BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DISTRIBUTED TASKS IN AN ENTERPRISE COMPUTING ENVIRONMENT

DISTRIBUOVANÉ SPOUŠTĚNÍ ÚLOH V PODNIKOVÉM VÝPOČETNÍM PROSTŘEDÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **MILAN TICHAVSKÝ**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. RADEK BURGET, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2023

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

148401

Institut:	Department of Information Systems (UIFS)
Student:	**Tichavský Milan**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Distributed Tasks in an Enterprise Computing Environment**
Category:	Parallel and Distributed Computing
Academic year:	2022/23

Assignment:

1. Study principles of distributed computing, workload isolation with Linux containers, workload management with Kubernetes/OpenShift, and AMQP message bus communication.
2. Study the related industry standards, such as Jenkins.
3. Design a lightweight event-based task framework for executing arbitrary workloads in OpenShift.
4. Implement the design from the previous step. Include comprehensive unit and functional tests.
5. Evaluate the solution by comparing it to industry standards. Weight the advantages and disadvantages of custom-tailored solutions and one-size-fits-all solutions, especially in the broader context of the business—integration with other systems, evolving objectives, etc.
6. Summarize the achieved results.

Literature:

- Hohpe, G., Woolf, B.: Enterprise integration patterns, Addison-Wesley, 2003.
- Further resources according to the instructions of the supervisor.

Requirements for the semestral defence:
Items 1 through 3

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Burget Radek, doc. Ing., Ph.D.**
Consultant:	Andrew Hills
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	26.10.2022

Abstract

In a microservices architecture, messages are commonly used for communication between individual services. There is a general need to listen to messages sent on a message bus and react to them by triggering job execution. This thesis presents all essential considerations to be made when solving this problem. It comes up with an execution framework that enables such workflow by executing tasks in arbitrary container images on OpenShift. The solution consists of a Flask API that contains the execution logic and a STOMP client written in Python that receives messages from the message bus and sends them to the API. Test scenarios are included to showcase the functionality, and the solution is evaluated by comparing it with a Jenkins setup. Also, an alternative design using Tekton pipelines is discussed. The second problem this thesis focuses on is the execution of periodically scheduled tasks and suggests using Kubernetes CronJob objects instead of implementing anything custom.

Abstrakt

V architektuře orientované na mikroslužby jsou zprávy běžně používaným prostředkem pro komunikaci mezi jednotlivými službami. Obecně je zde potřeba naslouchat zprávám odeslaným na sběrnici a reagovat na ně spouštěním úloh. Tato práce prezentuje všechny podstatné úvahy k vyřešení tohoto problému. Přichází s rámcem pro spouštění úloh, který vykonává úlohy v libovolných kontejnerech na OpenShiftu. Řešení se skládá z API napsaného ve Flasku, které obsahuje spouštěcí logiku, a klienta, který přes STOMP příjmá zprávy ze sběrnice a poslílá je na API. Součástí jsou i testovací scénáře, které předvádějí funkčnost celého systému. Řešení je vyhodnocováno porovnáváním s existující aplikací postavené na nástroji Jenkins. Rovněž je diskutovaný alternativní návrh využívající Tekton. Druhým problémem, kterým se tato práce zabývá, je provádění pravidelně naplánovaných úloh. Namísto implementace vlastního řešení navrhuje použití Kubernetes objektů CronJob.

Keywords

distributed computing, job execution, microservices, messaging, STOMP, ActiveMQ, OpenShift, Kubernetes, containerization, Tekton

Klíčová slova

distrubuované výpočty, spouštění úloh, mikroslužby, posílání zpráv, STOMP, ActiveMQ, OpenShift, Kubernetes, kontejnerizace, Tekton

Reference

TICHAVSKÝ, Milan. *Distributed Tasks in an Enterprise Computing Environment.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Radek Burget, Ph.D.

Rozšířený abstrakt

Software je v dnešní době úplně všude. Ať už běží v telefonech a počítačích nebo řídí dopravu, všechen tento software musí být nějakým způsobem vytvořen, dodán zákazníkům a udržován aktualizacemi a bezpečnostními záplatami. Celý proces tvorby aplikací začíná sběrem požadavků a specifikací, na základě čehož je navržen, naimplementován, otestován, zabalen do instalačních balíčků a nahrán na servery po celém světě, díky čemuž si jej mohou uživatelé jednoduše nainstalovat na svá zařízení.

Přirozeně, velké množství nástrojů a automatizace je využívano pro každý krok tohoto procesu, aby byla cena programů co nejnižší a aktualizace mohly být dodávány spolehlivě. Automatizace zároveň pomáhá udržovat nezbytné bezpečnostní standardy a v případě zjištění zranitelnosti v nějaké populární komponentě umožňuje okamžitě zareagovat a zaktualizovat celé portfolio aplikací. Taková automatizace, která se stará převážně o část procesu, kdy jsou aplikace sestavovány a nahrávány na servery, je provozována například ve firmě Red Hat. Skládá se z množství aplikací, které spolu komunikují pomocí zasílání zpráv. Tyto zprávy si mezi sebou nevyměňují přímo, ale přes sběrnici.

Cílem této práce je vyvinout řešení, které bude naslouchat na zprávy poslané po sběrnici. Podle uživatelské konfigurace je bude filtrovat a jako reakci spouštět úlohy na OpenShiftu. Díky tomu bude možné nahradit existující službu postavenou na nástroji Jenkins, která má bezpečnostní problémy a je náročná na údržbu. Nicméně ne všechny úlohy jsou spouštěny současným řešením jako reakce na zprávu na sběrnici. Některé z nich jsou spouštěny na základě časového plánu. Původním záměrem bylo vyvinout software, který by spuštěl oba typy úloh. Jenže během studia souvisejících technologií začalo být jasné, že pro provádění pravidelně naplánovaných úloh stačí využít existujících nástrojů, které jsou součástí OpenShiftu. Tato práce tedy doporučuje použití Kubernetes CronJob objektů pro migraci tohoto typu úloh.

Výsledné řešení se tedy zaměřilo na reaktivní spouštění úloh. Skládá se ze dvou aplikací – z API napsaného ve Flasku a klienta, který přes STOMP příjmá zprávy ze sběrnice a poslílá je na API. Důvod pro rozdělení funkcionality na dvě části je ten, že kromě plně automatizovaného spouštění aplikací je vyžadováno, aby mohli uživatelé manuálně předat zprávu aplikaci a tím spustit úlohu. Toto se hodí převážně pro ladění nebo v situaci, kdy z nějakého důvodu nebyla úloha vykonána. API tedy dostane požadavek obsahující zprávu, na základě logiky definované uživatelem filtruje zprávy, které má ignorovat a které ne. Poté vygeneruje Kubernetes Job objekt, do kterého vloží data z obdržené zprávy, a nakonec pošle požadavek na vytvoření tohoto objektu na server s OpenShiftem. Klient při startu načte ze sdílené konfigurace fronty, ke kterým se má připojit, a pak pouze přeposílá zprávy.

Výsledkem této práce je tedy systém dvou aplikací, které jsou schopny spouštět úlohy jako reakci na zprávy na sběrnici. Obě aplikace jsou schopny reagovat na chybové stavy a výpadek jednotlivých mikroslužeb, se kterými komunikují. Důraz byl kladen na dobře provedené logování, jelikož se jedná o jeden z nejlepších způsobů, jak zjišťovat aktuální stav aplikace a ladit problémy. Ověření funkčnosti celého systému lze udělat manuálně za pomocí instrukcí v README.md a přiloženého pomocného programu. Testovací prostředí využívá lokálně běžícího ActiveMQ brokera a umožnuje otestování obou způsobů přihlášení, tedy jak přihlášení pomocí uživatelského jména a hesla, tak pomocí certifikátů. Pro zvýšení kvality a snížení pravděpodobností regresí je řešení pokryto množstvím automatizovaných jednotkových testů.

Distributed Tasks in an Enterprise Computing Environment

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Ing. Radek Burget, Ph.D. The supplementary information was provided by Andrew Hills, B.S. I have listed all the literary sources, publications, and other sources used during this thesis's preparation.

....
Milan Tichavský
May 7, 2023

Acknowledgements

I want to thank my supervisor, doc. Ing. Radek Burget, Ph.D., for his help and support throughout writing this thesis. Also, I would like to express my sincere gratitude to Andrew Hills, B.S., for consulting my work and providing valuable feedback.

