INTEROPERABILITY AMONG EVENT-DRIVEN MICROSERVICE-BASED SYSTEMS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALI BAYRAMÇAVUŞ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

FEBRUARY 2022

Approval of the thesis:

INTEROPERABILITY AMONG EVENT-DRIVEN MICROSERVICE-BASED SYSTEMS

submitted by **ALI BAYRAMÇAVUŞ** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Prof. Dr. Ali H. Doğru
Supervisor, **Computer Engineering, METU** _____

Dr. M. Çağrı Kaya
Co-supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Assist. Prof. Dr. Pelin Angın
Computer Engineering, METU _____

Prof. Dr. Ali H. Doğru
Computer Engineering, METU _____

Assist. Prof. Dr. Selma Nazlıoğlu
Software Engineering, Atılım University _____

Date: 10.02.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Ali Bayramçavuş

Signature :

ABSTRACT

INTEROPERABILITY AMONG EVENT-DRIVEN MICROSERVICE-BASED SYSTEMS

Bayramçavuş, Ali
M.S., Department of Computer Engineering
Supervisor: Prof. Dr. Ali H. Doğru
Co-Supervisor: Dr. M. Çağrı Kaya

February 2022, 77 pages

This work presents our proposed solution to provide interoperability among systems that have event-driven microservice architecture using different middleware technologies. Publish/subscribe technology is an essential part of event-driven architectures, and these technologies, specifically through Kafka and RabbitMQ, are targeted in this work. Our interoperability tool proposes a way to solve interoperability problems, as a microservice platform allowing more than two systems to work together. Experiments, which are conducted incorporating Kafka-based and RabbitMQ-based systems, prove the applicability of the proposed interoperability tool under stress when subject to tens of thousands of bi-directional messages transmitted per second.

Keywords: interoperability, event-driven systems, microservices

ÖZ

OLAY TABANLI MİKROSERVİS ESASLI SİSTEMLER ARASINDA BİRLİKTE ÇALIŞABİLİRLİK

Bayramçavuş, Ali
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Ali H. Doğru
Ortak Tez Yöneticisi: Dr. M. Çağrı Kaya

Şubat 2022 , 77 sayfa

Bu çalışma, farklı ara katman teknolojileri kullanan olay odaklı mikroservis mimarisine sahip sistemler arasında birlikte çalışabilirliği sağlamak için önerdiğimiz çözümü sunmaktadır. Yayınlama/abone olma teknolojisi, olay odaklı mimarilerin önemli bir parçasıdır ve bu teknolojiler, özellikle Kafka ve RabbitMQ, bu çalışmada hedeflenmiştir. Birlikte çalışabilirlik aracımız, ikiden fazla sistemin birlikte çalışmasına izin veren bir mikroservis platformu olarak birlikte çalışabilirlik sorunlarını çözmenin bir yolunu önermektedir. Kafka tabanlı ve RabbitMQ tabanlı sistemler arasında yapılan deneyler, önerilen birlikte çalışabilirlik aracının, saniyede iletilen on binlerce çift yönlü mesaja maruz kaldığında stres altında uygulanabilirliğini kanıtlamaktadır.

Anahtar Kelimeler: birlikte çalışabilirlik, olay tabanlı sistemler, mikroservisler

To my family

ACKNOWLEDGMENTS

I would like to thank my advisor Prof. Dr. Ali H. Doğru for his constant support, friendship, and guidance. He is a real gentleman and beyond being my advisor, he is a role model both for my academic and family life.

I would like to thank Dr. M. Çağrı Kaya. This thesis would not be possible without his support, help, and motivation. I really appreciate your help throughout my master's journey. You are one of the kindest persons I have ever met.

Lastly, I would like to express my gratitude to my dear family members; my mother Fatma Bayramçavuş, my father Bahadır Bayramçavuş, my sister Elif Bayramçavuş, and my brother Ömer Bayramçavuş. Thank you for your constant support and your belief in me.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vi

ACKNOWLEDGMENTS viii

TABLE OF CONTENTS ix

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF ABBREVIATIONS xv

CHAPTERS

1 INTRODUCTION 1

1.1 Background 1

1.2 Problem Statement 2

1.3 Approach 2

1.4 Contribution 3

1.5 Outline of Thesis 3

2 BACKGROUND AND RELATED WORK 5

2.1 Microservice Architecture 5

2.1.1 Doing a Small Thing but Doing It Exactly Right 5

2.1.2 Benefits of Microservice Architecture 6

2.2 Docker 8
2.2.1 Benefits of Docker 9
2.2.1.1 Works on My Machine Problem 9
2.2.1.2 Isolated environments 9
2.2.1.3 Development 10
2.2.1.4 Scaling 10
2.2.2 Virtual Machine 10
2.3 Event-Driven Architecture 11
2.3.1 Benefits of an event-driven architecture 12
2.3.1.1 Scale and fail independently 12
2.3.1.2 Develop with agility 12
2.3.1.3 Audit with ease 14
2.3.1.4 Cut costs 14
2.3.2 Event Router 14
2.3.2.1 Point-to-Point Messaging Pattern 15
2.3.2.2 Publish/Subscribe Messaging Pattern 15
2.4 Interoperability 17
2.4.1 Importance of Interoperability 18
3 PROPOSED TOOL 19
3.1 The Problem 19
3.2 The Solution 22
3.3 The Proposed Tool in Detail 23
3.3.1 Converter Configurator 26

3.3.1.1 registerConverter . . . 27
3.3.1.2 publishToCommonBroker . . . 27
3.3.1.3 subscribeToCommonBroker . . . 28
3.3.2 Middleware Converter . . . 29
3.3.2.1 publishToCommon . . . 30
3.3.2.2 publishToSystem . . . 30
3.3.3 Common Middleware . . . 30
3.4 Example . . . 31
3.5 How to Extend . . . 37
3.5.1 How to Implement a New Middleware Converter . . . 38
4 CASE STUDY - INTEROPERABILITY AMONG ACCESS CONTROL SYSTEMS . . . 41
4.1 Access Control System . . . 41
4.2 Our Scenario . . . 42
4.3 Test Setup . . . 43
4.3.1 Simulation with Docker . . . 52
4.4 Discussion and Test Results . . . 52
5 CONCLUSION AND FUTURE WORK . . . 57
5.1 Conclusion . . . 57
5.2 Future Work . . . 57
REFERENCES . . . 59
APPENDICES
A DOCKERFILES . . . 63

A.1 Publisher and Subscriber Services for System using RabbitMQ as Message Broker 63

A.2 Publisher and Subscriber Services for System using Kafka as Message Broker 63

A.3 Middleware Converter for System using RabbitMQ as Message Broker 64

A.4 Middleware Converter for System using Kafka as Message Broker . . 64

A.5 Converter Configurator 65

A.6 Log Manager 65

B DOCKER-COMPOSE FILES 67

B.1 Starting All Message Broker Applications in Each System and Interoperability Tool 67

B.2 Starting Services of First Access Control System 69

B.3 Starting Services of Second Access Control System 71

B.4 Starting Log Service 73

B.5 Starting Converter Configurator and Middleware Converters of the Interoperability Tool 75

LIST OF TABLES

TABLES

Table 4.1 Average Time Delays Between Publishers And Subscribers in Milliseconds 53

LIST OF FIGURES

FIGURES

Figure 2.1 A High-level View for Docker 8

Figure 2.2 Containers vs Virtual Machines 11

Figure 2.3 Event-Driven Architecture Example 13

Figure 2.4 Publish/Subscribe System Example 16

Figure 3.1 Problem Overview 20

Figure 3.2 Extended Problem Overview 20

Figure 3.3 Complex Version of the Problem 22

Figure 3.4 Tool in Action 23

Figure 3.5 Tool in Detail 24

Figure 3.6 Activity Diagram of the Tool 25

Figure 3.7 Component Diagram of Given Example 33

Figure 3.8 Sequence Diagram of Given Example 34

Figure 3.9 Activity Flow for Implementing A New Middleware Converter . 39

Figure 4.1 Component Diagram of Case Study 45

Figure 4.2 Sequence Diagram of Case Study 46

LIST OF ABBREVIATIONS

PUB/SUB	Publish/Subscribe
REST	Representational State Transfer
API	Application Programming Interface
PAAS	Platform As A Service
OS	Operating System
CPU	Central Processing Unit
SSL	Secure Sockets Layer
TLS	Transport Layer Security
JSON	JavaScript Object Notation
XML	Extensible Markup Language
IT	Information Technology
CRM	Customer Relationship Management

CHAPTER 1

INTRODUCTION

1.1 Background

Microservice architecture [1, 2] has gained lots of attention in software engineering as a development paradigm. Small and autonomous services are the building blocks of a microservice architecture. Each service should be self-contained and should implement a single business capability within a bounded context. A bounded context [3] can be defined as a natural division within a business and it provides an explicit boundary within which a domain model exists.

Microservice architecture is a type of application architecture where the application is developed as a collection of services. It provides the framework to develop, deploy, and maintain microservice architecture services independently. Microservices [3] are small, independent, and loosely coupled. A single team of developers should be able to write and maintain a service. Each service is a separate codebase, not related to any codebases of the other services, and this lets developers work on a small codebase rather than a giant codebase, that is why this codebase is easy to be managed by a small development team. Each service can be deployed independently. A team can update an existing service they are responsible, without rebuilding and redeploying the entire application. Each service is responsible for persisting its own data or external state, this differs from the traditional model where a separate data layer handles data persistence. Communication between services is done by using well-defined interfaces. Internal implementation details of each service are hidden from other services. Microservice architecture supports polyglot programming; services do not need to share the same technology stack, libraries, or frameworks.

Event-driven microservice architecture [4, 5] is another architectural style that lets architects create highly decoupled services with the help of middleware and message broker [6] technologies. In this style, each service is only aware of the middleware, not any of the other services, and only communicates with the middleware. This makes the entire application to be more decoupled, which is a good feature to have for the applications.

1.2 Problem Statement

The number of applications for handling messaging among microservices in an event-driven architecture is increasing every day, each one solving a more specific or general problem for messaging among microservices. This increasing number of applications creates a heterogeneity problem when we are required to integrate multiple systems using different messaging applications. In this work, we propose a solution, that is extensible for any messaging applications for this heterogeneity problem.

1.3 Approach

We were heavily influenced by the event-driven microservice architecture [4, 5] while designing our proposed interoperability tool because our main goal is to make event-driven microservice applications interoperable and that is why we have emphasized this architecture. Being influenced by this architecture has let us create a tool that can be extended to other message brokers which are not yet considered to be used by applications we are trying to integrate.

We have designed our proposed interoperability tool also based on the microservice architecture. The tool composes different components, the communication between components is usually done through a message broker which is also a component in the tool, however, user-specified commands and some internal commands are received and forwarded among components using the APIs [7] exposed by each component. The message forwarding among the applications we are trying to make interoperable is conducted by the message broker of the proposed tool, so the main job is

done using an publish/subscribe mechanism.

1.4 Contribution

The contribution of this thesis is twofold: Firstly, the architecture we used to handle the interoperability problem is presented. Secondly, the tool, which is created based on the architecture, is implemented and tested with a real-life scenario.

1.5 Outline of Thesis

In chapter 2, some background information is provided on Microservice Architecture, Docker, Event-Driven Architecture, and Interoperability.

In chapter 3, we described the problem we tried to solve in detail, then presented our solution in depth, presented a very small example about how to use the tool, and lastly, we tried to explain how anyone can extend the capabilities of our proposed interoperability tool.

In chapter 4, the case study is explained in detail. Some background information about access control systems is provided to better explain the case study. The case study scenario, where we were trying to make two different systems interoperable, is described. We then explained how we conducted our test for this case study, and how we simulated case study using Docker. And lastly, we presented our thoughts about the results got from the tests.

We concluded this thesis with chapter 5, and some future work ideas are also given in that chapter.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, some background information is provided on microservice architecture, Docker, event-driven architecture, and interoperability.

2.1 Microservice Architecture

Microservice architecture [3] is an architectural style that structures an application as a collection of services that are

- Highly maintainable and testable
- Loosely coupled
- Independently deployable
- Organized around business capabilities
- Owned by a small team

Fast and reliable delivery of complex applications is handled easily using microservice architecture patterns. It also lets an organization evolve its technology stack since each microservice could be deployed independently.

2.1.1 Doing a Small Thing but Doing It Exactly Right

In monolithic applications [8], the codebase grows as the number of new features is added to the application. Even with a very clear and clever design of a monolithic application, the arbitrary in-process boundaries break down over time. Similar methods'

code becomes spread all over the codebase, that is why fixing a bug or implementing a new feature becomes harder and harder over time in monolithic applications' codebases.

Software architects and developers fight against these forces to ensure their code is more cohesive by creating abstractions or modules. Cohesion is to have related code grouped together. When we think about microservices, cohesion is an arbitrary concept. Robert C. Martin's definition of the *Single Responsibility Principle* [9] states "Gathering together those things that change for the same reason, and separate those things that change for different reasons.", which also reinforces the importance of cohesion.

In microservice architecture, the same approach is taken to independent services. Service boundaries are focused on business boundaries to make it more clear where the code lives for a given set of functionality. Keeping services focused on explicit boundaries enables us to not grow the code so much so that we do not have to deal with all of the difficulties associated with the big codebase.

2.1.2 Benefits of Microservice Architecture

There are various benefits of microservices. The key benefits of microservices are

- **Technology Heterogeneity.** We are free to choose any technology while developing a service since it will be a part of a bigger system composed of many collaborating services. The only important thing is that the microservice does what it is supposed to do, so it is not important what technology stack it uses from the perspective of other services. This allows developers to pick the right tool for each job, rather than having to select a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator.
- **Scalability.** In monolithic applications, we have to scale everything together. If we are required to scale a small part of the overall system because of a performance issue, and if that part is scattered around the whole monolith, we have to scale everything together as a piece. However with small independent services, if one part needs scaling, we have to only scale the microservice that handles

that part of the overall system. This allows developers to make the right decision about where to do the scaling, and lets them work on this scaling issue faster since the code for that service is never going to be a messy code like some monolith's code, due to its poor cohesion.

- **Easy Deployment.** A very small change in the code of a million-line-long monolithic application requires the whole application to be deployed in order to release the change. Since the whole application is going to change by this very small intervention, it is a high-risk deployment. That is why these high-risk deployments happen very infrequently because of the fear of making an undesirable error. This means that changes build up between releases until the new version of the application goes to production with lots of changes. This makes the difference between the new version and the old version bigger; the bigger the difference between versions, the higher the risk that something will get wrong. Since each microservice is deployed separately from other microservices, we can make our changes on the related microservice and just deploy its new version without affecting the other services. This allows us to get our code deployed faster with minimum risk of creating an error for the whole system. And if a problem occurs, we know the source, which is the new version of the microservice we deployed, so we can easily go back to the previous deployment of that service to make a fast rollback. This means we can get new features to production faster.

- **Replaceability.** In big organizations, there is always a big legacy system [10] every developer avoids touching. These big old legacy systems are not easy to replace because they were developed using very old technologies no one knows about today and most probably they run on very specific hardware which is very hard to find nowadays. It is extremely risky to try to replace it with a new version because it is a big and risky job in case of failure. However, with microservice architecture, each service is already very small and failure of one microservice does not mean the whole functionality of the whole system is hampered. That is why replacing a microservice is very easy compared to replacing a monolithic application and it is not a risky job at all.

- **Organizational Alignment.** Smaller teams working on small codebases are

much more productive than bigger teams working on the same big codebase. We can align our microservice architecture to our organization architecture, which lets us minimize the number of people working on any codebase to hit the nice spot of team size and productivity.

2.2 Docker

Docker [11, 12, 13, 14] is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers.

