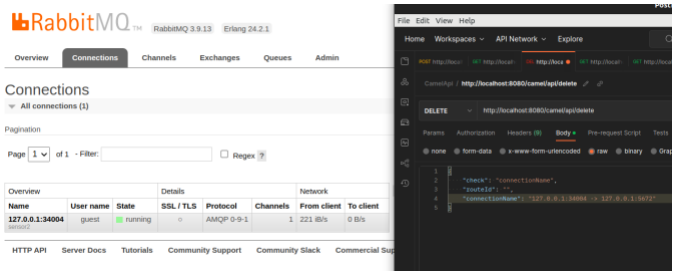

Figure 6

After delete request:

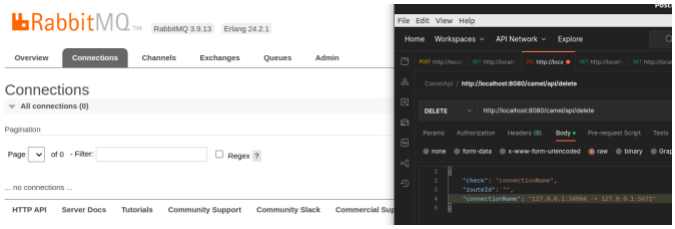

Figure 7

(iii) We issue a get request to the */allchanells* endpoint, as seen below in Postman, in order to obtain all of the accessible channels. By looking at the user interface, we can confirm that we do, in fact, receive all of the active connections.

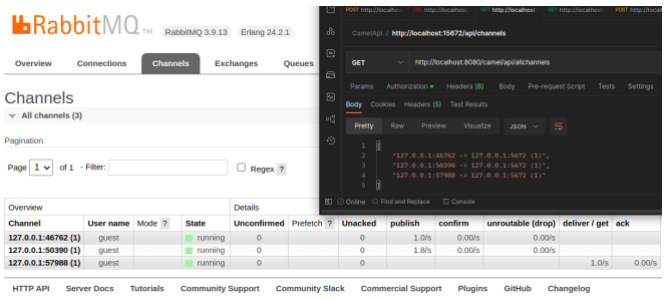

Figure 8

(iv) We send a get request to the endpoint */allchannels/details* as shown below to obtain more information about the currently accessible channels.

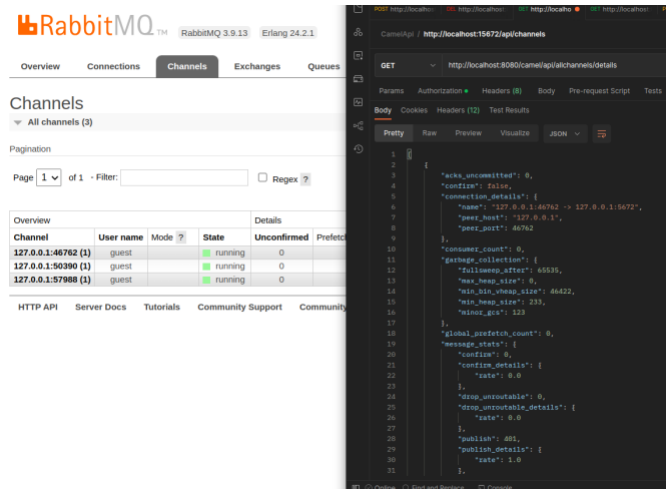

Figure 9

(v) Send a message to the /channelrate endpoint. With this endpoint, we may get statistics for a specific message channel, as seen in the accompanying screenshot. Here, we get messaging rates for the channel "127.0.0.1:50390 -> 127.0.0.1:5672 (1)", which is currently up and running, so we get the corresponding message.

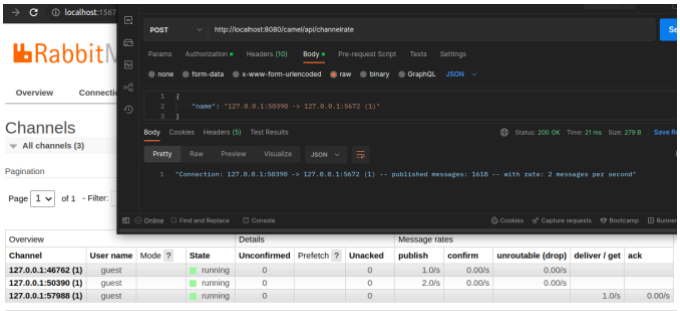

Figure 10

(vi) Using the endpoint */queue*, we may obtain the queues' current state in real time. The queues are keeping track of the messages they receive from the sensors even when the connection between RabbitMQ and Mosquitto is not active. In the scenario below, both queues save the messages that were received, resulting in a message like this :