Contents

1 Introduction **4**

2 Distributed systems and message communication **5**
2.1 Distributed systems and microservice architecture . . . 5
2.2 Communication in a microservice architecture . . . 6
2.3 Queues, topics, and virtual topics . . . 7
2.4 Red Hat's messaging infrastructure and its limitations . . . 7
2.5 Messaging protocols . . . 9
2.6 Simple (or Streaming) Text Orientated Messaging Protocol . . . 10

3 Linux containers and orchestration **12**
3.1 Linux containers . . . 12
3.2 Container orchestration . . . 13
3.3 OpenShift . . . 16
3.4 Tekton . . . 16

4 Existing Solution **18**
4.1 Jenkins setup . . . 18
4.2 Job definitions format . . . 18
4.3 Jenkins problems . . . 20

5 Design **21**
5.1 Problem definition . . . 21
5.2 Scheduled jobs . . . 21
5.3 Requirements for triggered jobs . . . 22
5.4 The design overview . . . 22
5.5 The client . . . 23
5.6 The API . . . 25

6 Implementation **30**
6.1 Configuration files . . . 30
6.2 Generation of Job resources . . . 32
6.3 Logging . . . 34
6.4 Arbitrary code execution . . . 35
6.5 Flask and error handling . . . 35
6.6 stomp.py library . . . 36

7 Testing and Evaluation **38**

7.1 Unit testing and continuous integration 38
7.2 Comparison with similar tools 39

8 Conclusion **41**

Bibliography **42**

A CD Contents **43**

List of Figures

2.1 Diagrams visualizing the differences between queue, topic, and virtual topic. Source: Uhrig [11]. 8
2.2 Communication between a message broker and a client that is connected as a subscriber. The acknowledgment mode is auto, so there are no ACK frames sent. 11

3.1 Kubernetes architecture. The control plane in the text refers to a master node in the diagram. Source: Kaplarevic [5]. 13
3.2 Diagram visualizing a Tekton Trigger and its communication with a ClusterInterceptor. Source: Tekton documentation [7]. 17

5.1 The design overview. The message bus and OpenShift are part of the existing infrastructure. The filled line marks an HTTP request, and the dashed line marks a response to this request. 23

Chapter 1

Introduction

Software is pretty much everywhere – it is running on our smartphones and laptops, but also in places where it is less visible, such as traffic light controllers. This software must be manufactured, delivered to customers, and supported with necessary updates and patches. This manufacturing process starts with a collection of the business requirements or a specification of what has to be done, based on which the application is designed, implemented, tested, packaged, and then uploaded to servers across the world so that users can comfortably install it on their computers.

Of course, a lot of tools and automation are used for every step of the process so that the costs of the resulting software are kept as low as possible and updates to applications can be delivered reliably. Lately, there has been much more emphasis on the security of these systems as threats like supply chain attacks are becoming more popular among bad actors. Automation helps maintain all the necessary security standards. Suppose a vulnerability is found in some popular software component. In that case, automation can be used to quickly rebuild the entire portfolio of products leading to a much shorter response time than what engineers could achieve with manual work.

An example of such automation is Red Hat's release pipeline, which is used internally for the packaging and uploading part of the process while enabling the execution of some security checks. It comprises many applications that communicate by exchanging messages through a message bus. This thesis aims to build a lightweight task execution framework for listening to these messages, filtering through them, and as a reaction, executing arbitrary workloads in OpenShift. This will enable to sunset an existing service, which serves a similar function but has security problems and high maintenance costs.

However, not all jobs in the existing systems are triggered by incoming messages. Some of them are launched according to a given schedule. So, to address the need, this thesis will discuss how existing tools can be leveraged to execute these scheduled jobs.

The most exciting parts of this project are digging deep into the messaging infrastructure and getting familiar with OpenShift, which has become a widely used platform for workload execution. Also, working on an application that will be running in production one day is a big motivation for me.

Chapter 2

Distributed systems and message communication

This chapter describes all essential message communication concepts necessary to understand the solution's context and to design the task execution framework in later chapters. It starts from a broader perspective discussing the distributed systems and microservice architecture, which often leverages messaging for communication between components. Then it describes the basic messaging patterns and discusses the specifics of Red Hat's internal message bus and its limitations, which need to be understood by anyone interacting with the service. Finally, it focuses on the protocols the message bus uses to communicate with its clients.

2.1 Distributed systems and microservice architecture

There are two main approaches to executing workloads when solving a problem. The centralized approach aims to keep all the computation centralized on a single computer, while the distributed approach utilizes computational resources across multiple separate computation nodes.

Building distributed systems can lead to better system performance by enabling more scaling options. It also removes central points of failure, resulting in higher availability. However, this architecture brings its own problems related to managing multiple computational nodes.

An example of a centralized system is a traditional monolithic architecture, where the application is built as one self-contained, independent unit with one codebase handling all the business requirements. Therefore, the whole application must be rebuilt and deployed to make any changes.

An example of a distributed system is a microservice architecture. In this architecture, the application is developed as a collection of services. Each service has a single responsibility and is self-contained. It has its own data persistence and is deployed independently of other services, providing a simple application programming interface (API) for communication.

Because of the independent deployment, services can be updated and patched faster than the development process of large monolithic applications enables. Smaller services with a single responsibility have a smaller codebase with fewer dependencies. Therefore, each application is maintained by more focused teams, leading to better communication

and less management overhead. When applications are decoupled, adding or removing a new service is a relatively easy task.

While each service becomes easier to develop, the whole system is more complex. Debugging becomes more challenging because a bug could occur in any of the services or somewhere between them. Also, it might be unclear which team should take responsibility and the ownership of identification of the problem and implementing the fix. Writing integration tests is harder because applications are updated independently, and environment setup takes more work. Anyone designing a microservice architecture should set some basic guidelines for the choice of languages and frameworks so that the technology stack is more cohesive and have a standardized approach for logging to follow users' requests between services.

2.2 Communication in a microservice architecture

With different architectures come different ways to communicate. As Anil [1] puts it, "In a monolithic application running on a single process, components invoke one another using language-level method or function calls." Unlike microservices, the communication is usually tightly coupled, even though decoupled programs can be written using dependency injection. Because function calls are relatively cheap, there is much more communication between components.

One of the main principles of microservices design is to have loosely coupled services because the network is not always reliable, and failure of a single application should not lead to a crash of the entire system. The inter-process communication is expensive, and the network has some latency, so there is pressure to have as few remote procedure calls as possible, leading to more information being passed in every call. For all these reasons, microservices have to integrate asynchronously.

One of the popular options is to use a synchronous HTTP/HTTPS protocol. Not only is this an excellent protocol for exposing the microservice application to the outside world, but it is also widely used for communication between services. It is preferred for integration to work well when the request is created asynchronously. It means that while the application is waiting for the response it needs to receive to continue, the process is not blocked, and it can create another request or execute a callback function. Usually, services implement API that conforms to the representational state transfer (REST) design principles. In this context, the most important constraint is statelessness, which means every request contains all the necessary information and no data related to the request is stored on the server. Also, note that one of the trade-offs of using HTTP/HTTPS communication is that it has a single receiver.

Another option is to use some asynchronous protocol, such as AMQP, which leverages asynchronous messages. Unlike the previous approach, the client sends the message without expecting a response, at least not immediately. It does not send the message directly to other services but rather to the message broker, which acts as an intermediary between clients, takes care of routing, and can persist messages if the client is temporarily disconnected. This means that even though the client sends one message to the message broker, it can be sent to zero or multiple receivers.

Besides communicating with the message broker, clients can send messages to the message bus. While these two terms have similar and overlapping meanings, I think it is good to explain them entirely, at least how to understand them in the context of this thesis. According to Hohpe [3], "A message bus is a combination of a canonical data model, a

common command set, and a messaging infrastructure to allow different systems to communicate through a shared set of interfaces." The main distinction is that referring to the message bus is often meant as referring to the whole messaging infrastructure. This infrastructure is generally decentralized and can consist of several message brokers – middleware applications that provide the messaging functionality.

2.3 Queues, topics, and virtual topics

There are several different ways message brokers can deliver messages between services. Let's explore basic patterns supported by Apache ActiveMQ „Classic" 5[1], the message server part of Red Hat's infrastructure with which my solution interacts.

Queues facilitate a direct communication channel between producers and consumers. Consumer reads messages one by one, and ActiveMQ removes them from the queue on receiving the acknowledgment messages. If multiple consumers are connected to the queue, messages are load-balanced to them so that only one gets the message. This pattern is also called a point-to-point channel. Queues are durable by default, meaning messages are stored on the message bus even after the client disconnects, waiting in a queue until they are not processed.