We can get two definitions:

1. Docker is a set of tools to deliver software in containers.
2. Containers are packages of software.

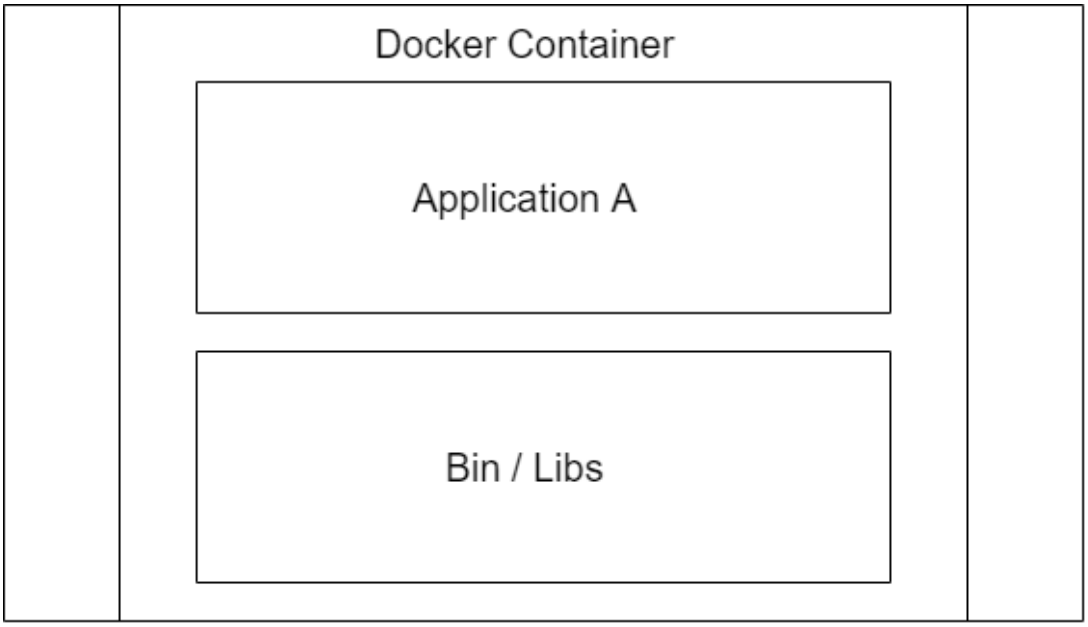

Figure 2.1: A High-level View for Docker (adapted from [11])

How containers include the application and its dependencies are illustrated in Figure 2.1. With containers we can create an isolated environment for the application running inside the container, the applications running on separate containers never interfere with each other, or with the application running outside of the containers. In cases

we need the applications running on different containers to interact with each other, Docker offers tools to do so.

2.2.1 Benefits of Docker

Containers package applications and this brings many benefits. In the following sections we can examine these benefits with a couple of scenarios:

2.2.1.1 Works on My Machine Problem

When developers finish the application, the application works on their machine, it may even go through a testing pipeline. However when the application is sent to the server, it fails. This is known as the **"works on my machine problem"**. One way to solve this problem is that installing every other software developers used in their machine to the server machine, and developers' machines are always loaded with all kinds of other tools that are maybe unnecessary for the application they are developing. This solution is very cumbersome and even if after this solution the application may still fail on the server because of different OS-level configurations. Containers are the best solution for this problem. If the developers run the application on a container on their machine, and if it successfully runs on that container, then we are good to go. We can just install that container to the server machine, since the container already has all of the necessary tools packaged, we do not have to do anything else.

2.2.1.2 Isolated environments

When we have a couple of different applications that were developed for example with Python, and we need to deploy these Python applications to a server that already has another application requiring Python 2.7 but our Python application requires a different version of Python. Installing a different version of a programming language on the same machine and expecting applications just figure out which version it should use would cause a disaster, and probably none of them will work as expected. How-

ever if we package each application in a container with the necessary Python version, and then deploy all of the containers to the server machine, everything should work smoothly.

2.2.1.3 Development

When a developer joins a team, he/she has to install all of the required services like postgres database, redis, mongodb etc. just to make the developed application run. However it is not enough to just install all of them, h/she also has to install the right versions the team uses. It is a painfully long and easily failed process for just starting to work. Thanks to containers, we can provide all of the necessary services and tools to new developers easily by just installing the containers in his/her machine.

2.2.1.4 Scaling

Starting and stopping docker containers are extremely fast because they have little overhead. However big and complex microservice systems used by millions of people sometimes get overloaded, and they need to meet the demand for a better user experience. In this case, we can first identify which microservices are overloaded and then spin up more containers which packaged these microservices, so that the load is distributed among all of the containers running the same microservice. Spinning up more than one for the same container is also beneficial in case of a container failure.

2.2.2 Virtual Machine

All of the benefits mentioned in the previous section have seemed like solved with virtual machines [15]. They solve the similar problem but they are not the same as Containers. Figure 2.2 gives a rough idea of the difference between containers and virtual machines.

There is an efficiency difference between a virtual machine and a docker solution for moving Application A to an incompatible system “Operating System B”. Running

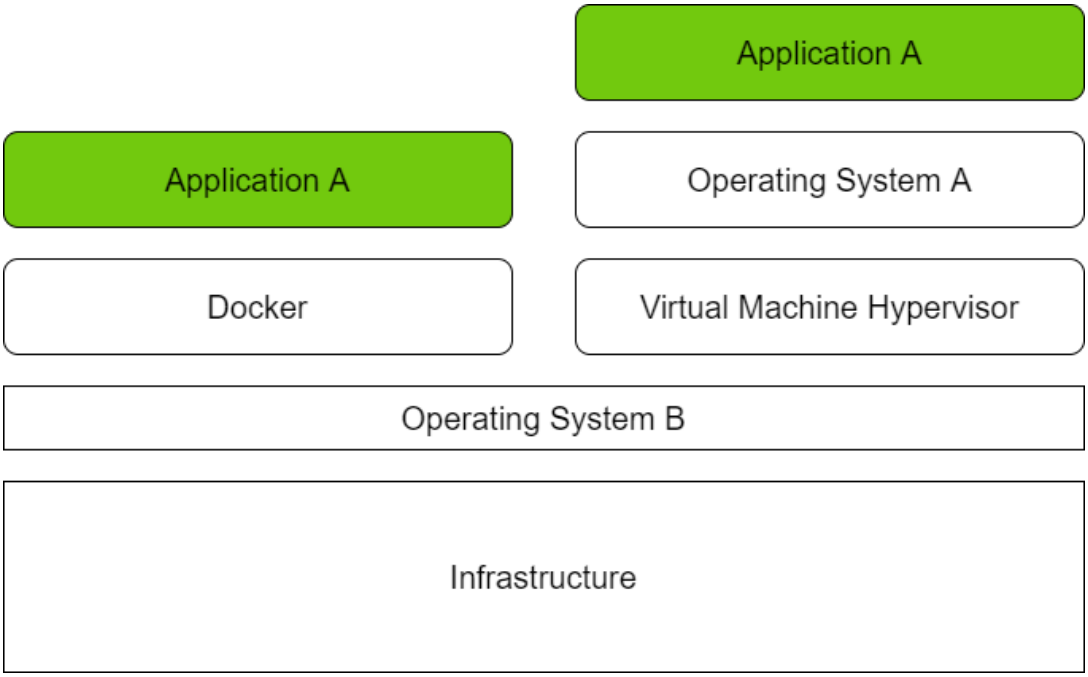

Figure 2.2: Containers vs Virtual Machines (adapted from [11])

software on top of containers is almost as efficient as running it "natively" outside containers. This is quite an advantage when compared to virtual machines.

2.3 Event-Driven Architecture

Event-driven architecture [4, 5] based on microservices is very common in modern applications. The main reason why it is used commonly is that it lets software architects create highly decoupled services. In event-driven architecture, events [5] are the main way to trigger microservices and create a communication style among microservices. An event can be described as a change in the state, or an update, like an item being placed in a shopping cart in an e-commerce website. With events we can process the state such as the item purchased, its price, and a delivery address, or we can use events as identifiers, a notification that an order was shipped.

There are three main components in event-driven architecture:

- **Event Producer.** It publishes events to the router.

- **Event Router.** It filters events published by event producers and pushes the events to the related consumers.
- **Event Consumer.** It receives the pushed events from the router.

The way all communication happens between services, which are either event producers or event consumers, is via the event router. Microservices do not know anything about other microservices; they are only communicating with the event router. That is how highly decoupled microservices are created. Creating a highly decoupled microservices application is an important architecture task because it lets the application be scaled, updated, and deployed independently.

In Figure 2.3, we can see an example of an event-driven architecture for an e-commerce site. With this architecture, the e-commerce application reacts to changes from a variety of sources during times of peak demand, without crashing the application or over-provisioning resources.

2.3.1 Benefits of an event-driven architecture

2.3.1.1 Scale and fail independently

With event-driven architecture, we make our microservices to be only aware of the event router, not each other. This means our services can communicate with each other since they are all using the same event router, and if one of the services fails, the rest will keep running.

2.3.1.2 Develop with agility

With event routers, developers are no longer need to write custom code to poll, filter, and route events between microservices, all of these jobs are automatically done by the event router. The event router will filter and push events, produced from event producers to related event consumers. Using an event router removes the need for heavy coordination between producer and consumer services, that is how the development process speeds up.

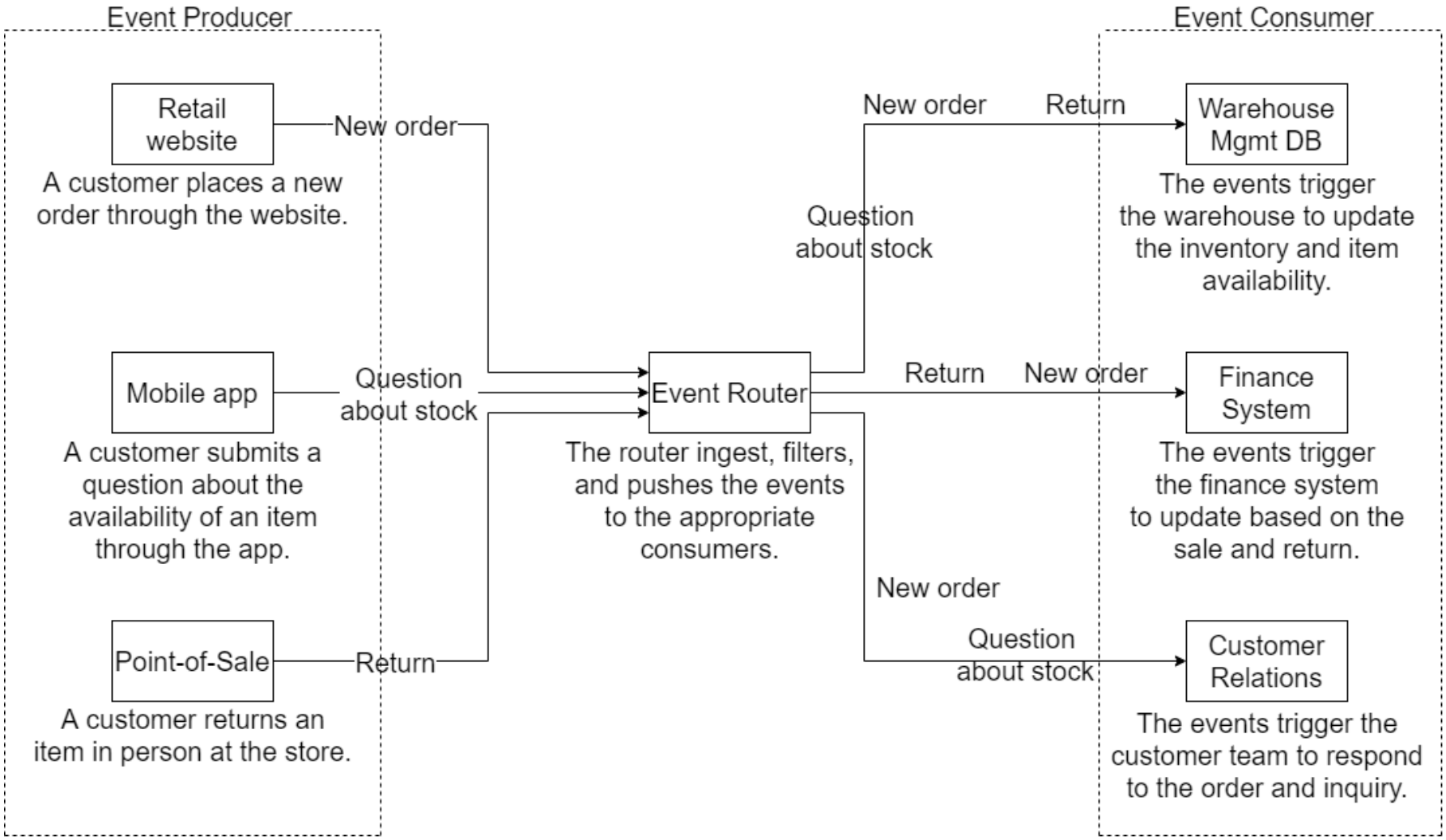

Figure 2.3: Event-Driven Architecture Example(adapted from [5])

2.3.1.3 Audit with ease

Since the event router sits in between each microservice in our microservices system, it can be used as a centralized location to audit our application and define policies. We can define policies on the event router to restrict who can publish and subscribe to the router and control which users and resources have permission to access data. We can add encryption mechanisms to our events both in transit and at rest.

2.3.1.4 Cut costs

Event-driven architecture is based on a push mechanism rather than a poll, so everything happens on-demand as the event presents itself in the router. Being push-based has many advantages, one of them is our services do not have to continuously poll the router for an event. This means less network bandwidth consumption, less CPU utilization, less idle fleet capacity, and fewer SSL/TLS handshakes. This also means microservices developed in event-driven architecture are more light-weighted.

2.3.2 Event Router

Event routers, commonly known as message brokers [6], are an inter-application communication technology to help build a common integration mechanism to support cloud-native, microservices-based, serverless, and hybrid cloud applications.

A message broker is a software that enables systems, applications, and services to communicate with each other to exchange data. The message broker translates messages between formal messaging protocols to make services communicate with each other. This is how completely independent services talk to each other even if they are built with a completely different technology stack.

Message brokers do validation, storing, routing, and delivering of messages to the appropriate services. They are basically intermediaries between different services to enable them to communicate with each other without knowing the other services at all. Senders issue messages to the message broker without knowing where the receivers are, whether or not they are active, or how many of them there are. This

enables us to create highly decoupled services within systems.

The reliable message storage and guaranteed delivery of messages are the key aspects of message brokers. Message brokers depend on a component called message queue that stores the messages and sends messages to the consuming applications for reliable messaging and guaranteed delivery. The message queue makes sure that messages are stored in the exact order in which they were transmitted and remain in the queue until the consumer service consumes the message.

Message brokers make asynchronous messaging between services possible. Message brokers prevent the loss of data and enable services to function properly even in the connectivity and latency issues. The main benefit of asynchronous messaging is that it guarantees messages will be delivered once and only once and in the correct order to the services.

2.3.2.1 Point-to-Point Messaging Pattern

This pattern is utilized in message queues with a one-to-one relationship between the message's sender and receiver. Each message in the queue is sent to only one recipient and is consumed exactly once. When a message needs to be delivered exactly once, the point-to-point messaging pattern is the right call. Payroll and financial transaction processes are good scenarios to use this pattern. In these scenarios, both senders and receivers need a guarantee that each payment will be sent once and only once.