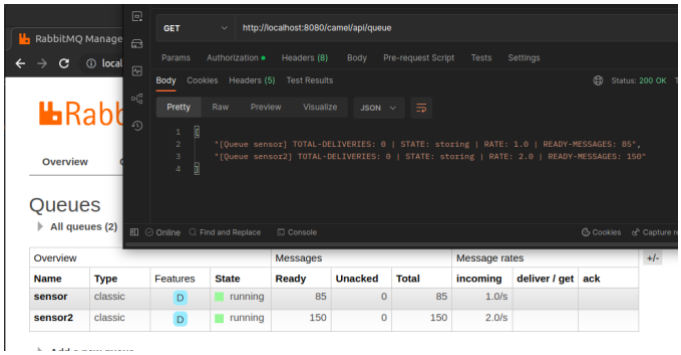

Figure 11

On the other hand, when the connection is active, the queues deliver the messages to the Eclipse Mosquitto client. Now the "sensor" queue serves an active connection, so the corresponding message is the following:

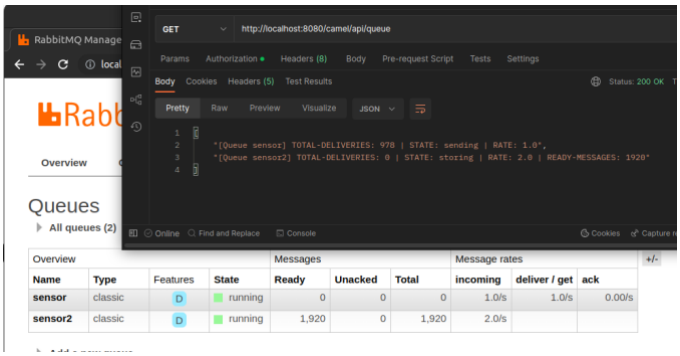

Figure 12

Message exchange use case

Below, we perform an example that exchanges messages between a sensor that is connected to our source (RabbitMQ) and our destination (Mosquitto client). Firstly, we run the Apache Camel Application and create a messaging channel with a post request at our API as shown above in the first request. By entering the user interface, we should see under the connections tab a new connection and under the channels tab a new channel.

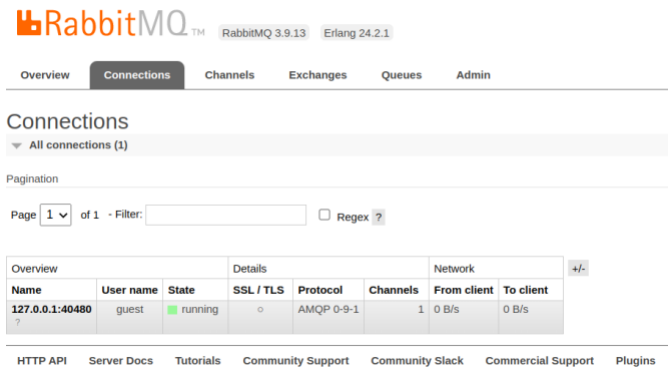

Figure 13

Then, after specifying the desired CLI options, we execute the sensorConfig.py file. This will connect the sensor to the RabbitMQ broker and enable message sending from the sensor. In the view below, we can see the messages our sensor transmitted as well as the newly established connection:

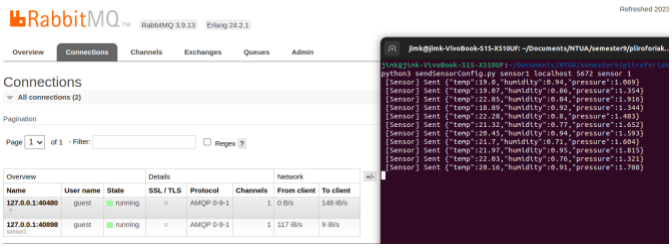

Figure 14

The Mosquitto client needs to be connected at this point. To accomplish that, we must recollect the subject we used to support the link between the two brokers. In our situation, the topic is "topicsensor," hence our client must subscribe to that topic. In addition to specifying the topic we want to subscribe to, we run the command mosquitto sub. By doing this, a straightforward MQTT version 5/3.1.1 client will be created, which will subscribe to our topic and print any messages it gets.

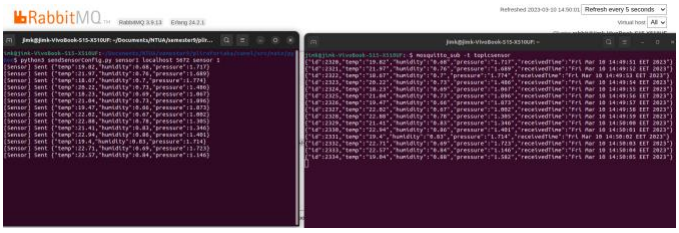

Figure 15

The Mosquitto client is visible on the right terminal; it not only receives messages from the sensor but also appends a timestamp to each message so that we can determine the exact time the client received it.