Topics, on the other hand, provide a publish-subscribe communication channel. Multiple consumers can subscribe to the given topic, and once the publisher sends a message to the topic, this message is replicated and delivered to each of the consumers. Topics are not durable, so clients must be online to receive messages.

To get the best of both worlds, there are *virtual topics*. Producers publish messages on topics, and each subscriber has a queue called the consumer queue associated with it, which acts as a subscription to the given topic. ActiveMQ then delivers messages from these topics to their corresponding consumer queues. This enables a publish-subscribe communication pattern with durable consumers without setting up durable topics. Support for virtual topics (or virtual destinations) is one of the reasons why ActiveMQ brokers were chosen for the message bus architecture. See Figure 2.1 for a better understanding of these concepts.

Note that regular queues are used in the local development environment. This is because, from the client's point of view, the consumer queue is just a regular queue that adheres to certain naming conventions. Therefore, there is no need to bother setting up virtual topics to test the application.

2.4 Red Hat's messaging infrastructure and its limitations

The following chapter is adopted from internal documentation [9], [10]. In Red Hat's pipeline context, the message bus comprises highly available Apache ActiveMQ "Classic" 5 messaging servers (abbreviated as ActiveMQ in the following text). These ActiveMQ brokers are grouped into several logical clusters, each consisting of a database and two ActiveMQ brokers – one that delivers the messages and the second ready for failover. Applications connecting to the message bus through the provided server-side load balancer do not need to worry much about the specifics of how those clusters are networked together and can rely on the message bus to deliver messages to the expected endpoints with some magic happening under the hood.

[1]Project website: https://activemq.apache.org/components/classic/

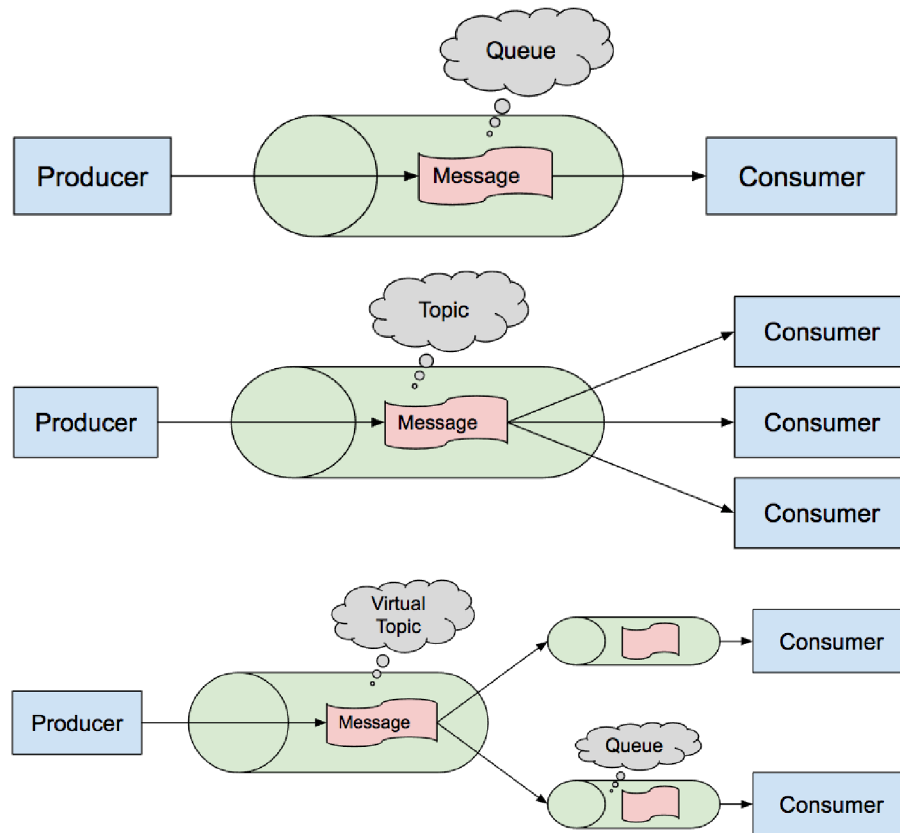

Figure 2.1: Diagrams visualizing the differences between queue, topic, and virtual topic. Source: Uhrig [11].

An essential feature of message bus architecture is the guaranteed delivery of messages, usually accomplished using message persistence. It is a best practice to persist messages between transactions and during them so that at any moment of the message journey through the infrastructure, there is at least one database storing the message. Once the transaction is completed, which is usually signaled by sending an acknowledgment, the message is removed from the database of the sending application. This way, even if any number of services suddenly crashes, the message will still exist in the system once those services come back online. Message persistence also covers the case if the application is offline when the message should be delivered. However, there are usually timeouts on how long messages are retained before they are deleted, so the service cannot be offline for too long if it does not want to miss any messages.

This aspect of the infrastructure design that focuses on not losing any messages has its consequences. All these disk reads and writes come with performance costs, and while it can guarantee that messages are delivered, it cannot guarantee that they will be delivered only once. Consider the case when the broker fails right after the transaction is finished before it can delete the message from its database. Then, on startup, it loads the message from the database and tries to redeliver it, so it is delivered twice. Therefore, if the goal is to execute jobs as a reaction to receiving a message, these jobs have to be idempotent, meaning the effect of executing the job once is the same as if executed multiple times.

As mentioned at the beginning of this section, the message bus is considered a highly available service that runs all the time. Therefore, to save time and engineering effort, it is common for publishers not to store messages locally, which may lead to missed messages if the message bus goes offline.

Similarly, it is an important decision when designing a subscriber if it is worth persisting messages until they are fully processed. Subscribers usually prefetch many messages, so when the application crashes, it loses multiple messages if they are not persisted. Alternatively, the application can use the message bus databases and acknowledge messages only after processing them.

Another limitation of the message bus is that it cannot guarantee that messages are delivered in the same order they were sent. Thinking about task execution in the context of this thesis, it is not a problem, but it is something to remember when dealing with messaging systems.

2.5 Messaging protocols

When any two applications want to communicate successfully, they have to come up with a set of rules they will follow. Multiple widely used protocols exist in messaging, so there is no need to reinvent the wheel. Message brokers and buses generally support more protocols, so each service connecting to them can use a different communication format. Red Hat's internal message bus supports only a subset of protocols that the ActiveMQ broker offers[2], namely OpenWire, Stomp v1.1, AMQP v1.0. These wire protocols define the format of the data passed over the network.

AMQP, which stands for *Advanced Message Queuing Protocol*, is a binary protocol emphasizing reliability and interoperability, as it was initially designed for use in the finance sector. It has multiple versions; the most popular ones are 0-9-1 and 1.0. The latter was standardized under International Standards Organization as ISO/IEC 19464:2014. Even though one might suppose that version 1.0 would be just a minor update, these two are entirely different protocols[3].

OpenWire is a default protocol of ActiveMQ. It implements Java Messaging Service API and is similar to AMQP, as it is also a binary protocol designed for high-performance messaging.

Simple (or Streaming) Text Orientated Messaging Protocol (STOMP) is a text protocol that aims to be easy to implement, at least for the client. As opposed to AMQP, it supports only a small subset of typical message operations.

While it was initially expected to use AMQP protocol for communication between the client and the message bus, STOMP turned out to be a better idea, primarily because of the better library support, as discussed in Section 5.5.1. Therefore, the following section will describe STOMP communication in more detail.

[2] The complete list can be found at https://activemq.apache.org/connectivity.

[3] https://www.rabbitmq.com/specification.html

2.6 Simple (or Streaming) Text Orientated Messaging Protocol

This section contains the most important aspects of the STOMP based on the STOMP specification [8]. It discusses the STOMP version 1.1, because that is the version that ActiveMQ and Red Hat's internal message broker support.

The communication is based on an exchange of frames, which are usually encoded as UTF-8 text and should be sent via some reliable protocol such as TCP. Each frame consists of a command, optional headers, and an optional body. STOMP does not use the concept of queues, topics, or virtual topics (in this paragraph, altogether referred to as "queues" for better readability). Instead, the STOMP server is modeled as a set of destinations. The client then sends messages to these destinations using a SEND frame. While STOMP treats the destinations as some opaque string and does not care about their contents, queues have a strictly defined syntax. Thanks to this, message brokers that provide support for queues and STOMP (as ActiveMQ does) internally map destination strings to queues according to their syntax rules. STOMP also keeps the delivery semantics of messages to each server, which is another example of its simplicity.