2.3.2.2 Publish/Subscribe Messaging Pattern

This pattern is often referred to as "pub/sub" [16, 17]. In this pattern, the producer publishes messages to a topic in the message broker and subscribers or consumers subscribe to topics from which they want to receive messages. A topic in the message broker context is a means to identify published messages and send them to correct subscribers. When a producer publishes a message on a topic, all of the consumers of that topic receive the same message from the message broker. This is similar to broadcast-style message distribution, there is a one-to-many relationship between

producers and consumers. An example scenario for this pattern could be an airline system. If an airline updates a landing time or delay status of a flight, multiple other services could make use of this information, the operators of visual displays notifying the public, ground crews performing aircraft maintenance and refueling, and baggage handlers, flight attendants, and pilots preparing for the plane's next trip. A pub/sub messaging pattern [16, 17] is appropriate for such use cases.

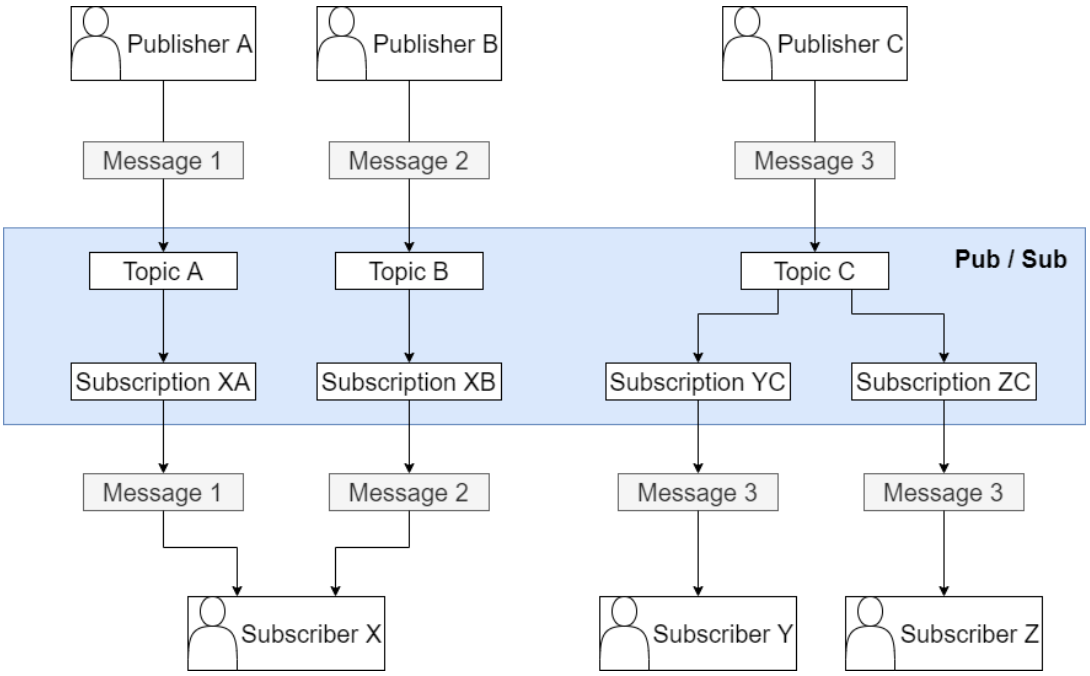

Figure 2.4: Publish/Subscribe System Example (adapted from [17])

Figure 2.4 shows an example usage of publish/subscribe architecture. There are a couple of things happening in this figure:

- There are three publishers, Publisher A, Publisher B, and Publisher C.
- There are three subscribers, Subscriber X, Subscriber Y, and Subscriber Z.
- There are three topics in the message broker, Topic A, Topic B, and Topic C.
- Subscriber X is subscribed to two topics, Topic A and Topic B, and that is why there are Subscription XA and Subscription XB in the message broker.
- Subscriber Y is subscribed to Topic C, that is why there is Subscription YC in the message broker.

- Subscriber Z is also subscribed to Topic C, that is why there is Subscription ZC in the message broker.

- When Publisher A publishes a message to Topic A, Subscriber X will receive this message via Subscription XA

- When Publisher B publishes a message to Topic B, Subscriber X will receive this message via Subscription XB

- When Publisher C publishes a message to Topic C, Subscriber Y and Subscriber Z will both receive this message via Subscription YC and Subscription ZC respectively.

2.4 Interoperability

Interoperability [18, 19, 20, 21, 22, 23, 24] can be described as the ability of two or more systems, components or objects to communicate in a way so that they share data and use information. Systems that are interoperable can exchange information in real-time, without needing support from IT or without behind-the-scenes coding.

Interoperability of systems can be as simple as a customer relationship management (CRM) system that provides deep integrations with automation software to create a flow of information between sales and marketing, or it can be a complex interoperability problem like integrations of multiple different systems, which can have different purposes, different architectures, and different technology stacks, to create a bigger system.

There are mainly three types of interoperability:

- **Technical Interoperability.** This is concerned with hardware, software components. It makes possibilities for machine-to-machine communication.

- **The Syntactical Interoperability.** This deals with the data formats. When data is transferred from one system to another system, the data should be well-defined with schemes and encoding like JSON, XML [25].

- **The Semantic Interoperability.** This deals with the understanding of data rather than the content of data. Semantic interoperability aims to make systems communicate with each other even if they use different data structures, it aims to make data interpreted the same way in both the sender and receiver systems.

2.4.1 Importance of Interoperability

Nearly all institutions depend on multiple software systems to operate, for example in all medium or big size companies there are divisions like human resources, marketing, finance, operations management, and IT, and all of these divisions heavily use some software for their job, but there are lots of cases these divisions work together, so the software applications they use should also be able to work together. So all these software applications used by different divisions need to interoperate with each other to make everything smooth for the employees. By interoperating all these software applications, we can derive future plans, business goals, etc. for the company. The importance of interoperability [26] can be seen in situations like this.

CHAPTER 3

PROPOSED TOOL

We created a tool that will handle interoperability problem among event-driven microservices systems using different message brokers [27]. In this chapter, the tool we created for the interoperability among different event-driven microservices systems is introduced.

3.1 The Problem

Messaging between event-driven microservices in a system is usually handled with message brokers like RabbitMQ [28], Kafka [29], ZeroMQ [30]. Since each message broker [6] has its best use case scenarios, different systems choose to use different message brokers according to their needs.

The problem arises when we need to combine multiple systems into a bigger system or when we want some microservices in different systems to communicate for a common goal. So our problem is the heterogeneity of message brokers in different event-driven microservices systems.

Figure 3.1 shows two different but very simple microservices systems. The problem in this diagram is exactly what we tried to solve but a very simple version of the problem.

- There are two Microservice systems, named Microservice System 1 and Microservice System 2
- Each microservice systems have two microservices, named Microservice 1 and

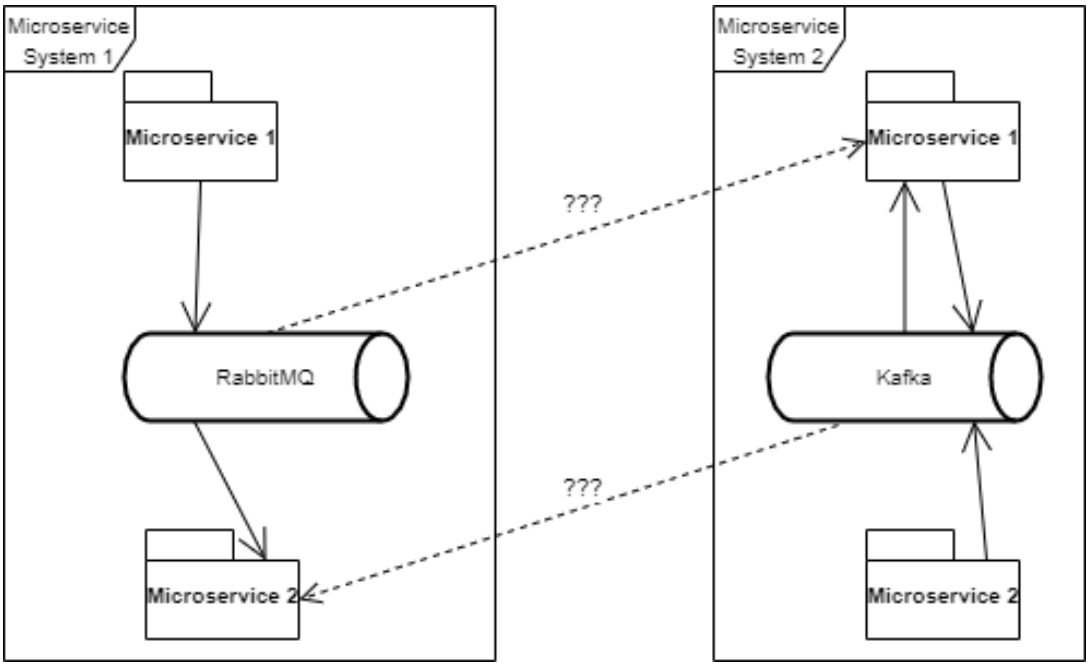

Figure 3.1: Problem Overview

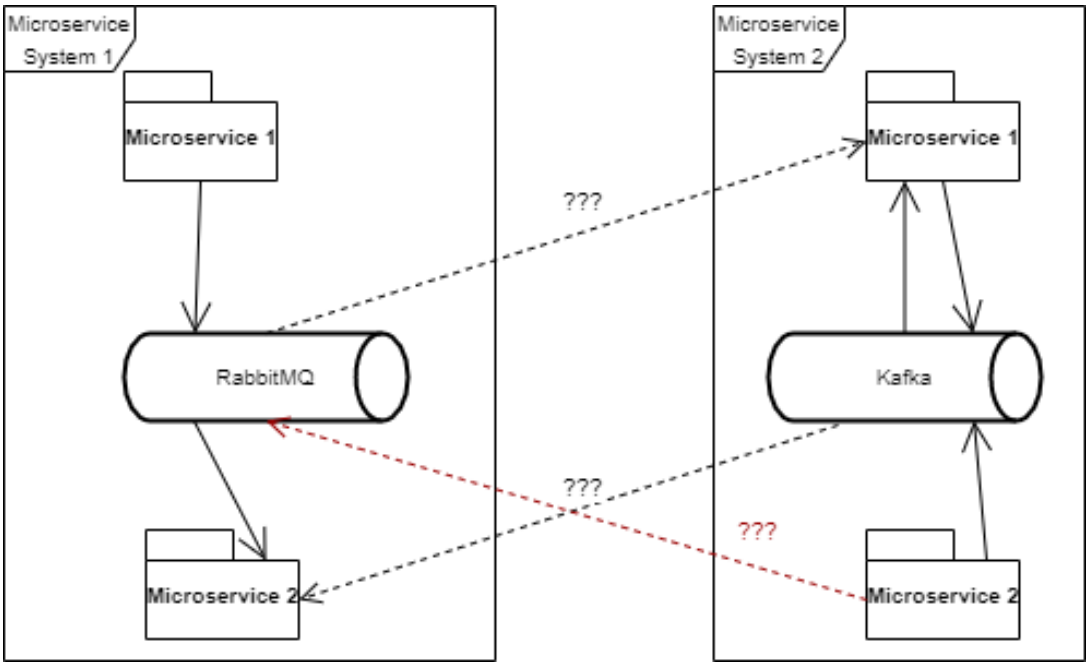

Figure 3.2: Extended Problem Overview

Microservice 2

- In Microservice System 1, the microservices communicate through RabbitMQ
- In Microservice System 2, the microservices communicate through Kafka
- How can Microservice 1 in System 2 get messages from System 1?
- How can Microservice 2 in System 1 get messages from System 2?

For Figure 3.1 solution may seem easy. Microservice 1 in System 2 can just connect to RabbitMQ of System 1 to get messages, and Microservice 2 in System 1 can just connect to Kafka of System 2 to get messages. However, what if Microservice 2 in System 2 also wants to get messages from System 1? Should it also connect to RabbitMQ of System 1 to get messages as depicted in Figure 3.2? This is not a feasible solution to handle messaging between different systems, because every microservice which wants to get messages from other systems has to connect to the message broker of that system to get messages. This simple way of solving this problem increases the complexity of microservices that wants to get messages from other systems because these microservices have to implement new code to handle subscribing to a completely different message broker. Each microservice in a system should only know about the message broker of its native system. Otherwise, as the number of systems to interoperate increases the number of message brokers each microservice has to handle will increase. It means developers of these microservices will have to implement new code when there is another system to communicate. Adding new features to a perfectly running service is always a problem, because developers of that service may change, or the developers may not be very good with the new message broker. That is why solving interoperability problem in the context of each microservices' implementation is not a feasible solution and it is very error-prone especially when we need to interoperate complex systems with different number of microservices and different message brokers such as depicted in Figure 3.3.

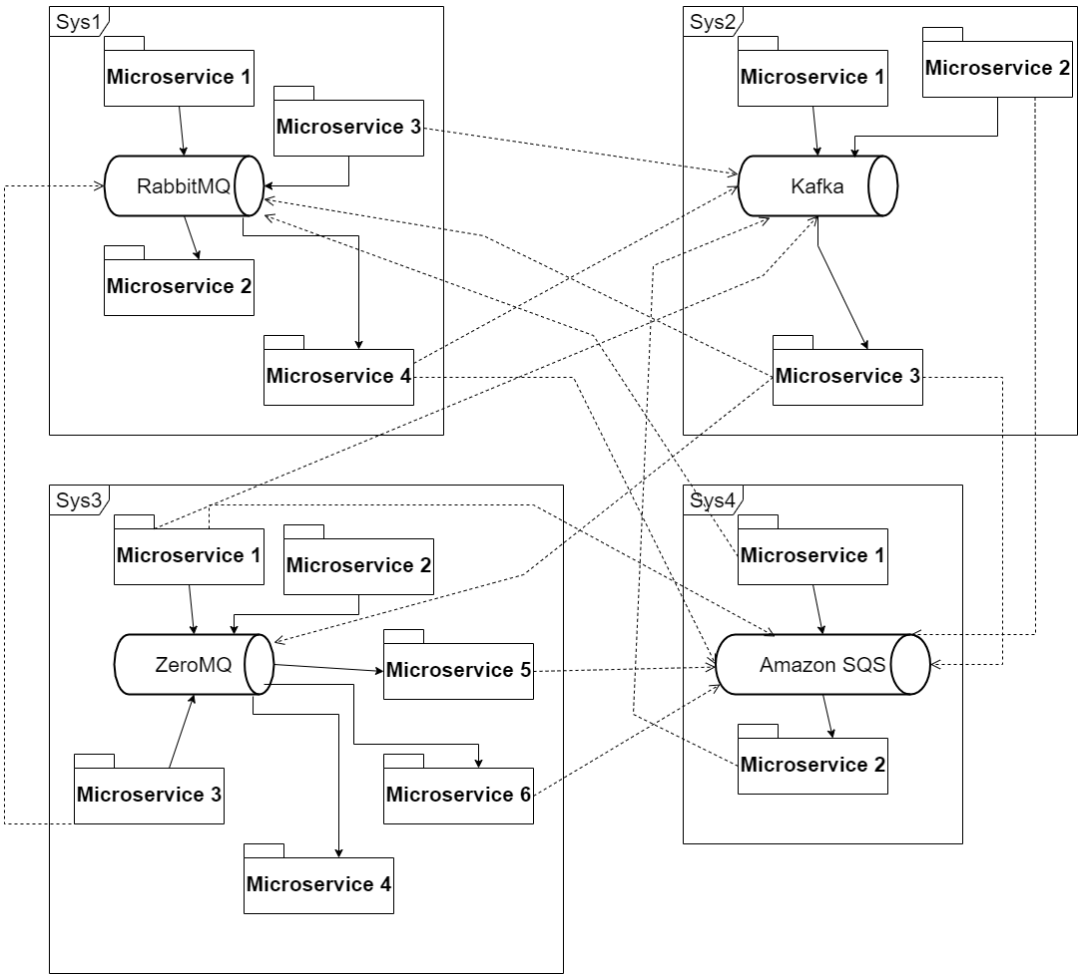

Figure 3.3: Complex Version of the Problem

3.2 The Solution

Our proposed tool aims to provide interoperability among various systems which use different message broker technologies for their publish/subscribe operations [27]. The developers of microservices have to make no change to their microservice to get messages from other systems, so our tool does not require them to make changes in the existing microservices.