As seen in the sample below, many sensors can simultaneously send messages to the destination topic.

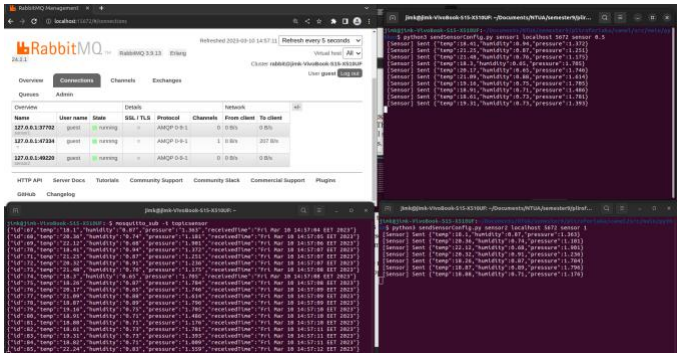

Figure 16

XV. Conclusion

Data can be transferred and processed asynchronously between two components by using messaging. The producers, the broker, and the customers are the elements that take part in this integration. Enterprise Integration Patterns makes it possible to use an event-driven architecture and all of its **benefits**, without the need to reinvent the wheel [10]:

- Agility because many teams can separately construct the producer/consumer components.
- Improved deployment due to the autonomous deployment of each producer/consumer component
- Due to asynchronous producer/consumer components, high performance
- Excellent scalability due to the ability to add more producers and consumers to spread the processing load.
- The component that generates the message and the one that consumes it are not coupled. They don't have to be available at the same time or be aware of each other's private information.
- As a producer component may swiftly emit a message when there is a change, consumers will be informed and processed right away, reducing the risk of data staleness.

However, employing EIP integration methods may have the following drawbacks:

- It could be challenging to test the message flow between the various producer and consumer components since numerous components might be handling the same message at once.

- Given that the asynchronous processing of the data is dispersed among numerous producer/consumer components, complexity could be enhanced and it might be challenging to comprehend the entire data flow. Moreover, we must manage programming learning curves and message processing issues.

XVI. Useful Terms

- *Message Channel:* The Message Channel is an internal implementation detail of the Endpoint interface, where all interactions of the channel is via the Endpoint.

- *Message:* The message is the data record that represents the message part of the exchange.

- *ControlBus:* The Control Bus from the EIP patterns allows for the integration system to be monitored and managed from within the framework [8].

- *Producer:* A program that sends messages is a producer.

- *Consumer:* A program that mostly waits to receive messages.

- *Queue:* A sequential data structure with two primary operations: an item can be enqueued (added) at the tail and dequeued (consumed) from the head. Queues play a prominent role in the messaging technology space: many messaging protocols and tools assume that publishers and consumers communicate using a queue-like storage mechanism.

- *Exchange:* RabbitMQ exchanges are message routing agents configured by the virtual host in RabbitMQ. Exchange is in charge of routing messages to different queues using header attributes, bindings, and routing keys [4].

- *Topic:* MQTT topics are a form of addressing that allows MQTT clients to share information [5].

XVII. REFERENCES

[1] *Spring Boot*. (n.d.). Spring Boot. https://spring.io/projects/spring-boot

[2] *What is a REST API?* (n.d.). https://www.redhat.com/en/topics/api/what-is-a-rest-api

[3] Wikipedia contributors. (2023, February 13). *RabbitMQ*. Wikipedia. https://en.wikipedia.org/wiki/RabbitMQ

[4] Messaging that just works — RabbitMQ. (n.d.). https://www.rabbitmq.com/

[5] *Eclipse Mosquitto*. (2018, January 8). Eclipse Mosquitto. https://mosquitto.org/

[6] Wikipedia contributors. (2023a, February 9). *Advanced Message Queuing Protocol*. Wikipedia. https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

[7] Wikipedia contributors. (2023c, March 4). *MQTT*. Wikipedia. https://en.wikipedia.org/wiki/MQTT

[8] *Messaging Patterns Overview - Enterprise Integration Patterns*. (n.d.). Enterprise Integration Patterns. https://www.enterpriseintegrationpatterns.com/patterns/messaging/

[9] Apache Camel. (n.d.). *Documentation*. https://camel.apache.org/docs/

[10] A. (2021, September 3). *Pros & Cons of Different Enterprise Integration Patterns - Wizeline*. Wizeline. https://www.wizeline.com/pros-cons-of-different-enterprise-integration-patterns/