The communication flow is shown in Figure 2.2. It describes a situation when the client is a subscriber and does not send any messages. The client initiates the connection by sending a CONNECT frame containing the protocol version it wants to communicate with (or, alternatively, a list of versions it supports) and its host name. The client can send login credentials if the server requires them. As a response, the server sends back the CONNECTED frame. If the server rejects the connection, it should use an ERROR frame instead to signal this to the client.

Once connected, the client has multiple options for what to do. It can send messages, receive messages, work with transactions, or disconnect. During the work on this thesis, I did not find any use case for transactions in my solution; therefore, they will not be discussed. If the client wants to listen for incoming messages, it sends a SUBSCRIBE frame to the broker. In its headers, it sends the destination from which it wants to receive (for example, a string that corresponds to some customer queue on a message broker), id header, and an acknowledgment header. Within the connection, each subscription has to be identified by a unique id header so that the server can match UNSUBSCRIBE, ACK, and NACK frames to it.

Acknowledgment mode controls when the message broker considers the message to be delivered and deletes it from its queues (or any other mechanism it uses). There are three acknowledgment modes. *auto* means that it considers the message to be delivered as soon as it sends it to the client. In *client* mode, the client has to send an ACK frame to the message broker once it receives the message. Otherwise, the message broker can redeliver the message (depending on how the message broker is configured), and they can pile up in its queues. Also, this mode treats it as cumulative acknowledgment, so any preceding message is also considered to be delivered. The last mode is *client-individual*, which behaves the same as the *client* mode, but ACKs are not cumulative, so every message has to be acknowledged individually. NACK frames can be used explicitly to say the message was not delivered successfully.

When successfully subscribed, the client receives the messages sent to the given destination as MESSAGE frames. Once it wants to stop receiving them, it sends an UNSUBSCRIBE frame. The client can disconnect anytime by closing the socket it communicates through with the server. In order to make sure all sent messages were delivered to the

server, the client can gracefully shut down using the DISCONNECT frame. The client waits to receive the RECEIPT frame as confirmation, and then it can close the socket.

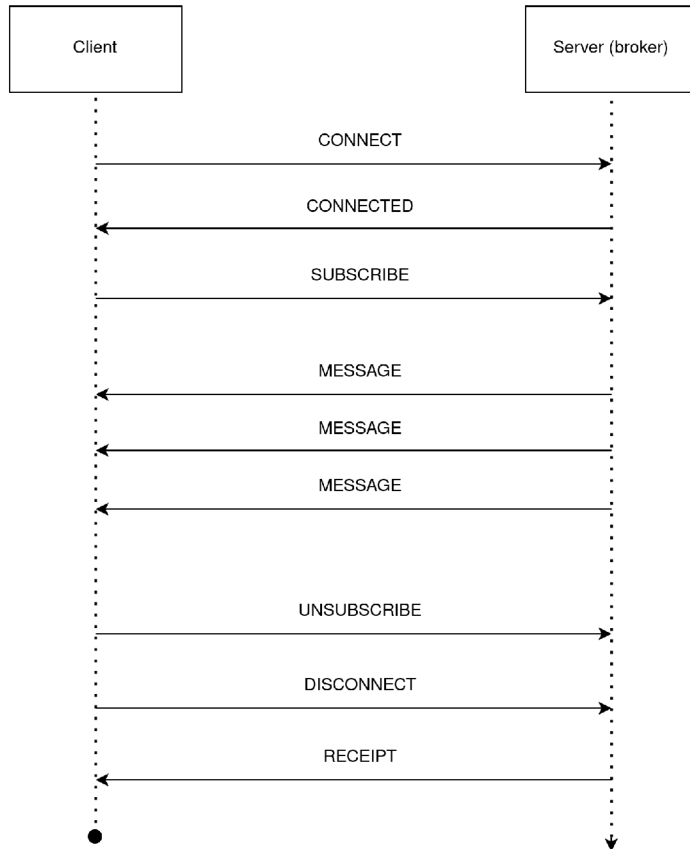

Figure 2.2: Communication between a message broker and a client that is connected as a subscriber. The acknowledgment mode is auto, so there are no ACK frames sent.

When the client wants to connect as a publisher to send messages, all it has to do is to connect to the server with CONNECT frame. Then it sends SEND frame to the server. Its headers contain the destination to which the message should be sent and the content type of the body, which then includes the message contents.

Chapter 3

Linux containers and orchestration

This chapter starts by explaining Linux containers, the motivation behind using them, and the tooling ecosystem. Then it digs into the concept of container orchestration and how container orchestration platforms can be used as execution environments.

3.1 Linux containers

Linux containers are technologies that enable the packaging of applications with all their dependencies, making them independent of the environment they are running on.

The main advantages of this technology include empowering agile development and making it easy to package the application's environment together with the application itself into a single bundle without worrying about the specifics of the given operating system it would be running on. Containers are light on resources compared to deployment on virtual machines. Engineers can use them on local workstations, allowing them to define a separate environment for individual projects, making dependency management less burden. It makes onboarding new people into projects faster because they spend less time setting up their environment. Given that files containing container definitions can be stored in a version control system (VCS) alongside the code, containers come in handy when adopting practices such as DevOps. Well-written applications can be scaled effortlessly using container orchestration, which will be described in more detail in the later sections.

Working with Linux containers is a straightforward process. At first, *a container image* is needed. It is a file (or files) containing the application and the environment, such as libraries, runtimes, and configuration files. Container images are built from *Dockerfiles*, text files with a human-readable definition of the image (these can be versioned in VCS). In most use cases, users build their images on top of the existing ones, so they do not have to build them entirely from scratch. Sometimes, when building the container image, starting with a fully-fledged container image such as *ubuntu* or *ubi9* containing all the essential Linux utilities and package managers like DNF or APT is better. Still, when optimizing for image sizes, engineers sometimes use more lightweight (sometimes called minimal) images such as *alpine* or *ubi9-minimal*, which are less feature-packed but still allow all the necessary tooling to be installed while taking fewer resources.

Once the container image is ready, the container engine (such as Docker or Podman) can be called to unpack the container image. It makes the API call to a Linux kernel, which starts a new process we refer to as *a container*. This container process is created with a `clone()` system call, as opposed to `exec()` or `fork()` utilities like Bash usually use.

For this reason, the container is just a Linux process with some extra isolation provided by kernel namespaces. The Linux container sees only the processes and filesystem created within the same namespace. There are no data structures in the Linux kernel representing containers other than the data structures representing the processes and the namespaces.

As container technologies became more popular and many new tools emerged, there has been an effort to standardize all the parts of the workflow. This standardization's primary objective has been ensuring that all those tools produce and consume the same artifacts, that swapping container tooling is possible without any significant issues, and that the investment into containerization is not vendor dependent. Currently, the widely used industry standards are set by Open Container Initiative, which contains the Runtime Specification, the Image Specification, and the Distribution Specification, which unify the most important parts of the container ecosystem. Therefore, when picking a new tool for working with containers, it is recommended to ensure it adheres to these specifications (the majority of mainstream tools do).

3.2 Container orchestration

"Container orchestration automates container provisioning, deployment, networking, scaling, availability, and lifecycle management" (IBM [4]). It can become beneficial when deploying and managing hundreds or thousands of containers. However, from the perspective of this thesis, I am primarily interested in using container orchestration platforms as an execution environment, enabling me to send it a container image and the job that should be executed, without needing to worry about the system resources, updating the systems and doing any other maintenance.