Figure 3.4 shows a simple overview of making the systems in Figure 3.1 interoperable.

The proposed tool lets us forward messages from one system to one or more other systems. It does not forward all messages from each system to each system, because

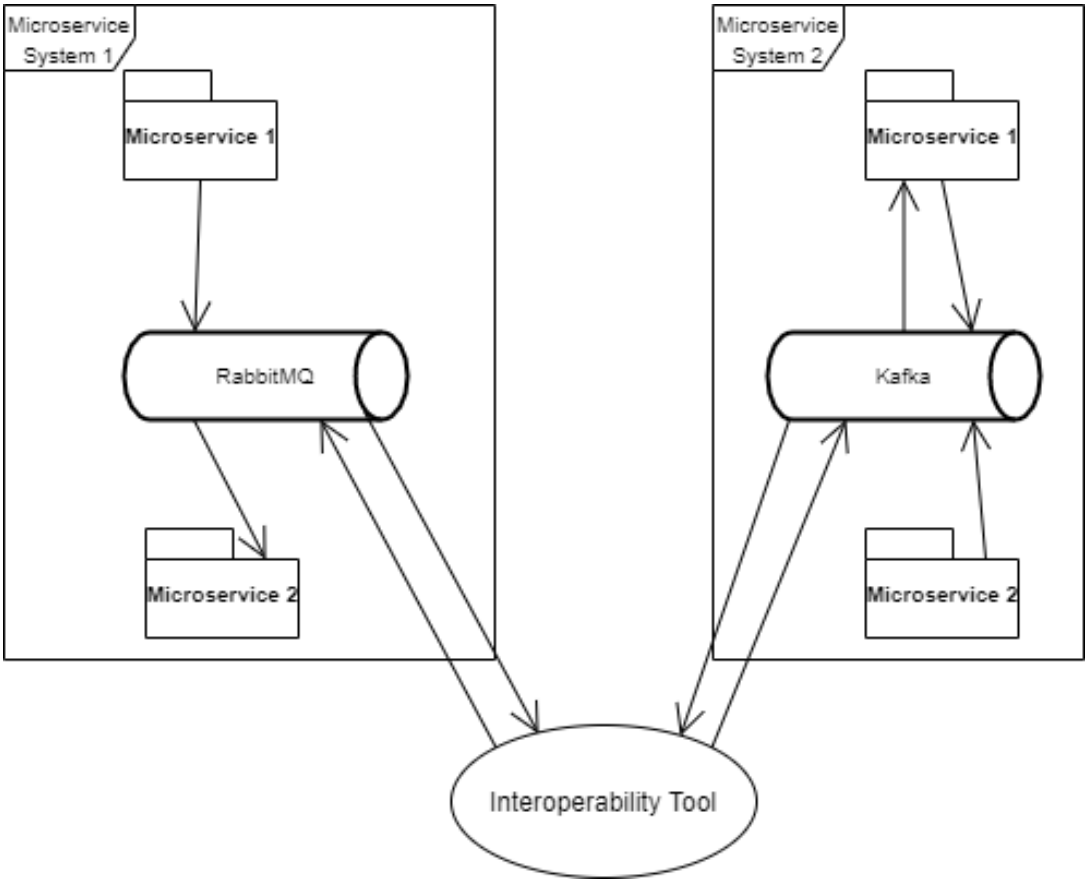

Figure 3.4: Tool in Action

doing so would increase network load among systems, and forwarding unwanted messages to a system where no microservice is interested in that message is an unnecessary task.

How does the tool know which messages to forward among systems? The tool is configurable at run-time, and the user of the tool can select what kind of messages needs forwarding from one system to one or more systems. That is why network traffic would not be affected so much.

3.3 The Proposed Tool in Detail

The proposed tool is aimed to provide interoperability among two or more event-driven microservice systems which use different middlewares for their publish/subscribe [16, 17, 27, 31] operations. The tool consists of three main parts as depicted in

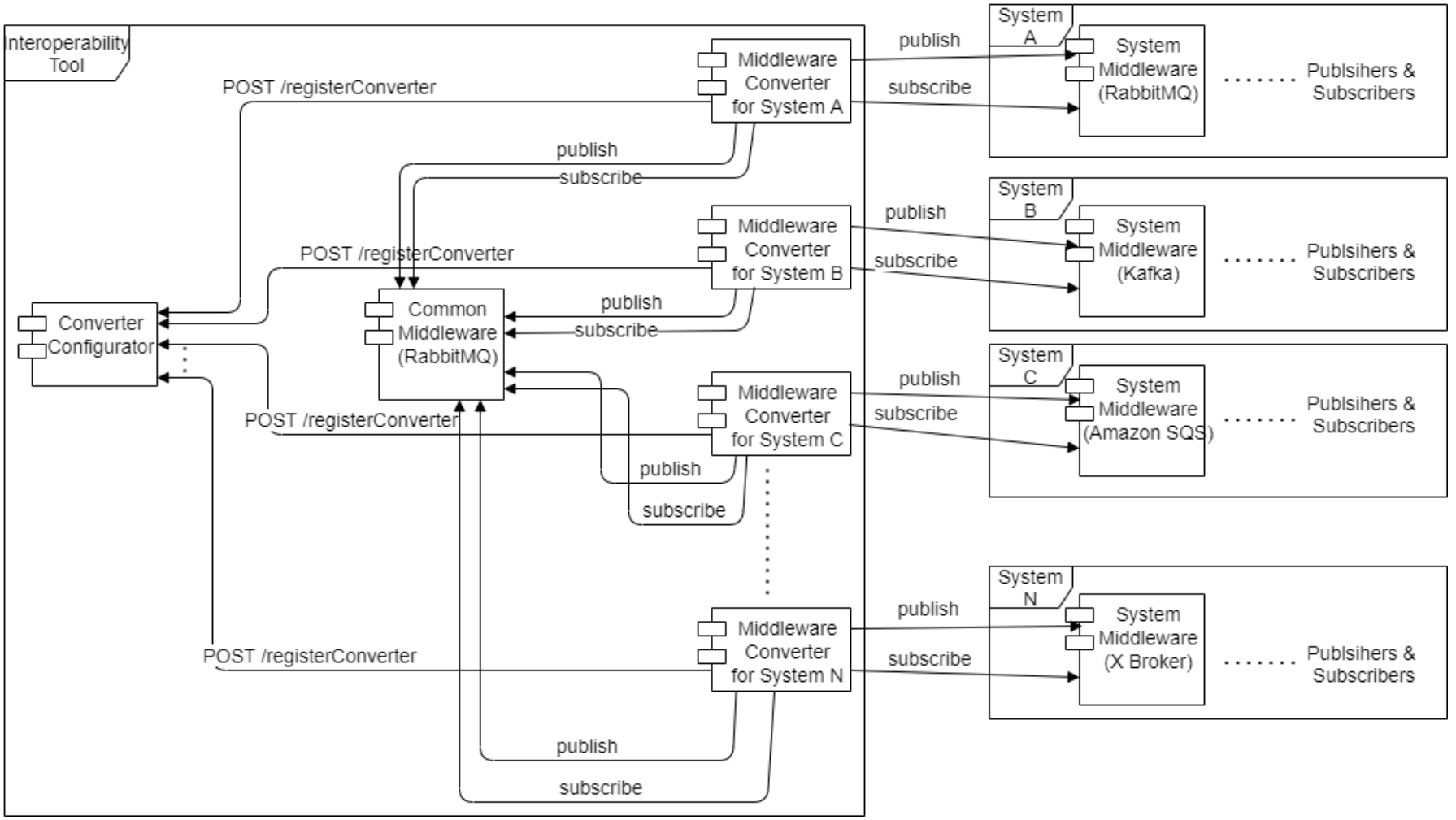

Figure 3.5: Tool in Detail

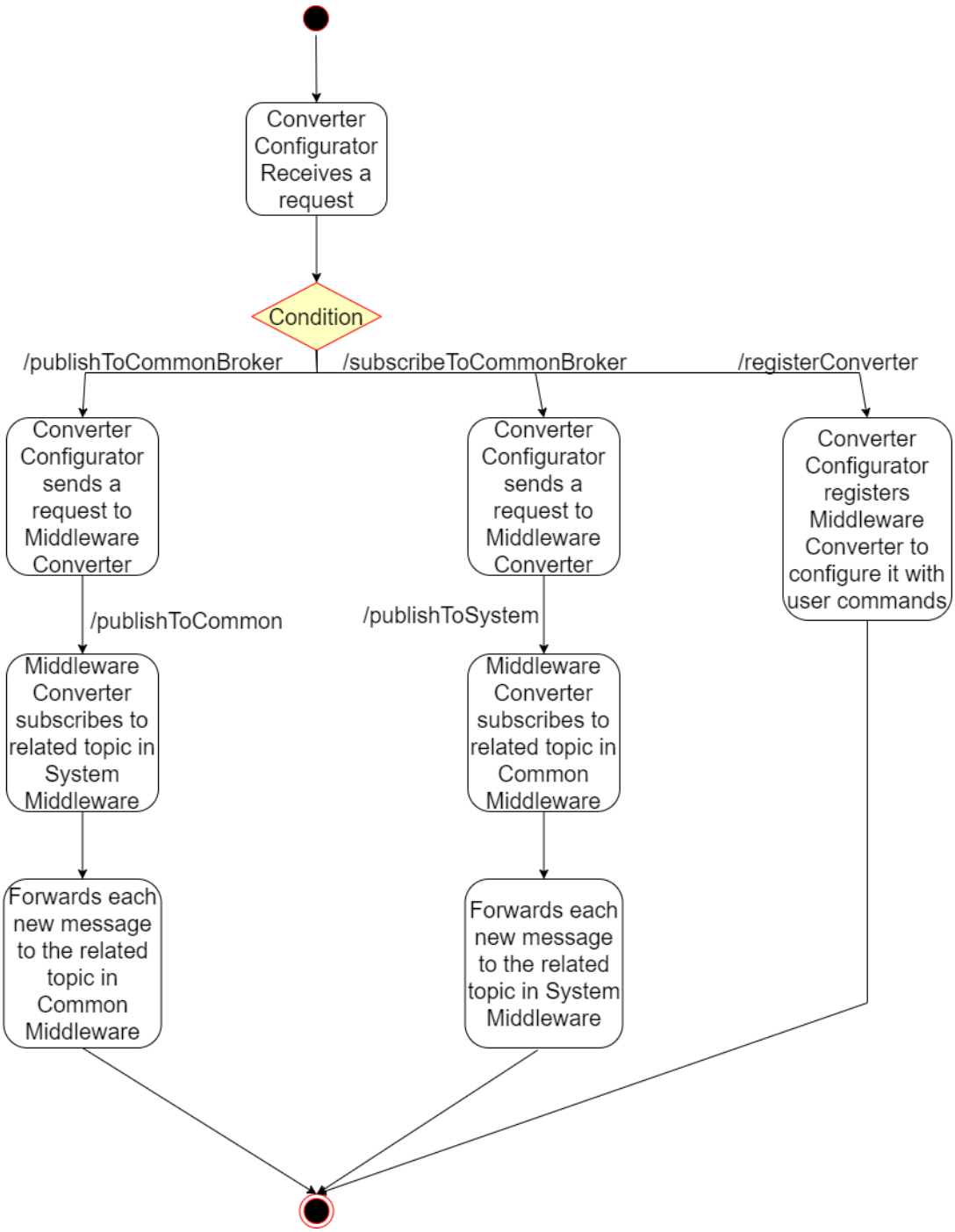

Figure 3.6: Activity Diagram of the Tool

Figure 3.5. These parts are

1. Converter Configurator
2. Middleware Converter
3. Common Middleware

Figure 3.5 represents the architecture of the proposed interoperability tool and the activity diagram in Figure 3.6 presents the flow of control in our interoperability tool. While Converter Configurator and Common Middleware components are single instances in the tool, there are as many Middleware Converter components as the number of target systems to interoperate. This architecture lets us distribute the forwarding logic between Middleware Converters and the Common Middleware component of the tool.

3.3.1 Converter Configurator

There is only one instance of the Converter Configurator component for the whole interoperability tool. The main responsibility of the Converter Configurator is orchestrating the Middleware Converter components so that they can forward messages among different systems. Converter Configurator serves a REST API [32, 33] through which users of the interoperability tool can configure the interoperability tool. User configuration lets the interoperability tool know the messages in specific topics in different systems should be forwarded to which topics in which systems. There are three endpoints in the API this component serves, these endpoints are:

- **/registerConverter**
- **/publishToCommonBroker**
- **/subscribeToCommonBroker**

3.3.1.1 registerConverter

This endpoint waits for POST requests. This endpoint is called by Middleware Converter components to let the Converter Configurator component hold necessary information about each Middleware Converter component. The request body should contain systemName, converterIP, converterPort, systemBrokerIP, systemBrokerPort parameters. The Converter Configurator component holds a map that has systemNames as key, and other parameters about Middleware Converters as value, so the Converter Configurator component does not allow two different Middleware Components to call this endpoint with the same systemName. converterIP and converterPort is the IP and port information of the REST API served by the Middleware Converter calling this endpoint. These IP and port information are necessary for the Converter Configurator because Converter Configurator sends requests to Middleware Converter components about which topics in their system broker they should subscribe and forward to Common Middleware, and which topics in Common Middleware they should subscribe and forward messages to which topics in their system broker. systemBrokerIP and systemBrokerPort are used to make sure no more than one Middleware Converter is using the same system middleware to get messages and to send messages. This endpoint returns "true" as a response if Middleware Converter can be added to the Converter Configurator to be configured with the requests of the user of the interoperability tool.

3.3.1.2 publishToCommonBroker

This endpoint also waits for POST requests. This endpoint is called by the users of the interoperability tool to configure one of the Middleware Converters to subscribe to one topic in its message broker of the system to get messages of that topic and publish those messages to a topic in the Common Middleware (message broker of the interoperability tool). The POST requests sent to this endpoint should contain systemName, systemBrokerTopic, and commonBrokerTopic parameters in the request body. When Converter Configurator receives requests from this endpoint, it first gets systemName variable from the request body and decides which Middleware Converter is responsible to handle this request, and then it sends another request to /publishToCommon

endpoint of the responsible Middleware Converter with the request body containing systemBrokerTopic, and commonBrokerTopic parameters. When the responsible Middleware Converter receives the request with those parameters, it subscribes to a topic named systemBrokerTopic in the request body and gets every message from that topic. Every new message in that topic of the message broker of the system is received by the Middleware Converter and then published to a topic named commonBrokerTopic in the Common Middleware. At the end of a request, which contains systemName, systemBrokerTopic, and commonBrokerTopic parameters in the request body, made to publishToCommonBroker endpoint. The messages in the topic named systemBrokerTopic in the message broker of the system, whose name is systemName, is forwarded to a topic named commonBrokerTopic in the Common Middleware, which is the message broker of our tool.