3.2.1 Kubernetes overview

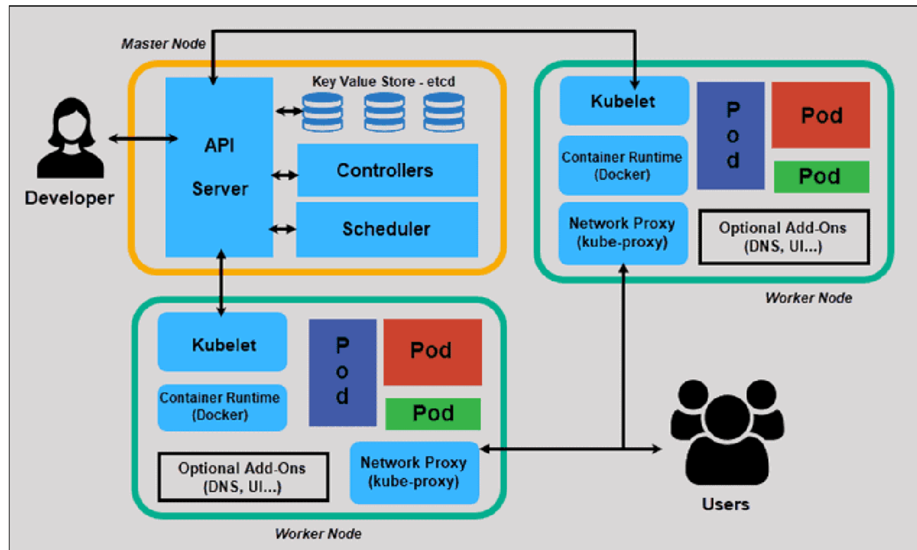

Figure 3.1: Kubernetes architecture. The control plane in the text refers to a master node in the diagram. Source: Kaplarevic [5].

One of the most popular orchestration platforms is Kubernetes, sometimes abbreviated K8s. The main component of its architecture is a Kubernetes cluster, which acts as an abstraction over one or multiple hosts capable of executing Linux containers. It contains *a control plane* and one or more *nodes*. Nodes are the workers executing the workload using the container runtime such as *CRI-O* or *containerd*. The basic idea is that users define declaratively how their cluster should look — which containers should be running on which nodes — and controllers running inside the control plane will do their best to match the actual state of the cluster with the defined one. The control plane also exposes Kubernetes API, as shown in Figure 3.1, enabling users to communicate with the cluster, and contains a key-value store called *etcd* with all the cluster definitions.

3.2.2 Kubernetes resources

The definition of nodes consists of individual Kubernetes resources, which can be defined in YAML or JSON. Kubernetes works with the smallest deployable units, which it calls Pods[1]. Pod is a group of one or more containers that share storage and networking. The following is a definition of a Pod in a YAML format, which creates one container on the OpenShift cluster that prints the text "Hello world!".

```yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: hello-world-pod
spec:
  restartPolicy: OnFailure
  containers:
  - name: hello-world
    image: ubi9
    command:
      - bash
      - '-c'
      - "echo 'Hello world!'"
```

In general, Pods are not created directly by the user and instead are created by workload resources. In the context of task execution, the most interesting are Jobs[2] and CronJobs[3]. While both of these resources create one or more Pods, the main advantage over defining Pods directly is that even if such created Pod fails in the middle of the execution, which sometimes happens because of the failing node, the Job resource starts a new one, making sure the job is actually finished. It lets the user define the exact behavior, such as how many times it tries to restart the Pod until it gives up or how many times it has to complete successfully for the whole Job execution to be successful.

Another advantage is the cleanup of the resources. Pods can be set to be deleted either immediately after execution or never. If the user wants to keep them around for some time (e.g., for debugging purposes), they have to set them never to delete and then schedule a CronJob to clean them periodically. On the other hand, Job definition provides

[1] Documentation: https://kubernetes.io/docs/concepts/workloads/pods/
[2] Documentation: https://kubernetes.io/docs/concepts/workloads/controllers/job/
[3] Documentation: https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/

a *ttlSecondsAfterFinished* field, which sets the number of seconds of how long should the Job resources be kept on the cluster after the Job completion.

To demonstrate, the following is a definition of a Job that will be kept on a cluster for 10 minutes after it finishes execution.

```yaml
---
apiVersion: batch/v1
kind: Job
metadata:
  name: hello-world-job
spec:
  ttlSecondsAfterFinished: 600
  template:
    spec:
      containers:
        - name: hello-world
          image: ubi9
          env:
            - name: USER
              value: Jack
          command:
            - bash
            - '-c'
            - 'echo "Hello $USER!"'
      restartPolicy: OnFailure
```

You may notice that the definition is very similar to the definition of a Pod. This Job also demonstrates how the define and use environment variables inside of a container. The resulting message printed on the specified container's standard output is "Hello Jack!". This will be helpful for injecting message data into the task definition later on. Also, note that the *apiVersion* field changed a bit. Different resources are part of different API groups. For a single resource, multiple versions can exist with different fields and structures, each accessible at different API versions. In the example above, the *v1* version of the *batch* group was used.

CronJob is similar to Job, but as the name suggests, it schedules the execution of a Job regularly according to a schedule written in a Cron format. For completeness, this is how to define a CronJob executed every Monday at 9 AM that says "Happy Monday!". The time is specified as Coordinated Universal Time (abbreviated as UTC), so it is necessary to adjust it to the local timezone. See Listing 1 for an illustration.

When the user defines these resources and creates them on the Kubernetes cluster, the control plane adds defaults for the fields the user did not define and adds a `status` field, which describes the object's current state.

```yaml
---
apiVersion: batch/v1
kind: CronJob
metadata:
  name: happy-monday-cron-job
spec:
  schedule: "0 9 * * 1"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: happy-monday
              image: ubi9
              command:
                - bash
                - '-c'
                - 'echo "Happy Monday!"'
          restartPolicy: OnFailure
```

Listing 1: An example of a CronJob definition.

3.3 OpenShift

OpenShift is an offering from Red Hat that builds on top of Kubernetes and as it was already mentioned, it is the target environment in which jobs should be executed. To explain it a bit better, I will borrow the explanation from Harrington [2], where the author uses an analogy with Linux operating systems. He compares Kubernetes to a kernel of distributed systems and says OpenShift is like the distribution, bundling Kubernetes and other components. OpenShift is then one from many bundles including Kubernetes that are available and is fine-tuned for specific use cases that the OpenShift project focuses on. Because it is a product sold to customers, Red Hat provides customer support and extensive testing.

Therefore, it is possible to use all the Kubernetes resources described above for workload execution. There might be some nuances or extra features that OpenShift offers, so it is always good to check its documentation.

3.4 Tekton

While Tekton is not directly used for container orchestration, it is a technology built on top of Kubernetes, and it was one of the considered tools to be used in a final solution. Therefore, I will spend this section explaining the basic concepts, so that its trade-offs can be discussed in Section 5.6.1.

Tekton is an open-source framework for creating CI/CD systems. It defines a set of Kubernetes Custom Resources that are used for building the pipelines. Tekton also supports the creation of event-driven pipelines that can be triggered as a response to committing a change to a Git repository or building a new container image.

To facilitate the event-driven pipeline execution, multiple Kubernetes objects have to be created, starting with an EventListener. It is a Kubernetes object provided by Tekton that listens for events at a specified port on a Kubernetes cluster. It runs in a dedicated Pod. It contains several triggers containing the filtering logic and the definition of a pipeline that should be executed if the filtering succeeds.

For every incoming request, which in our case would be a message triggering the task execution, the EventListener creates an HTTP request, sends it to a running Interceptor process, and waits for the response. This request contains the filtering logic and the message data. If multiple triggers are part of the EventListener definition, it creates a request for each of them. Interceptor is the component that does the payload's filtering, verification, and transformation and says if the pipeline should be triggered or not.[4]. For illustration, see Figure 3.2.

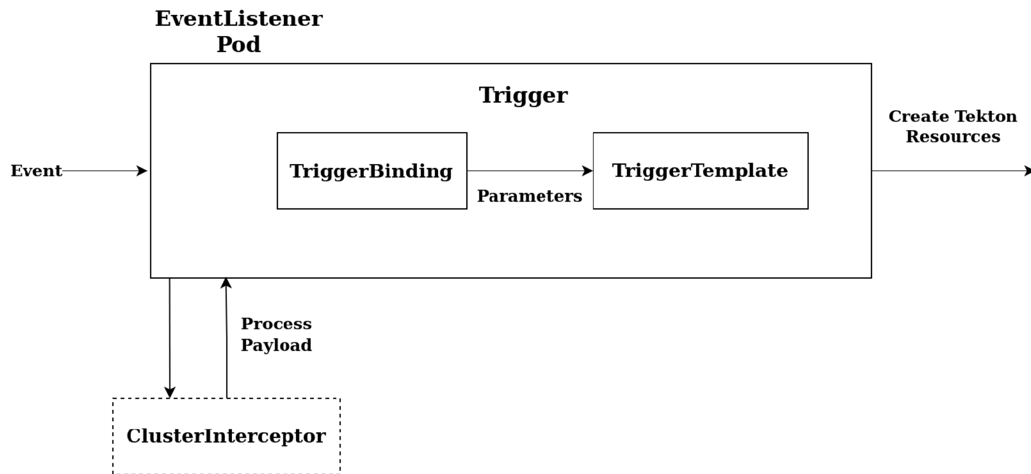

Figure 3.2: Diagram visualizing a Tekton Trigger and its communication with a ClusterInterceptor. Source: Tekton documentation [7].