3.3.1.3 subscribeToCommonBroker

This endpoint also waits for POST requests. This endpoint is called by the users of the interoperability tool. It configures one of the Middleware Converters to subscribe to a topic in the Common Broker to get messages of that topic, and also to publish those messages to a topic in its system's message broker to let the system components get those messages from the system broker. The POST requests sent to this endpoint should contain systemName, commonBrokerTopic, and systemBrokerTopic parameters in the request body. When Converter Configurator receives requests from this endpoint, it first gets systemName variable from the request body and decides which Middleware Converter is responsible to handle this request. Then it sends another request to /publishToSystem endpoint of the responsible Middleware Converter with the request body containing commonBrokerTopic, and systemBrokerTopic parameters. When the responsible Middleware Converter receives the request with those parameters, it subscribes to a topic named commonBrokerTopic in the Common Middleware, and gets every message from that topic. Every new message in that topic of the Common Middleware is received by the Middleware Converter and then published to a topic named systemBrokerTopic in the message broker of the system of the Middleware Converter component. At the end of a request, which contains systemName, commonBrokerTopic, and systemBrokerTopic parameters in

the request body, made to subscribeToCommonBroker endpoint. The messages in the topic named commonBrokerTopic in the Common Middleware is forwarded to a topic named systemBrokerTopic in the message broker of the system whose name is systemName.

3.3.2 Middleware Converter

Middleware converters are the components that do most of the forwarding between microservice systems. They are components directly in contact with the systems we are trying to make interoperable. They are creating the connection to message brokers of the systems so that they can subscribe to any topic in those systems. They can also publish messages to the message broker of those systems. Each Middleware Converter is responsible for exactly one system; it connects to only the message broker of one system and makes it interoperable with the others.

The first thing Middleware Converter does when it starts running is that it sends a request to /registerConverter endpoint of the Converter Configurator to let it know, it is up and ready to forward messages between the message broker of our tool, Common Middleware, and the message broker of the system it is connected.

The main responsibility of a Middleware Converter is to create a connection to the message broker of a system and message broker of the interoperability tool, Common Middleware, and also to store a mapping about which topic in the system's message broker corresponds to which topic in the interoperability tool's message broker and the other way around. The Middleware Converter works bi-directional consequently it can forward messages from topics of system's message broker to topics of interoperability tool's message broker, and the other way around.

The Middleware Converter serves a REST API [32, 33] through which Converter Configurator can configure it. There are two endpoints in the API, which are:

- **/publishToCommon**
- **/publishToSystem**

3.3.2.1 publishToCommon

This endpoint waits for POST requests. This endpoint is called by the Converter Configurator component to make the Middleware Converter, receiving this request, subscribe to a topic in the message broker of the system it is responsible, and forward received messages from that topic to a topic, possibly with a different name, in the message broker of interoperability tool, known as Common Middleware. The POST requests sent to this endpoint should contain systemBrokerTopic and commonBrokerTopic parameters in the request body. The Middleware Converter, which receives the POST request with the correct parameters, subscribes to the topic named in the systemBrokerTopic parameter in the message broker of the system and publishes the new messages to the topic named in the commonBrokerTopic parameter in the Common Middleware, that is the interoperability tool's message broker.

3.3.2.2 publishToSystem

This endpoint also waits for POST requests. The requests made to this endpoint basically do the same as the previous endpoint but the other way around. This endpoint is also called by the Converter Configurator component to make the Middleware Converter, receiving this request, subscribe to a topic in the message broker of interoperability tool, and forward received messages from that topic to a topic, possibly with a different name, in the message broker of the system this Middleware Converter component is responsible for. The POST requests sent to this endpoint should contain commonBrokerTopic, and systemBrokerTopic in the request body. The Middleware Converter, which receives the POST request with the correct parameters, subscribes to a topic named commonBrokerTopic in the Common Middleware and publishes the new messages to the topic named in the systemBrokerTopic in the message broker of the system.

3.3.3 Common Middleware

The Common Middleware component is the component that handles the message forwarding among Middleware Converters. The architecture of the interoperability tool

is also event-driven [4, 5, 22, 31] microservice architecture like the systems we are trying to make interoperable. The Middleware Converters are just microservices that publish messages to topics and subscribe to topics to get new messages, that is why we thought it is best to use a message broker to handle the messaging among Middleware Converters. Since the forwarding of messages among Middleware Converters is such a complicated task and there is already a good solution for this task, we choose to use the ready solution, which is message brokers [6, 34]. We choose to use RabbitMQ [28] as our message broker for the interoperability tool since it was created exactly to handle publish/subscribe messaging, and it is currently one of the best in the market.

3.4 Example

Although the detailed explanation of what each component does may seem a little long, the overall idea is easy to understand that is why we want to give a small example of how this tool would be used in a real-life scenario.

For this scenario, we have

- Two event-driven microservice systems, System A and System B.
- System A has two microservices, Microservice A-1 and Microservice A-2.
- System B has two microservices, Microservice B-1 and Microservice B-2.
- System A uses Kafka for communication between its services
- System B uses RabbitMQ for communication between its services
- Microservice A-1 publishes to a topic named 'topic A', and Microservice A-2 subscribes to that topic to get messages.
- Microservice B-1 publishes to a topic named 'topic B', and Microservice B-2 subscribes to that topic to get messages.

In this scenario, Microservice A-1 sends messages to 'topic A' in Kafka, and Microservice A-2 receives those messages from 'topic A' to do its job based on the

received messages. Microservice B-1 sends messages to 'topic B' in RabbitMQ, and Microservice B-2 receives those messages from 'topic B' to do its job based on the received messages. In this example our goal is to forward messages from 'topic A' in System A to 'topic B' in System B because we want Microservice B-2 to process those messages which are originally generated in System A. To make this happen, our tool is a perfect solution.

In Figure 3.7, we tried to show the steps happening inside the interoperability tool. On the right of the Figure 3.7, there are the two systems, System A, and System B. The Microservice A-1 in System A is publishing messages to a topic named "topic A" in System A's message broker, Kafka. And the Microservice A-2 subscribes to the same topic so that it can get messages published by Microservice A-1. The Microservice B-1 in System B is publishing messages to a topic named "topic B" in System B's message broker, RabbitMQ. And the Microservice B-2 subscribes to the same topic so that it can get messages published by Microservice B-1. These two completely different systems, which does not know anything about the other system is perfectly functioning, but we need to make Microservice B-2 also get messages from the topic named "topic A" in the message broker of System A. However we can not change the code of Microservice B-2 because it is running perfectly for so long and the all of the developers of it is gone now, so it is super risky to change the code of Microservice B-2 to implement a new feature that will make it connect a completely different message broker to subscribe to a topic, while at the same it can also use RabbitMQ as message broker. That is a perfectly good problem for our interoperability tool. Figure 3.7 and 3.8 show how we make those two systems interoperable with our proposed tool and they show execution of the proposed tool.

Following steps explain how the developed tool achieves interoperability between two systems:

1. Common Middleware component, message broker of our proposed tool, starts running

2. Converter Configurator component starts running

3. Middleware Converters, Middleware Converter for System A and Middleware

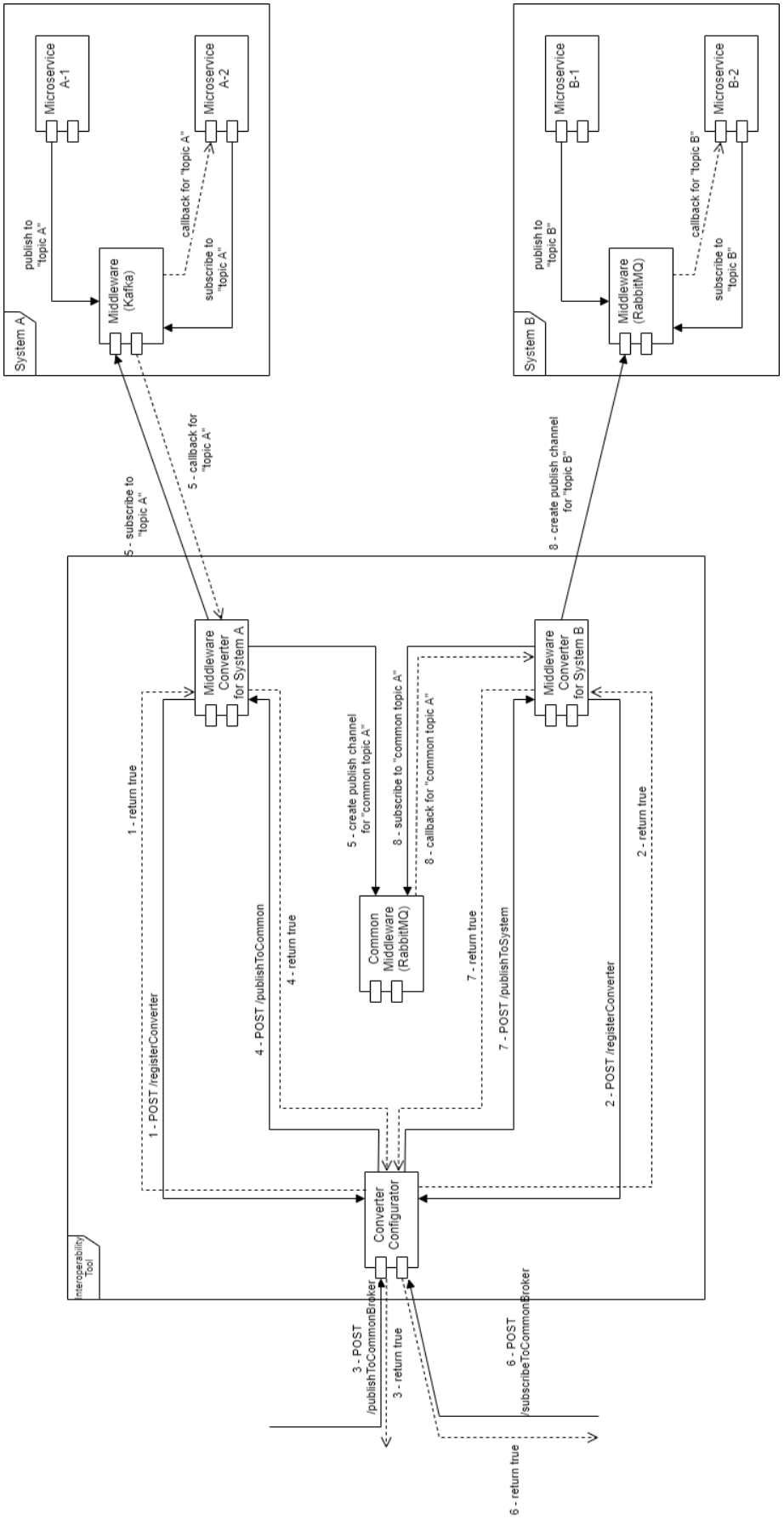

Figure 3.7: Component Diagram of Given Example

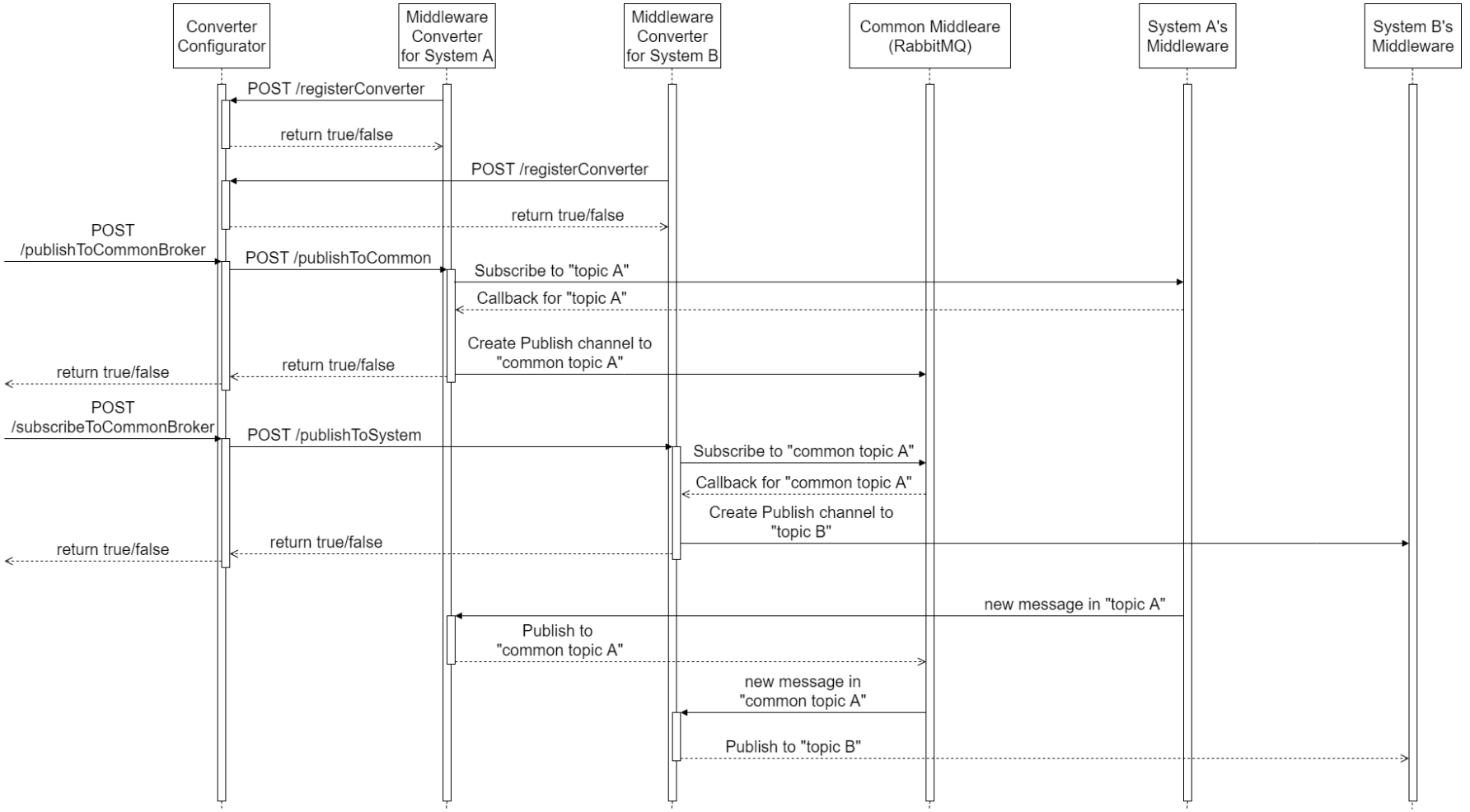

Figure 3.8: Sequence Diagram of Given Example

Converter for System B, start running.

4. Middleware Converters for System A and System B sends a POST request to the /registerConverter endpoint of the API exposed by Converter Configurator with request body that is shown in the following listing:

```
{
    "systemName": "System A or System B",
    "converterIP": "The IP of the machine or container
                    Middleware Converter is running",
    "converterPort": "The port from which Middleware
                      Converter exposed REST API",
    "systemBrokerIP": "IP of the message broker of
                       the system",
    "systemBrokerPort": "Port from which the message
                         broker of the system accepts
                         connections"

}
```

5. The user of the interoperability tool first wants to forward messages from a topic named "topic A" in the message broker of System A to a topic, with a name he/she will give with the request, in the Common Middleware. The user of the interoperability tool will send a request to /publishToCommonBroker endpoint of the API exposed by Converter Configurator component with the request body shown below:

```
{
    "systemName": "System A",
    "systemBrokerTopic": "topic A",
    "commonBrokerTopic": "common topic A"

}
```

6. The Converter Configurator firstly uses systemName parameter to find which Middleware Converter is responsible for handling this request. After finding the responsible Middleware Converter, in this example it is Middleware Converter

for System A, Converter Configurator sends a request to /publishToCommon endpoint of the API exposed by the Middleware Converter with the request body shown below:

```
{
    "systemBrokerTopic": "topic A",
    "commonBrokerTopic": "common topic A"
}
```

7. After receiving this request, the Middleware Converter for System A subscribes to a topic named "topic A" in the message broker of System A. And whenever it receives a message from this topic, it immediately publishes the received message to the topic named "common topic A" in Common Middleware, message broker of the interoperability tool.