Tekton Triggers ships with several Interceptors, most focusing on handling webhooks from major Git hosting providers. The most interesting one in the context of this thesis is a CEL Interceptor, which supports filtering and modifying payload using the Common Expression Language[5].

If the Interceptor says that the execution should continue, multiple Tekton objects are created, including the Pipeline object. This object can reference multiple Task objects because CI/CD pipelines usually contain multiple steps. In our case, only one Task object would be defined to execute the task we want to perform as a reaction to the message sent on a message bus.

[4]For more context, see Tekton Enhancement Proposal (TEP) 0026 at https://github.com/tektoncd/community/blob/main/teps/0026-interceptor-plugins.md and the pull request discussions at https://github.com/tektoncd/community/pull/229#discussion_r504698565.

[5]More information can be found here: https://github.com/google/cel-spec.

Chapter 4

Existing Solution

This chapter describes the current setup, explains the job definition format that is being used, and discusses the problems of using Jenkins for task execution.

4.1 Jenkins setup

The existing solution consists of a Jenkins instance that has the role of yet another application listening to a message bus, reacting to messages it subscribed to by executing tasks. The second role it has is the role of a controller. It periodically schedules task execution for those jobs that are triggered periodically.

Jenkins is an open-source automation server. It is commonly used for continuous integration and continuous delivery of products, which consists of building, testing, and deploying applications in an automated way. It is written in Java, and there is a huge amount of plugins that extend its capabilities.

The current setup of the Jenkins instance consists of one master controller running on top of the Java Runtime Environment. While it is possible to execute workloads on this controller node, it is considered bad practice because its environment has a lot of privileges. Therefore, it could lead to security issues. Jenkins is designed for distributed environments, so the master controller connects to Jenkins agents, each running inside a Virtual Machine. While there is an option to run these agents in containers, given that this is not the native way, it is hard to set up, its images are large, and there are many restrictions on what can be done with those containers. Having multiple agents also allows the use of different environments for different tasks and load-balancing of the workload among the execution nodes.

4.2 Job definitions format

There are multiple ways of configuring a Jenkins system. The simplest way is to click through the user interface (UI) to set up everything. Many users write Groovy scripts that invoke Jenkins API to automate this. While these scripts can do pretty much everything, it is necessary to know Jenkins internals well. This project uses a third approach of using Jenkins Configuration as a Code Plugin[1]. It provides an opinionated way to configure Jenkins using human-readable YAML files. It is relatively easy to use because it only requires users to translate the UI setup process they are comfortable with into code.

[1] The project website can be found at `https://www.jenkins.io/projects/jcasc/`.

Another plugin that is being used is Jenkins Job Builder[2]. It enables users to define jobs in human-readable formats YAML and JSON and then uses these definitions to configure a Jenkins instance. Again, it is a bit troubling there is no standard way of doing this.

The following is an example of a job definition (slightly modified to fit the page better)

```yaml
- job:
    name: rcm-errata-posterity
    node: rcm-compose
    project-style: freestyle
    defaults: global
    disabled: false
    display-name: "UMB QE/SHIPPED_LIVE Advisories"
    concurrent: true

    properties:
      - discard-old-builds
      - venv-manager

    triggers:
      - ci-trigger:
          override-topic: "Consumer.msg-bot.VirtualTopic.activity.status"
          no-squash: true

    parameters:
      - ci-message
      - snooze

    wrappers:
      - rcm-build-keytab

    builders:
      - shell:
          !include-raw-escape:
          - rcm-errata-posterity/run.sh

    publishers:
      - email:
          recipients: "spmm-release-jenkins-csb@redhat.com"
```

`node` definition at the top of the file specifies the machine on which the task should be executed. The task execution is managed by Jenkins agents, which are client Java processes running on the node. `properties` sets properties on Jenkins jobs, such as an option to automatically delete old job runs. `triggers` section defines the virtual topic whose messages trigger the job execution. To make sure that all the necessary secrets (such as Kerberos keytab file) are present on the worker, `wrappers` are used. `builders` define the actual job that should be executed. In this case, it is a shell script located in one of the subfolders that will be executed. `publishers` define what action should be taken once

[2]More information at https://jenkins-job-builder.readthedocs.io/en/latest/

the task finishes its execution. Here, it is used to send emails about failures to the specified email address.

4.3 Jenkins problems

One of the reasons why Jenkins is not sufficient anymore is its high maintenance costs associated with updating the application. Many plugins are used in the current setup, and updates usually cause conflicts between these plugins, causing existing task definitions to break. There are also frequent security issues requiring additional maintenance and upgrades. Also, it does not execute jobs in OpenShift, so there would have to be some work done anyways to enable this capability. And while it is technically possible to execute Jenkins agents in containers, it is more difficult to set up, and it limits some of the advantages that containers usually bring.

Therefore, the following chapters deal with the design of a solution that will supersede Jenkins in triggered and scheduled job execution and will be easier to configure, maintain, and be more flexible.

Chapter 5

Design

This chapter starts with the problem definition, lists the requirements that were collected during the analysis, and then describes the design that was implemented. It discusses the trade-offs that were made and the implications of these decisions.

5.1 Problem definition

From the assignment, the main objective of this thesis is to design and implement a lightweight event-based task framework for executing arbitrary workloads in OpenShift with the goal of migrating from the existing solution to the new one. To be more specific, there are two types of jobs that are currently being executed. There are *scheduled jobs* that are launched at certain times (for example daily) and *triggered jobs*, which are executed as a reaction to some message being sent on a message bus.

When looking at the performance requirements, I examined different job types separately. There are currently defined lower tens of scheduled jobs. Most of these jobs are executed daily. Some are scheduled more often, but always at most once an hour. For triggered jobs, based on the logs from the message bus, the number of messages that the current solution listens to is in the magnitude of lower thousands of messages per day. For one specific day, there were four messages per minute on average.

5.2 Scheduled jobs

For better readability and structure of this text, scheduled jobs will be discussed before diving into specifics of triggered jobs, even though it would make sense to start with the requirements of both types and then jump into the proposed solution.

At first, the idea was to make scheduled jobs part of the developed framework. However, I later realized that Kubernetes CronJobs resources could be used directly to rewrite those jobs, as described in Section 3.2.2. Therefore, it does not make any sense to reinvent the wheel or put some layer of abstraction above this. Also, note that even the Tekton community suggests using this Kubernetes resource rather than coming up with their own solution. For this reason, the rest of the design does not concern itself with scheduled job execution.

5.3 Requirements for triggered jobs

The following is the list of requirements and expectations about the system.

- The framework gets the messages from Red Hat's internal message bus described in Section 2.4.
- The solution must be able to subscribe and listen to a given virtual topic on the message bus.
- While most messages come from the message bus, users should be able to send messages into the solution to execute them manually. An example of such a workflow is engineers debugging their job definitions. They should be able to copy the message from the message bus logs and give it directly to the solution without needing to send a message through the message bus.
- Jobs should be defined in a format that can be tracked in a version control system. It is expected that job definitions will be code reviewed.
- For each job definition, users must specify a virtual topic. The messages sent to this virtual topic then trigger the job execution.
- Users can filter messages based on their contents, so the given job is launched only if the message contents meet user-defined criteria.
- The solution should execute its workloads in OpenShift, ideally being open for other execution environments to be added in the future.
- Defining a job should be user-friendly, and it should not take much time for the user to learn how to do it.
- There must be some mechanism for the injection of message data into the user-defined job.

Also, there are requirements that are important from the maintenance point of view, reliability, and testing.

- There should be extensive logging in place focused on making it easy to debug the application and to track messages traveling across the systems.
- There should be a reasonable amount of automated unit and functional tests.
- The solution should handle typical error cases that can happen, such as re-connection to the message bus if it is not available for a while.

5.4 The design overview

There are multiple ways the resulting application could be designed, and this section describes the approach I chose.

The resulting running solution consists of at least two processes. An API that receives messages and based on user-supplied configuration executes jobs on OpenShift. Furthermore, the client, which connects to the message bus via some messaging protocol, listens to messages and sends them to the API. Thanks to splitting the solution like this, engineers can create messages on the API manually, as required. See Figure 5.1 for illustration.

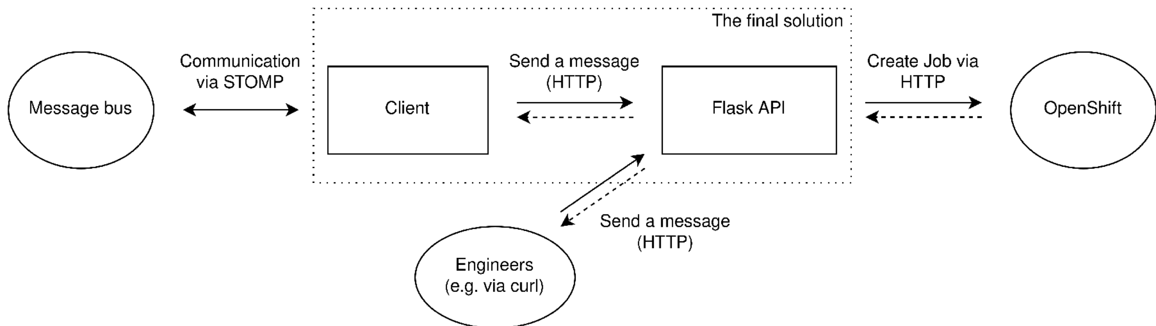

Figure 5.1: The design overview. The message bus and OpenShift are part of the existing infrastructure. The filled line marks an HTTP request, and the dashed line marks a response to this request.

5.5 The client

The client starts its execution by loading consumer queues, to which it should subscribe, from the user's definition configuration file. Because the API design allows it, this configuration file will be shared for both parts of the resulting solution. See Section 6.1 for more detail about the format of this file. Then, it subscribes to all the loaded queues using one of the messaging protocols that the message bus supports. From the point of view of the production version, it has to have access to these queues on the internal message bus. This authorization will be enabled by generating a certificate the client uses to authenticate. For local development, the client should support login with a username and password. On every message, the client validates that the message requires all the necessary fields and then sends it to the API.

The way it is currently implemented is that it waits for the response to arrive. Given that it is intended for the API and client to behave as one application, API processes the request quickly, and the performance requirements are relatively low, this is not a problem. However, if any of these statements stopped being true, it would be necessary to make sure that request-sending logic uses asynchronous requests. Alternatively, the client deployment could be horizontally scaled.

Lastly, Python was chosen as a technology in which the client was implemented because it is a high-level programming language perfectly suitable for writing smaller utilities.

5.5.1 Connecting to the message bus

As was mentioned in Section 2.4, the internal message bus supports communication via STOMP v1.1, AMQP v1.0, and OpenWire. In the local development environment, an instance of ActiveMQ is being launched. When downloading ActiveMQ and running it with out-of-the-box configuration, ports at which each of these protocols communicate can be found in a configuration file located at `apache-activemq-{version}/conf/activemq.xml`.

Choosing the appropriate protocol turned out to be more about picking the right library rather than about the features of the protocol itself. Regarding AMQP, two very actively supported libraries in the Python ecosystem provide a reasonable level of abstraction over protocol communication — *Pika* and *Kombu*. Both libraries are often used with RabbitMQ, a popular message broker. *Kombu* is internally used in Celery, a widely used distributed task queue. However, both of these libraries support only AMQP version 0-9-1, which is “a

completely different protocol"[1] than AMQP 1.0, and while there have been discussions[2] to bring AMQP 1.0 to *Kombu* for years, I do not think it will be implemented anytime soon.

ActiveMQ's website recommends using *stomp.py*[3] as a Python client library, which was eventually used. It supports STOMP protocol versions 1.0, 1.1, and 1.2, therefore being compatible with ActiveMQ. It is actively supported and easy to use. Its GitHub repository and the corresponding PyPI package also bundle a command line client, which can be used for testing purposes.

The recommended way to connect to the internal message bus is via a server-side load balancer it provides and then to subscribe to virtual topics via consumer queues. Any other way complicates the setup, and the client risks not getting all the messages. Therefore, there is not much setup that would be necessary to do regarding the locally used ActiveMQ instance.

5.5.2 The message format

While producers can send messages in different formats, it is necessary to set some common input message format expectations that the client has including what fields are required for a message to have to be correctly processed. To demonstrate, see the following message that was taken from the logs. Note that some fields were deleted for demonstration purposes and some obfuscated so that I am not unintentionally leaking any sensitive data.

```json
{
  "headers": {
    "JMSXUserID": "some-user",
    "amq6100_destination": "queue://Consumer.some.consumer.queue.>",
    "content-length": "300",
    "content-type": "text/plain; charset=UTF-8",
    "correlation-id": "323f2423-72ef-4496-8821-a4f34f269005",
    "message-id": "SomeMessageId",
    "subscription": "/queue/SomeVirtualTopic",
    "timestamp": "1679564400799",
    "when": "2023-03-23 09:39:57 UTC",
    "who": "user@redhat.com"
  },
  "msg": {
    "data": "some-data",
    "when": "2023-03-23 09:39:57 UTC",
    "who": "user@redhat.com"
  },
  "timestamp": 1679564400.0
}
```

`headers` contain metadata about the message and `msg` contains the message payload. Other top-level keys are present, such as `timestamp`. From headers, the most important ones are `amq6100_destination` and `correlation-id`. `amq6100_destination` specifies the consumer queue on which the message was received. The prefix marks AMQ-6100 Jira

[1] https://www.rabbitmq.com/specification.html
[2] https://github.com/celery/kombu/issues/548
[3] https://github.com/jasonrbriggs/stomp.py

ticket[4] that suggests that the destination should contain the consumer queue and not the virtual topic. For virtual topics, there are other fields. Note that the STOMP library does not care about the specifics, as it does not know anything about concepts such as topics, but it has to be given the string in a consumer queue format to subscribe to it successfully.

`correlation-id` is the identifier of the message. While other identifiers are present, this one is used for referring to the message in logs.

5.5.3 The message loss

Reacting to failures of the message bus and lost messages is another design consideration. As mentioned in 2.4, the message bus is considered a highly available service that backs its messages inside a database. Message loss is often caused by producers and consumers, who are not as critical as the message bus and do not implement advanced strategies for mitigating this problem.

A situation that can lead to message loss is the client's failure. It does not matter if it fails because of some internal error or because it will be shut down intentionally, but in both cases, some messages might be present on the system. It is common for subscribers to implement message caching. The reasons are to increase performance and lower requirements for network resources so that the subscriber does not communicate with the message bus as often. Therefore, there can be more than one message buffered on the client at any moment.

One of the possible strategies for mitigating message loss problems for the client is being smart about acknowledgments. The client can subscribe to the message bus in *client-individual* acknowledgment mode. Once it receives the message, it sends the message to the API and waits for the response. Only after getting a response that the job was successfully launched it sends an ACK frame to the message broker. The alternative is to have its own database, which would persist the message until the corresponding task is executed. In the resulting implementation, the *client-individual* acknowledgment tactic is used.

5.5.4 Another considerations

The client has to be able to deal with the disconnection of both the API and the message bus, even though it is unlikely for the latter one to be unavailable. When this happens, connecting to these services should not require human interaction once they are back online.

The way this was eventually implemented is that in case the client cannot send the message, it waits for a given time interval and then tries again. Similarly, when the message bus disconnects, it waits for some time and then tries to connect again and subscribe to all the virtual topics it should listen to. In both cases, it prints a log message that there was some communication problem. Therefore, when using log monitoring tools, these log messages can be filtered, and alerts can be set.

5.6 The API

The API is part of the solution that receives a message, does some filtering, and, based on the result, triggers or not triggers a job execution. It expects messages to be in the same format as the client receives them.

[4] https://issues.apache.org/jira/browse/AMQ-6100

When designing the API, the two most promising technologies that were considered were either leveraging Tekton or writing the application using Flask. In the end, I implemented a custom API using Flask. The following section describes the trade-offs between them in more detail because this will help understand all the design considerations.

5.6.1 Tekton vs. a custom application

If Tekton was used for the API part, users would have to write a lot of boilerplate code to define a simple job and to pass parameters into it because it would have to be written as a pipeline with one job[5]. Of course, they have to learn the format for writing jobs even in the custom solution. Therefore, the configuration file format was designed to mirror as much as possible OpenShift Job definition format so that it is as intuitive as possible, making it easy to migrate the definitions in both directions.