8. At this point, the messages in the topic named "topic A" in the message broker of System A are forwarded to a topic named "common topic A" in Common Middleware. Now we need to forward messages from topic named "common topic A" to topic named "topic B" in the message broker of System B.

9. The user of the interoperability tool now wants to forward messages from a topic named "common topic A" in the message broker of the interoperability tool to a topic, "topic B" for this example, in the message broker of System B. The user of the interoperability tool will send a request to /subscribeToCommonBroker endpoint of the API exposed by Converter Configurator component with the request body shown below:

```
{
    "systemName": "System B"
    "commonBrokerTopic": "common topic A"
    "systemBrokerTopic": "topic B",
}
```

10. The Converter Configurator again firstly uses systemName parameter to find which Middleware Converter is responsible for handling this request. After

finding the responsible Middleware Converter, in this example it is Middleware Converter for System B, Converter Configurator sends a request to /publish-ToSystem endpoint of the API exposed by the Middleware Converter with the request body shown below:

```
{
    "commonBrokerTopic": "common topic A"
    "systemBrokerTopic": "topic B",

}
```

11. After receiving this request, the Middleware Converter for System B subscribes to a topic named "common topic A" in the Common Middleware. And whenever it receives a message from this topic, it immediately publishes the received message to the topic named "topic B" in the message broker of System B.

12. Now, everything is complete for this example. The first request sent by the user of the interoperability tool made the Middleware Converter for System A forward each message in the topic named "topic A" in the message broker of System A to a topic named "common topic A" in the message broker of the interoperability tool, Common Middleware. The second request sent by the user of the interoperability tool made the Middleware Converter for System B forward each message in the topic "common topic A" in the Common Middleware to "topic B" in the message broker of System B. So at the end, our interoperability tool is forwarding each message in the topic named "topic A" in the message broker of System A to topic named "topic B" in the message broker of System B. Now Microservice B-2 can use messages that actually originated at System A to do its job.

3.5 How to Extend

It is obvious the brain of the interoperability tool is the Converter Configurator, it sends requests to Middleware Converters to make other systems interoperable. And the Common Middleware is the way how Middleware Converters forward messages

between different systems. So we designed the tool so that it can be extended by adding more Middleware Converters.

Currently, we have implemented two Middleware Converters. One of them can work with systems that are using Kafka as their message broker, and the other one can work with systems that are using RabbitMQ as their message broker. To extend the interoperability tool to make it also work with systems which are using a different message broker, a new Middleware Converter needs to be implemented that can work with the new message broker of interest.

3.5.1 How to Implement a New Middleware Converter

Currently, we have implemented two Middleware Converters, one can work with systems using Kafka [29] as its message broker and the other can work with systems using RabbitMQ [28]. However when someone needs a Middleware Converter for another message broker like ZeroMQ [30], a new Middleware Converter needs to be implemented that is going to forward messages between the new message broker and the message broker of the interoperability tool, which is RabbitMQ. The activity flow in Figure 3.9 presents the steps to follow while implementing a new Middleware Converter.

To set out the requirements for a new Middleware Converter, the inner workings of a Middleware Converter can be laid out. Middleware Converters serve as a REST API [32, 33] through which they accept requests from the Converter Configurator. The API has two endpoints /publishToCommon and /publishToSystem. One of the first things, when it starts running, is sending a POST request to /registerConverter endpoint of the API served by Converter Configurator with the request body containing systemName, converterIP, converterPort, systemBrokerIP, and systemBrokerPort parameters. It would be beneficial to quote the parameters here:

- **systemName** is the name of the system which is going to be made interoperable with other systems by the Middleware Converter.
- **converterIP and converterPort** is the IP and port of the REST API served by the Middleware Converter.

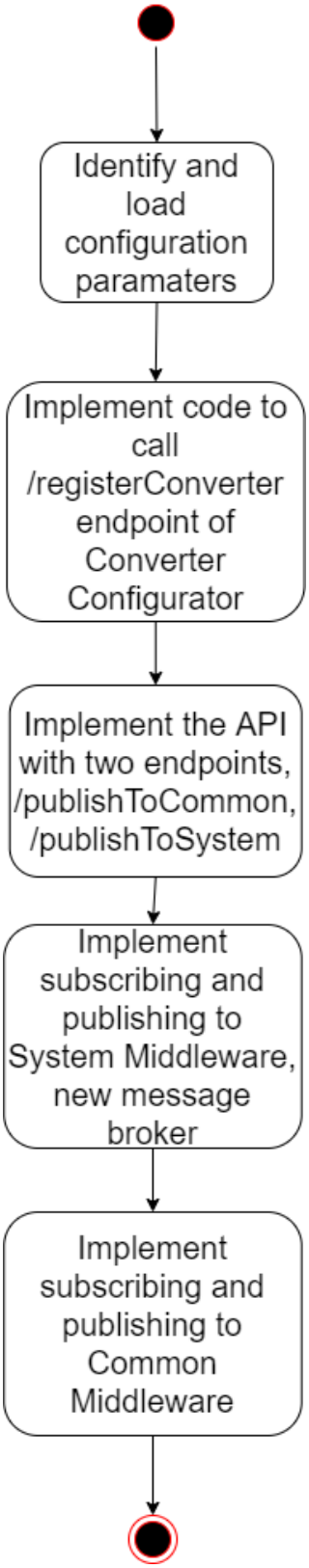

Figure 3.9: Activity Flow for Implementing A New Middleware Converter

- **systemBrokerIP and systemBrokerPort** is the IP and port of the message broker of the system Middleware Converter connects to subscribe and publish to topics.

The Middleware Converters we implemented for Kafka and RabbitMQ are configurable. A few items are made configurable: systemName, systemBrokerIP, systemBrokerPort, configuratorIP, configuratorPort, commonBrokerIP, and commonBrokerPort. The last two parameters, commonBrokerIP and commonBrokerPort, are IP and port information to connect to the message broker of the interoperability tool, Common Middleware. The configuratorIP and configuratorPort parameters are the IP and port information to send a request to Converter Configurator. The first three, systemName, systemBrokerIP and systemBrokerPort, is the configuration for Middleware Converter itself. With this information the Middleware Converter can send a request to Converter Configurator with the required parameters. Other than sending requests to Converter Configurator, it also needs to receive requests from Converter Configurator from the two endpoints, /publishToCommon and /publishToSystem. It also needs to implement a REST API for this, and accepts requests from the Converter Configurator. After getting a request from /publishToCommon endpoint with the request body containing systemBrokerTopic and commonBrokerTopic, the Middleware Converter should subscribe to the topic. The topic is named in the systemBrokerTopic parameter and in the new message broker. Each message is published to the topic named in the commonBrokerTopic parameter, in the message broker of the interoperability tool. After getting a request from /publishToSystem endpoint with the request body containing commonBrokerTopic and systemBrokerTopic, the Middleware Converter should subscribe to the topic named in the commonBrokerTopic parameter. That is in the message broker of the interoperability tool, and publish each new message to the topic named in the systemBrokerTopic parameter, in the new message broker of the system it is responsible.

CHAPTER 4

CASE STUDY - INTEROPERABILITY AMONG ACCESS CONTROL SYSTEMS

One of the best ways to test the applicability of a new application is actually using it. That is why we decided to test the interoperability tool with a real-life scenario. From this case study, we can understand the limits of our proposed tool better, and we can have some ideas about how to make it better. In this chapter, we are going to describe how we did our case study and we are going to show the results we got.

The main idea of the example scenario is to make different access control systems [35, 36] interoperable so that they can send data to each other. We created this example scenario to show the applicability of the proposed solution.

4.1 Access Control System

The access control system in the fields of physical security provides selective restriction of access to a place. Access control is based on the software application to solve the limitations of mechanical locks and keys. Instead of using mechanical keys, it let us use a wide range of credentials. The access control system grants access based on the credentials of the user. If a user is authorized to open a locked door and the user tries to open the door, the user can unlock it for a predetermined time, and this action is stored as a transaction. If a user is unauthorized to open a locked door and the user tries to open the door, the user cannot unlock the door and this action is also stored. The access control system also monitors the doors and starts an alarm if a door is forced to open or held open too long after being unlocked.

4.2 Our Scenario

In our case study scenario, we have a campus containing multiple buildings and there are a couple of different access control systems used in each building. We use access control systems to authorize users to enter rooms they are allowed to and monitor when a user enters a room, when a user exits a room, when a user tries to enter a room without authorization. However every building is not using the same access control system; each building may have different access control systems for security reasons. Access control of each building is chosen according to its security level. There is a security employee in each building whose job is to monitor the access control system and take action if necessary. However, some of the security employees in some buildings are also required to monitor other buildings than they are currently located. For example, security employees in Building A should monitor the movement of employees for both Building A and Building B, and security employees in Building B should monitor the movement of employees for both Building A and Building B. As stated earlier, the access control systems installed in Building A and Building B could be different, so we have to send each system activity between these buildings.

The access control system of Building A is publishing messages about the entrances to and exits from any toll gate to a topic named "track" in its message broker, which is RabbitMQ. Similarly, the access control system of Building B is publishing messages about the entrance to and exit from any toll gate to a topic named "movement" in its message broker, which is Kafka. Both access control systems are configurable to show the movements of employees. They can show movement data from various topics in their message broker, e.g. the access control system of Building A can show employee movement not only through the topic named "track" but also through other topics. This is also valid for the access control system of Building B.

For this case study, we have to forward messages from the topic named "track" in the message broker of the access control system of Building A to the message broker of the access control system of Building B with a topic name of our choice. We also have to forward messages from the topic named "movement" in the message broker of the access control system of Building B to the message broker of the access control system of Building A with a topic name of our choice. We are going to

forward messages from topic named "track" in RabbitMQ of Building A to a topic named "buildingA_movement" in Kafka of Building B, and we are going to forward messages from topic named "movement" in Kafka to a topic named "buildingB_track" in RabbitMQ of Building A. We are going to configure each access control system to show system activity data also from these topics: "buildingB_track" for Building A's system and "buildingA_movement" for Building B's system.

To summarize, we are required to forward messages published to some topics in both middlewares to each other. However, there are different access control systems that are using different message broker technologies, and in the future, there could be other message broker technologies to integrate together. For this kind of a problem, our proposed tool is a good solution because it is extensible to other message broker technologies that are not supported yet, and it is configurable for messages regarding their related topics and message brokers.

4.3 Test Setup

Our proposed interoperability solution is tried and tested with the following situation:

- We tried to integrate two access control systems located in Buildings A and B
- The message broker of the access control system in Building A is RabbitMQ, and
- The message broker of the access control system in Building B is Kafka
- The access control system of Building A is publishing system data to "track" topic in its message broker
- The access control system of Building B is publishing system data to "movement" topic in its message broker
- The access control system of Building A is configured to show system data to security employees from "track" and "buildingB_track" topics.
- The access control system of Building B is configured to show system data to security employees from "movement" and "buildingA_movement" topics.

- Our proposed interoperability solution is configured to forward messages:
  - From the topic named "track" in the message broker of Building A's system to the topic named "buildingA_movement" in the message broker of Building B's system
  - From the topic named "movement" in the message broker of Building B's system to the topic named "buildingB_track" in the message broker of Building A's system
- With this configuration, both systems can receive desired messages

In Figure 4.1, we tried to show what is happening inside the interoperability tool with the requests coming from the user, or admin, of the tool. The two access control systems of Building A and Building B are depicted in the figure, but the details of them are left out because the details are not related to what is happening in the interoperability tool. The requests sent among the components are shown with the numbers, which show the order of the requests. To better understand what is the order of requests, it is better to look at the Figure 4.2.

The sequence diagram in Figure 4.2 shows how the user configures the interoperability tool so that the desired communication is achieved. The sequence diagram shows how both Middleware Converters for Building A and Building B are configured and then how a message in the topic named "track" in the message broker of Building A's system is forwarded to a topic named "buildingA_movement" in the message broker of Building B's system, and how a message in the topic named "movement" in the message broker of Building B's system is forwarded to a topic named "buildingB_track" in the message broker of Building A's system.

Following steps explains how the interoperability tool makes two access control systems interoperable:

1. Common Middleware component, message broker of our proposed tool, starts running
2. Converter Configurator component starts running

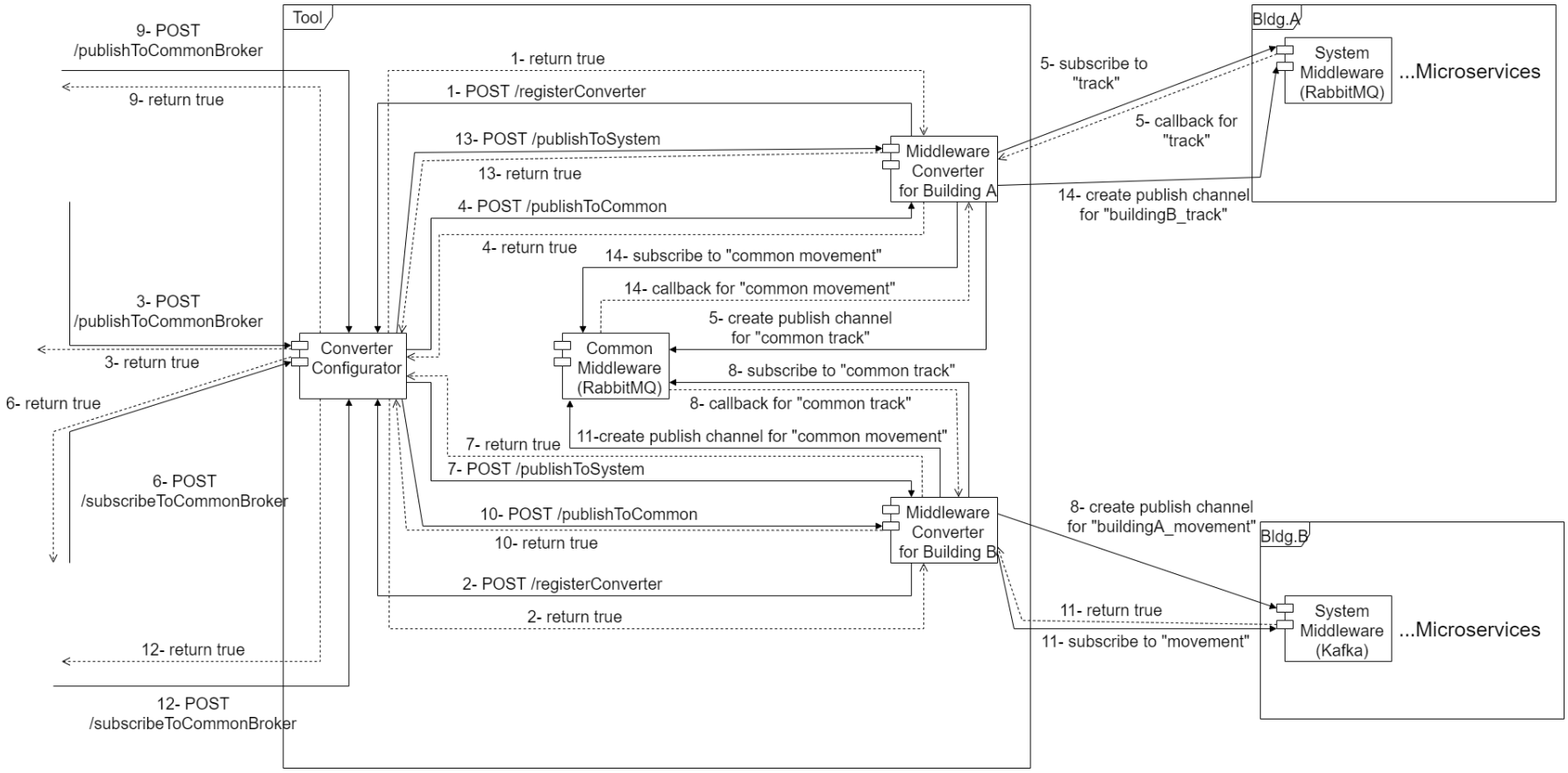

Figure 4.1: Component Diagram of Case Study

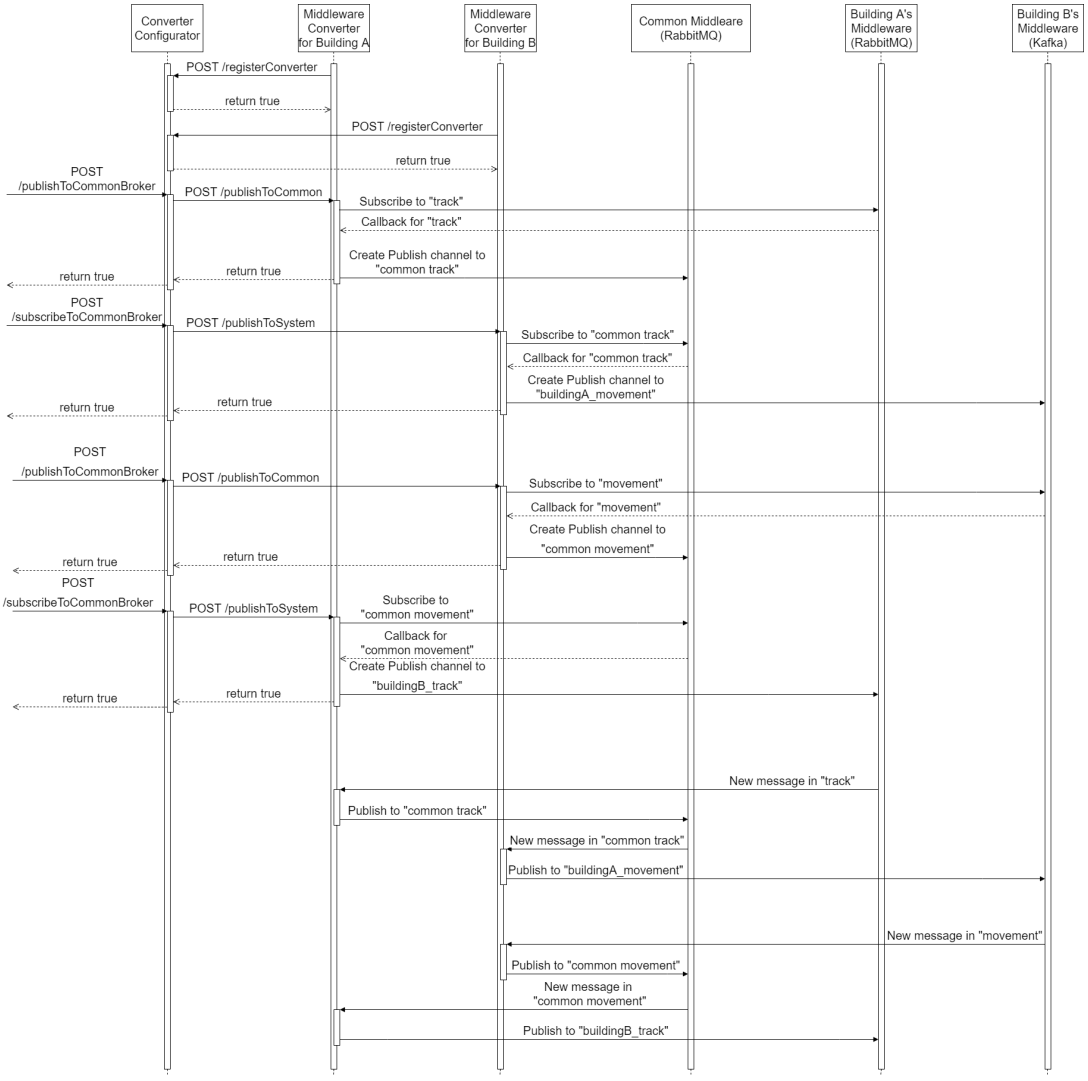

Figure 4.2: Sequence Diagram of Case Study

3. Middleware Converters, Middleware Converter for Building A and Middleware Converter for Building B, start running.

4. Middleware Converters for Building A and Building B sends a POST request to the /registerConverter endpoint of the API exposed by Converter Configurator with request body as shown below:

```
{
    "systemName": "Building A or Building B",
    "converterIP": "The IP of the machine or container
                    Middleware Converter is running",
    "converterPort": "The port from which Middleware
                      Converter exposed REST API",
    "systemBrokerIP": "IP of the message broker of
                       the system",
    "systemBrokerPort": "Port from which the message
                         broker of the system accepts
                         connections"
}
```

5. At this point, Converter Configurator is aware of the two systems. However, there are currently no messages forwarded between systems. Now, the admin of the interoperability tool needs to send 4 requests to configure the interoperability tool so that the desired forwarding is achieved. In the sequence diagram, the first two requests sent by the admin make the interoperability tool forward messages from the topic named "track" in the message broker of Building A's system to the topic named "buildingA_movement" in the message broker of Building B's system. The other two requests sent by the admin make the interoperability tool forward messages from the topic named "movement" in the message broker of Building B's system to the topic name "buildingB_track" in the message broker of Building A's system.

6. The admin, or user, of the interoperability tool, first sends a request to /publishToCommonBroker endpoint of the API exposed by the Converter Configurator to forward messages from the topic named "track" in the message broker of

Building A's system to the topic named "common track" in the message broker of the interoperability tool, Common Middleware. The request sent by the user has the following structure:

```
{
    "systemName": "Building A",
    "systemBrokerTopic": "track",
    "commonBrokerTopic": "common track"
}
```

7. After getting the above request from the user, the Converter Configurator firstly finds which Middleware Converter is responsible for handling forwarding of messages from a message broker of a system to the message broker of the interoperability tool, Common Middleware. Then, it sends a request to /publishToCommon endpoint exposed by the Middleware Converter with a request body as shown below:

```
{
    "systemBrokerTopic": "track",
    "commonBrokerTopic": "common track"
}
```

8. After receiving this request, the Middleware Converter for Building A subscribes to a topic named "track" in the message broker of Building A's system. It publishes the received messages from this topic to the topic named "common track" in Common Middleware.

9. Now, the messages in the topic named "track" in the message broker of Building A's system are forwarded to a topic named "common track" in the Common Middleware.

10. The admin of the interoperability tool now sends another request which will make the interoperability tool get messages from a topic named "common track" in Common Middleware and forward them to a topic named "buildingA_movement"

in the message broker of Building B's system. The request is sent to /subscribeToCommonBroker endpoint of the API exposed by the Converter Configurator component with the request body as shown below:

```
{
    "systemName": "Building B",
    "commonBrokerTopic": "common track",
    "systemBrokerTopic": "buildingA movement"
}
```

11. After getting the above request from the user, the Converter Configurator again firstly finds which Middleware Converter is responsible for handling forwarding of messages from the message broker of the interoperability tool to a message broker of a system. Then, it sends a request to /publishToSystem endpoint exposed by the Middleware Converter with a request body as shown below:

```
{
    "commonBrokerTopic": "common track",
    "systemBrokerTopic": "buildingA movement"
}
```

12. After receiving this request, the Middleware Converter for Building B subscribes to topic named "common track" in the Common Middleware. And it publishes the received messages from this topic to the topic named "buildingA_movement" in the message broker of Building B's system.

13. Now, messages in the topic named "track" in the message broker of Building A's system are forwarded to a topic named "buildingA_movement" in the message broker of Building B's system.

14. The admin, or user, of the interoperability tool, now sends a request to /publishToCommonBroker endpoint of the API exposed by the Converter Configurator to forward messages from the topic named "movement" in the message broker of Building B's system to the topic named "common movement" in the message broker of the interoperability tool, Common Middleware. The request sent by the user has the following structure:

```
{
    "systemName": "Building B",
    "systemBrokerTopic": "movement",
    "commonBrokerTopic": "common movement"
}
```

15. After the getting above request from the user, the Converter Configurator firstly finds which Middleware Converter is responsible for handling forwarding of messages from a message broker of a system to the message broker of the interoperability tool, Common Middleware. Then, it sends a request to /publishToCommon endpoint exposed by the Middleware Converter with a request body as shown below:

```
{
    "systemBrokerTopic": "movement",
    "commonBrokerTopic": "common movement"
}
```

16. After receiving this request, the Middleware Converter for Building B subscribes to a topic named "movement" in the message broker of Building B's system. It publishes the received messages from this topic to the topic named "common movement" in Common Middleware.

17. Now, the messages in the topic named "movement" in the message broker of Building B's system are forwarded to a topic named "common movement" in Common Middleware.

18. The admin of the interoperability tool now sends another request which will make the interoperability tool get messages from topic named "common movement" in Common Middleware and forward them to a topic named "buildingB_track" in the message broker of Building A's system. The request is sent to /subscribeToCommonBroker endpoint of the API exposed by the Converter Configurator component with the request body as shown below:

```
{
```

```
    "systemName": "Building A",
    "commonBrokerTopic": "common movement",
    "systemBrokerTopic": "buildingB track"

}
```

19. After getting the above request from the user, the Converter Configurator again firstly finds which Middleware Converter is responsible for handling forwarding of messages from the message broker of the interoperability tool to a message broker of a system. Then, it sends a request to /publishToSystem endpoint exposed by the Middleware Converter with a request body as shown below:

```
{

    "commonBrokerTopic": "common movement",
    "systemBrokerTopic": "buildingB track"

}
```

20. After receiving this request, the Middleware Converter for Building A subscribes to topic named "common movement" in the Common Middleware. And it publishes the received messages from this topic to the topic named "buildingB_track" in the message broker of Building A's system.

21. Now, messages in the topic named "movement" in the message broker of Building B's system are forwarded to the topic named "buildingB_track" in the message broker of Building A's system.

22. Everything is completed for testing. After this point, every message published in the topic named "track" in the message broker of Building A's system is forwarded to a topic named "buildingA_movement" in the message broker of Building B's system. Also every message published in topic named "movement" in the message broker of Building B's system is forwarded to a topic named "buildingB_track" in the message broker of Building A's system.

4.3.1 Simulation with Docker

We have used containerization[11, 12, 13, 14] technology excessively to simulate our case study and to log time delays for a message that take place during its reach from its publisher to a subscriber. Firstly, we have created Docker images for publisher and subscriber services in access control systems in Building A and Building B. Each application in each Docker image is designed to send some log information to a log service so that we can use these log data later. Secondly, we created a Docker image which will run the Converter Configurator application and manage Middleware Converters. And lastly, we created two Docker images containing each Middleware Converter applications, one working with Kafka as the message broker of the system, and the other one working with RabbitMQ as the message broker of the system. All of the `Dockerfiles` for creating these Docker images can be found in Appendix A.

We have Docker images which are responsible for simulating publisher and subscriber services in access control systems and the components of our interoperability tool. We can now create access control systems and the interoperability tool by combining these Docker images. We are required to create an environment where each component of this case study is working like it is in real-life. To achieve this, we have used Docker network and `docker-compose`. All of the docker-compose files to create the simulation environment can be found in Appendix B, and all of the code for this simulation can be found in [37].

4.4 Discussion and Test Results

The event-based microservice architecture of the proposed interoperability tool enables scalability [38]. The proposed interoperability tool can offer communication for numerous systems at the same time. In this case study, for the sake of simplicity, we conducted our tests with two systems: the access control systems [35, 36] of Building A and Building B.

We have tested our solution 5 times with an increasing number of published messages

per second from both Building A's and Building B's publishers. Since our interoperability tool forwards messages in both directions for this case study scenario, the number of messages forwarded in our tool is the sum of messages published in both systems.

Table 4.1: Average Time Delays Between Publishers And Subscribers in Milliseconds

		Published messages per second				
Source	Dest	40	100	200	400	500
A Pub	**A Sub**	**1.42**	**1.30**	**1.10**	**1.98**	**2.88**
A Pub	Conv. A	1.38	1.39	1.16	2.38	4.24
Conv. A	Conv. B	1.86	1.35	1.23	4.04	6.04
Conv. B	B Sub	2.94	2.19	2.25	7.72	8.11
A Pub	***B Sub***	***6.18***	***4.93***	***4.64***	***14.14***	***18.39***
B Pub	**B Sub**	**2.70**	**2.31**	**2.14**	**6.42**	**6.64**
B Pub	Conv. B	2.98	2.36	2.63	21.04	22.95
Conv. B	Conv. A	2.24	1.47	1.48	4.85	6.06
Conv. A	A Sub	1.18	1.17	1.00	2.54	3.86
B Pub	***A Sub***	***6.40***	***5.00***	***5.11***	***28.43***	***32.87***

In Table 4.1, we tried to show the average time delays in milliseconds for a message to reach from publisher to subscriber.

A message, which is published to the topic named "track" in the message broker of the access control system of Building A, both goes to a subscriber microservice in the Building A's system and to a subscriber microservice in the Building B's system with the help of our proposed tool. The first row shows the time delays between the publisher and the subscriber in Building A's system. The second, third, and fourth rows show the path for a message to be forwarded to Building B's system with our

proposed tool. The second row shows the time delays between the publisher and the Middleware Converter for Building A. The third row shows the time delays between the Middleware Converter for Building A and the Middleware Converter for Building B. The fourth row shows the time delays between the Middleware Converter for Building B and the subscriber microservice in Building B's system. The fifth row shows the overall time for a message to be forwarded from the publisher microservice of Building A's system to the subscriber microservice of Building B's system.

A message, which is published to the topic name "movement" in the message broker of the access control system of Building B, both goes to a subscriber microservice in the Building B's system and to a subscriber microservice in the Building A's system with the help of our proposed tool. The sixth row shows the time delays between the publisher and the subscriber in Building B's system. The seventh, eighth, and ninth rows show the path for a message to be forwarded to Building A's system with our proposed tool. The seventh row shows the time delays between the publisher and the Middleware Converter for Building B. The eighth row shows the time delays between the Middleware Converter for Building B and the Middleware Converter for Building A. The ninth row shows the time delays between the Middleware Converter for Building A and the subscriber microservice in Building A's system. The tenth row shows the overall time for a message to be forwarded from the publisher microservice of Building B's system to the subscriber microservice of Building A's system.

From Table 4.1, we can understand that our interoperability tool can forward tens of thousands of messages between different systems based on different message brokers within a second. In the first three tests, with 40, 100, and 200 messages per second from each publisher, the time delays between each component are very similar. However, after the fourth test with 400 messages per second from each publisher, time delays between components that are communicating with Kafka, the message broker of the access control system of Building B, increased drastically. In [31, 39], the authors stated that the throughput of RabbitMQ is better than the throughput of Kafka in the basic setup, which is for a single node, single producer/channel, single-partition, and no replication. However, Kafka's performance can be increased significantly by increasing the number of partition counts. In our case study, we use both RabbitMQ and Kafka in their basic setup, that is the reason why time delays between compo-

nents around RabbitMQ are smaller than the time delays between components around Kafka.

Our primary concern while developing our interoperability tool was achieving interoperability among different event-driven microservice systems using different message brokers as their publish/subscribe mechanism. Also there were no directly related work for interoperability of message brokers. That is why we could not compare our tool with other tools in terms of attributes such as performance and usability. We only presented time delays to give intuition about the performance of our interoperability tool.

CHAPTER 5

CONCLUSION AND FUTURE WORK

5.1 Conclusion

In this thesis, we presented our solution to handle interoperability among event-driven microservice-based systems using message brokers that execute their publish/subscribe mechanism. Our proposed tool is also designed as an event-driven microservice-based publish/subscribe system. The components named Middleware Converters are publishers and subscribers of the Common Middleware component, which is the message broker of our proposed tool. Common Middleware, the message broker of the tool, is used to send messages from and to Middleware Converters in order to forward messages among message brokers of different systems. Using another message broker, namely Common Middleware, and using a separate Middleware Converter per system let us design a tool that is extensible to other systems using other message broker applications. Based on our observations from the case study, we can state that our interoperability tool can forward tens of thousands of messages among different systems based on different message brokers within a second.

5.2 Future Work

Our experimentation in the case study was only, for the sake of simplicity, between two different systems using different message broker technologies as their publish/-subscribe mechanism. Doing tests with more than two systems and observing how the performance of our interoperability tool is going to be affected could be future work.

To make our tool better and faster, we need to lower the communication delay between different systems. However, a message created in one system is required to travel also in our tool to be forwarded to the target system. The time a message spends in the interoperability tool could be a bottleneck if there are millions of messages coming from different systems, so studying and improving the performance of the interoperability tool could be another future work.

Currently, our tool does not do anything about semantic interoperability, it only forwards messages from one system to other systems, which are syntactical interoperability. Adding a semantic interoperability feature, like automatic topic matching among different systems, to our interoperability tool could be a good future work.

Our tool currently forwards the messages among systems without touching the message itself. An adaptation of message formats can be another future work.

Our tool does not offer a search mechanism to find a microservice in a system. Adding search mechanism to find relevant microservices to subscribe for a system could be a future work, since our tool is integrating diverse systems and it is the only integration and search point.

Currently there is no user interface to configure and monitor the integration of systems. Creating an user interface for our tool could be another future work.

REFERENCES

[1] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: current and future directions," *ACM SIGAPP Applied Computing Review*, vol. 17, no. 4, pp. 29–45, 2018.

[2] "Microservice architecture style - azure application architecture guide." Available at `https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices`.

[3] S. Newman, *Building microservices*. " O'Reilly Media, Inc.", 2021.

[4] B. M. Michelson, "Event-driven architecture overview," *Patricia Seybold Group*, vol. 2, no. 12, pp. 10–1571, 2006.

[5] "Event-driven architecture." Available at `https://aws.amazon.com/event-driven-architecture/`.

[6] I. C. Education, "What are message brokers?," Jan 2020. Available at `https://www.ibm.com/cloud/learn/message-brokers`.

[7] J. Bloch, "How to design a good api and why it matters," in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pp. 506–507, 2006.

[8] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 000149–000154, IEEE, 2018.

[9] R. C. Martin, "The single responsibility principle," May 2014. Available at `https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html`.

[10] N. Serrano, J. Hernantes, and G. Gallardo, "Service-oriented architecture and legacy systems," *IEEE software*, vol. 31, no. 5, pp. 15–19, 2014.

[11] "Devops with docker." Available at https://devopswithdocker.com/part1/.

[12] C. Boettiger, "An introduction to docker for reproducible research," *ACM SIGOPS Operating Systems Review*, vol. 49, no. 1, pp. 71–79, 2015.

[13] B. B. Rad, H. J. Bhatti, and M. Ahmadi, "An introduction to docker and analysis of its performance," *International Journal of Computer Science and Network Security (IJCSNS)*, vol. 17, no. 3, p. 228, 2017.

[14] I. Miell and A. Sayers, *Docker in practice*. Simon and Schuster, 2019.

[15] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends," *Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[16] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.

[17] "What is pub/sub?." Available at https://cloud.google.com/pubsub/docs/overview.

[18] M. C. Kaya, A. Karamanlioglu, İ. Ç. Çetintaş, E. Çilden, H. Canberi, and H. Oğuztüzün, "A configurable gateway for dds-hla interoperability," in *Proceedings of the 2019 Summer Simulation Conference*, pp. 1–11, 2019.

[19] B. Elvesæter, A. Hahn, A.-J. Berre, and T. Neple, "Towards an interoperability framework for model-driven development of software systems," in *Interoperability of enterprise software and applications*, pp. 409–420, Springer, 2006.

[20] M. Aragão, P. Moreno, and A. Bernardino, "Middleware interoperability for robotics: A ros–yarp framework," *Frontiers in Robotics and AI*, vol. 3, p. 64, 2016.

[21] M. A. Jarwar, S. Ali, M. G. Kibria, S. Kumar, and I. Chong, "Exploiting interoperable microservices in web objects enabled internet of things," in *2017 Ninth International Conference on Ubiquitous and Future Networks (ICUFN)*, pp. 49–54, IEEE, 2017.

[22] E. S. Pramukantoro and H. Anwari, "An event-based middleware for syntactical interoperability in internet of things.," *International Journal of Electrical & Computer Engineering (2088-8708)*, vol. 8, no. 5, 2018.

[23] H. Mueller, "What is software interoperability and how can it boost profits and productivity?," Jun 2021. Available at https://www.formstack.com/resources/blog-software-interoperability.

[24] "Eprosima integration service." Available at https://integration-service.docs.eprosima.com/en/latest/.

[25] N. Nurseitov, M. Paulson, R. Reynolds, and C. Izurieta, "Comparison of json and xml data interchange formats: a case study.," *Caine*, vol. 9, pp. 157–162, 2009.

[26] T. D. Team, "What is interoperability and why do we need it?," Apr 2018. Available at https://www.dermengine.com/blog/dermatology-emr-software-integration.

[27] A. Bayramcavus, M. C. Kaya, and A. H. Dogru, "Interoperability of microservice-based systems," in *2021 13th International Conference on Electrical and Electronics Engineering (ELECO)*, pp. 594–598, 2021.

[28] "Rabbitmq documentation." Available at https://www.rabbitmq.com/documentation.html.

[29] "Apache kafka documentation." Available at https://kafka.apache.org/documentation/#implementation.

[30] "Zeromq | get started." Available at https://zeromq.org/get-started/.

[31] P. Dobbelaere and K. S. Esmaili, "Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper," in *Proceedings of the 11th ACM international conference on distributed and event-based systems*, pp. 227–238, 2017.

[32] L. Li, W. Chou, W. Zhou, and M. Luo, "Design patterns and extensibility of rest

api for networking applications," *IEEE Transactions on Network and Service Management*, vol. 13, no. 1, pp. 154–167, 2016.

[33] L. Li and W. Chou, "Design and describe rest api without violating rest: A petri net based approach," in *2011 IEEE International Conference on Web Services*, pp. 508–515, IEEE, 2011.

[34] V. John and X. Liu, "A survey of distributed message broker queues," *arXiv preprint arXiv:1704.00411*, 2017.

[35] "Know about access control systems and their types with features," Sep 2021. Available at `https://www.elprocus.com/understanding-about-types-of-access-control-systems/`.

[36] A. Venckauskas, N. Morkevicius, and K. Kulikauskas, "Study of finger vein authentication algorithms for physical access control," *Elektronika ir elektrotechnika*, no. 5, pp. 101–104, 2012.

[37] A. Bayramcavus, "Thesis code," Sep 2021. Available at `https://github.com/alibayramcavus/ThesisCode`.

[38] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pp. 243–246, IEEE, 2017.

[39] W. Sriborrirux and P. Laortum, "Healthcare center iot edge gateway based on containerized microservices," in *Proceedings of the 2020 4th International Conference on Intelligent Systems, Metaheuristics & Swarm Intelligence*, pp. 24–29, 2020.

APPENDIX A

DOCKERFILES

A.1 Publisher and Subscriber Services for System using RabbitMQ as Message Broker

```
FROM python:alpine3.14

WORKDIR /app

RUN pip install pika

COPY *.py .

CMD ["python", "ServiceSimulator.py"]
```

A.2 Publisher and Subscriber Services for System using Kafka as Message Broker

```
FROM python:alpine3.14

WORKDIR /app

RUN pip install pika
RUN pip install kafka-python

```

```
COPY *.py .

CMD ["python", "ServiceSimulator.py"]
```

A.3 Middleware Converter for System using RabbitMQ as Message Broker

```
FROM python:alpine3.14

WORKDIR /app

RUN pip install pika
RUN pip install flask
RUN pip install requests

COPY *.py .

CMD ["python", "main.py"]
```

A.4 Middleware Converter for System using Kafka as Message Broker

```
FROM python:alpine3.14

WORKDIR /app

RUN pip install pika
RUN pip install kafka-python
RUN pip install flask
RUN pip install requests

COPY *.py .

```

```
CMD ["python", "main.py"]
```

A.5 Converter Configurator

```
FROM python:alpine3.14

WORKDIR /app

RUN pip install flask
RUN pip install requests

COPY *.py .

CMD ["python", "main.py"]
```

A.6 Log Manager

```
FROM python:slim-buster

WORKDIR /app

RUN pip install pika
RUN pip install psycopg2-binary

COPY *.py .

CMD ["python", "main.py"]
```

APPENDIX B

DOCKER-COMPOSE FILES

B.1 Starting All Message Broker Applications in Each System and Interoperability Tool

```yaml
version: "3.8"

services:
  # sys1 rabbit
  rabbit_sys1:
    image: rabbitmq:3.9-management
    networks:
      - sys1_net

  # sys2 zookeeper
  zookeeper:
    image: 'bitnami/zookeeper:latest'
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
    networks:
      - sys2_net

  # sys2 kafka
  kafka:
    # https://hub.docker.com/r/bitnami/
    # kafka/#full-configuration
```

```
    # Accessing Kafka with internal and external clients
    image: 'bitnami/kafka:latest'
    ports:
      - 5682:5682
    environment:
      - KAFKA_BROKER_ID=1
      - KAFKA_CFG_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
      - >
        KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP=
        CLIENT:PLAINTEXT,EXTERNAL:PLAINTEXT
      - >
        KAFKA_CFG_LISTENERS=CLIENT://:9092,
        EXTERNAL://:5690
      - >
        KAFKA_CFG_ADVERTISED_LISTENERS=
        CLIENT://kafka:9092,EXTERNAL://localhost:5690
      - KAFKA_INTER_BROKER_LISTENER_NAME=CLIENT
    networks:
      - sys2_net
    depends_on:
      - zookeeper


  # tool rabbit
  rabbit_tool:
    image: rabbitmq:3.9-management
    networks:
      - tool_net

  # log rabbit
  rabbit_log:
```

```
    image: rabbitmq:3.9-management
    networks:
      - log_net


networks:
  sys1_net:
    external:
      name: sys1_net
  sys2_net:
    external:
      name: sys2_net
  tool_net:
    external:
      name: tool_net
  log_net:
    external:
      name: log_net
```

B.2 Starting Services of First Access Control System

```
version: "3.8"

services:
  sys1-pub1:
    image: service_rabbit
    environment:
      - SERVICE_NAME=sys1-pub1
      - RABBITMQ_IP=rabbit_sys1
      - RABBITMQ_PORT=5672
      - PUBLISHERS_CONF=track--2500
      - LOG_ENABLED=True
```

```
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
    restart: always
    networks:
      - sys1_net
      - log_net
    depends_on:
      - sys1-sub1
      - sys1-sub2


  sys1-sub1:
    image: service_rabbit
    environment:
      - SERVICE_NAME=sys1-sub1
      - RABBITMQ_IP=rabbit_sys1
      - RABBITMQ_PORT=5672
      - SUBSCRIBERS_CONF=track
      - LOG_ENABLED=True
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
    restart: always
    networks:
      - sys1_net
      - log_net

  sys1-sub2:
    image: service_rabbit
    environment:
      - SERVICE_NAME=sys1-sub2
```

```
      - RABBITMQ_IP=rabbit_sys1
      - RABBITMQ_PORT=5672
      - SUBSCRIBERS_CONF=buildingB_track
      - LOG_ENABLED=True
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
    restart: always
    networks:
      - sys1_net
      - log_net

networks:
  sys1_net:
    external:
      name: sys1_net
  log_net:
    external:
      name: log_net
```

B.3 Starting Services of Second Access Control System

```
version: "3.8"

services:
  sys2-pub1:
    image: service_kafka
    environment:
      - SERVICE_NAME=sys2-pub1
      - KAFKA_IP=kafka
      - KAFKA_PORT=9092
      - PUBLISHERS_CONF=movement--2500
```

```
      - LOG_ENABLED=True
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
    networks:
      - sys2_net
      - log_net
    depends_on:
      - sys2-sub1
      - sys2-sub2


  sys2-sub1:
    image: service_kafka
    environment:
      - SERVICE_NAME=sys2-sub1
      - KAFKA_IP=kafka
      - KAFKA_PORT=9092
      - SUBSCRIBERS_CONF=movement
      - LOG_ENABLED=True
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
    networks:
      - sys2_net
      - log_net

  sys2-sub2:
    image: service_kafka
    environment:
      - SERVICE_NAME=sys2-sub2
      - KAFKA_IP=kafka
```

```
      - KAFKA_PORT=9092
      - SUBSCRIBERS_CONF=buildingA_movement
      - LOG_ENABLED=True
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
    networks:
      - sys2_net
      - log_net

networks:
  sys2_net:
    external: true
    name: sys2_net
  log_net:
    external: true
    name: log_net
```

B.4 Starting Log Service

```
version: "3.8"

services:
  masterdb:
    image: postgres:13.4-alpine3.14
    volumes:
      - db_data:/var/lib/postgresql/data
    environment:
      - POSTGRES_PASSWORD=masterdbpassword
    ports:
      - 5000:5432
    networks:
```

```
      - log_net

  logger:
    image: newlogger
    environment:
      - DB_IP=masterdb
      - DB_PORT=5432
      - DB_NAME=logDB
      - DB_PASSWORD=masterdbpassword
      - BROKER_IP=rabbit_log
      - BROKER_PORT=5672
    restart: always
    depends_on:
      - masterdb
    networks:
      - log_net

networks:
  log_net:
    external:
      name: log_net

volumes:
  db_data:
    external: true
    name: db_data
```

B.5 Starting Converter Configurator and Middleware Converters of the Interoperability Tool

```
version: "3.8"

services:
  configurator:
    image: configurator
    environment:
      - CORE_SERVICE_PORT=8080
      - CORE_BROKER_IP=rabbit_tool
      - CORE_BROKER_PORT=5672
    ports:
      - 8080:8080
    networks:
      - tool_net

  sys1-manager:
    image: manager_rabbit
    environment:
      - SYSTEM_NAME=sys1-manager
      - MANAGER_IP=sys1-manager
      - MANAGER_PORT=8080
      - SYSTEM_BROKER_IP=rabbit_sys1
      - SYSTEM_BROKER_PORT=5672
      - CORE_BROKER_IP=rabbit_tool
      - CORE_BROKER_PORT=5672
      - CORE_SERVICE_IP=configurator
      - CORE_SERVICE_PORT=8080
      - LOG_ENABLED=True
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
```

```
    networks:
      - sys1_net
      - tool_net
      - log_net
    depends_on:
      - core

  sys2-manager:
    image: manager_kafka
    environment:
      - SYSTEM_NAME=sys2-manager
      - MANAGER_IP=sys2-manager
      - MANAGER_PORT=8080
      - SYSTEM_BROKER_IP=kafka
      - SYSTEM_BROKER_PORT=9092
      - CORE_BROKER_IP=rabbit_tool
      - CORE_BROKER_PORT=5672
      - CORE_SERVICE_IP=configurator
      - CORE_SERVICE_PORT=8080
      - LOG_ENABLED=True
      - LOG_ONLY_AT_THE_END=True
      - LOGBROKER_IP=rabbit_log
      - LOGBROKER_PORT=5672
    networks:
      - sys2_net
      - tool_net
      - log_net
    depends_on:
      - core


networks:
```

```
  sys1_net:
    external:
      name: sys1_net
  sys2_net:
    external:
      name: sys2_net
  tool_net:
    external:
      name: tool_net
  log_net:
    external:
      name: log_net
```