With Tekton, Common Expression Language (CEL) would have to be used for filtering which jobs to execute. Alternatively, a new Interceptor could be implemented to support any other language, but it does not seem like a good idea because in that case, it would probably be better to learn CEL. Implementing an API from scratch enables one to choose other approaches to filtering, such as letting engineers define a Python function returning a Boolean value acting as a filter. Therefore, users do not have to learn a new language because most of them have at least some experience with Python. However, it does not mean that this would not have any downsides. Empowering users to use a Turing complete language means they can write filters that could crash the whole application or execute malicious code, taking control over the machine. This is one of the reasons to use more restrictive languages such as CEL. How these problems are tackled is discussed in Chapter 6.4.

Tekton is quite a complex tool, requiring a solid understanding of OpenShift. A lot of people use it just to write their pipeline without worrying much about what is happening under the hood. However, given that it would be used for a non-typical use case, problems might arise, and troubleshooting could require more profound knowledge. Naturally, even the custom implementation has some inherent complexity, but it is arguably easier to read Python code.

Lastly, implementing custom API grants more control over the service behavior, tailoring it to current needs and can enable extending it with support for other execution environments in the future.

The advantage of using Tekton is that it provides a graphical user interface (GUI). This is a nice-to-have feature rather than a requirement, as the whole application is designed to be used from the terminal. Given that the resulting jobs are executed in OpenShift, they can be inspected from the OpenShfit web console, so part of the workflow would have GUI support even with the Flask design.

Also, in the enterprise context, there might be some instances of Tekton already provisioned by other teams, so it could lead to less maintenance work than deploying one's own instance.

Weighting all these trade-offs, I decided to move forward with implementing a custom API. Note that no matter what approach I would pick, the client would have to be implemented because Tekton does not have any plugin that would connect it directly to

[5] Maybe, a Custom Resource could be defined to ease the burden, but I did not get that far in prototyping it.

the message bus. Implementing such a plugin would be more challenging than writing a separate utility.

5.6.2 The interface

The interface is designed as REST API that works with JSON as a data transfer format for both getting the requests and sending responses. This is because the goal is to use a format that is a good middle ground for both machine and human communication. The alternatives considered were HTML and plaintext, but these are not suitable for machine communication. Together with the JSON payload, an error code is returned to mark the success or failure of the request, where codes in the range of 400 to 499 signal problems with the client data and codes in the range of 500 to 599 signal problems with the server.

If the job was successfully launched, the response indicates which job was triggered and is returned together with response code 200. Note that in both following examples, the `description` line is wrapped and formatted to fit the page, even though, according to the specification, this is not a valid JSON.

```
{
    "description": "Message triggered execution of the following jobs:
                     ['hello-job'].",
    "name": "OK"
}
```

In case any error occurs, `name` field will contain the exception. In the case of the following exception, error code 400 is returned.

```
{
    "description": "Message headers have to contain 'correlation-id' key.
                    Message: {'headers': {}, 'msg': {'say_hello': 'true'}}",
    "name": "Validation Error"
}
```

API has one endpoint to create a request, one to check the health status of the application, and the root one that returns the URL to the repository, which contains more information about the service.

5.6.3 Flask

Flask is a widely popular micro framework for developing web applications written, and it is excellent for writing REST APIs. It is written in Python. Because it is a micro framework, it contains only the core functionality, and support for things like object-relational mapping or authentication is provided via extensions. It is also WSGI compliant. WSGI stands for Web Server Gateway Interface, and it is a Python standard that specifies an interface between web servers and Python web applications and application servers. This means that instead of being forced into using a specific web server, users writing their applications in Flask can pick from a plethora of WSGI-compliant web servers, choosing the one that matches their needs.

One of the features I wanted to implement was to be able to reload the job definitions in a running application. Unfortunately, this is not possible because all the user configuration,

including job definitions, is loaded on startup into the `config` attribute of the Flask object. And, according to the Flask's documentation [6], "All application setup must be completed before you start serving your application and handling requests. This is because WSGI servers divide work between multiple workers or can be distributed across multiple machines. If the configuration changed in one worker, there is no way for Flask to ensure consistency between other workers."

On the upside, Flask has a feature of running the app in debug mode, which makes it reload every time it notices there are some changes in the code base. This feature is used when launching the application by `make run`, which executes the `flask` command with debug option like this: `flask -debug run`.

5.6.4 Task execution

To execute the task, a Kubernetes Job object is generated from the user job definition and then sent to the OpenShift API. The main reason behind generating it is to let users pass message payload to the job itself. Another reason is to be able to define some shared characteristics for all jobs, such as cleanup time, enabling users to focus on defining the tasks themselves. Otherwise, users could create a Job for each task they want to execute, and no generation would be needed.

The other considered approach was to use OpenShift templates that would be created on application startup and uploaded to the cluster. Then every time a job was triggered, a new Job would be created out of those templates, passing the message payload into them. However, I did not see much value in creating those templates compared to creating a new resource every time because the template would have to be also generated, so it would needlessly overcomplicate it. Furthermore, it would limit the library choices because not all libraries support all the OpenShift command line client features.

5.6.5 Connecting to OpenShift cluster

Three ways of connecting to the OpenShift cluster are via the command line client, REST API, and client library. For this application, I used *openshift-restclient-python* library, because it can be called directly from Python code, and no additional software needs to be installed, so the task framework is independent of the environment.

Other libraries are available, such as *openshift-client-python* (both are under *openshift* namespace on GitHub). Still, the downside of this library is that it requires OpenShift command line tools to be installed in the environment in which it is running. The upside is that because it uses the OpenShift command line tool under the hood, it has more capabilities than the library I chose, which supports only basic operations on OpenShift resources. However, basic operations are enough for my use case, so I decided to go with a smaller runtime environment size.

When connecting to the OpenShift cluster, the user has to log in first and choose the project. Project is a Kubernetes namespace with some extra annotations and provides a way to split resources in a single cluster. Each user may have different access rights and permissions for different projects. It also enables setting limits on resource consumption on a project basis.

5.6.6 Authentication

Authentication of the API will not happen in the application itself. While some of the Flask extensions are available, it would mean adding a database to the design but mostly, enterprises usually require their applications to have Single Sign On (SSO) capabilities, which could not be simply implemented on this layer. Therefore, authentication of the API will be handled during deployment, either using Kerberos on the web server or OAuth proxy when deployed to OpenShift.

Chapter 6

Implementation

The following chapter discusses the most interesting and important aspects of the implementation.

6.1 Configuration files

There are two configuration files that both parts of the solution consume. The first is a file containing job definitions, which the user supplies. The second is `common/config.py`, which mainly sets the basic configuration from a deployment point of view and specifies at which file path the job definition file will be expected.

For the job definition file, the YAML format is used. It was chosen mainly for its readability and popularity. The other considered format was JSON, and while both are human-readable formats, JSON focuses more on being easy to parse and generate, making it slightly harder to write and read than YAML. Version 1.2 of YAML specification was released to make YAML a superset of JSON, which means conversion between the two is possible. For conversion from JSON to YAML, this is true under the condition that JSON files do not have duplicate keys[1].

In the context of the Python ecosystem, this leads to two popular ways of defining, validating, and parsing a YAML file and its schema. The first one is to convert the YAML file to JSON and use a library such as *jsonschema* to validate it against the defined schema, which can also be written in YAML and then in code converted to JSON. This is great for migrating between formats and for the flexibility it enables.

The second approach is to use a YAML library. In the final solution, *StrictYAML* is used. There are some other popular Python YAML libraries; however, not all of them support validation. The main advantage over using *jsonschema* is the error message format. When converting YAML to JSON, the information about line numbering is lost. This can make searching for a typo in larger files harder because the only way to locate a mistake is by using the order of keys. Also, duplicate keys are not checked because they are valid according to the JSON specification.

Another aspect of library choice is arbitrary code execution. One of the considered libraries, *PyYAML*, converts YAML documents directly to a Python object. Therefore, when using its `load` function, it is necessary to make sure that the document is coming from a trusted source because the attacker could construct a Python object in a way that a mali-

[1] More on the topic of converting between the two can be found at http://yaml.org/spec/1.2-old/spec.html#id2759572.

cious function would be executed.[2] That is why this library also implements a `load_safe` function that supports constructing only simple Python objects, such as integers or lists. This approach is similar to the *StrictYAML*, which limits the objects used to represent the YAML document by default. More on arbitrary code execution can be found in Section 6.4.

An important feature or downside of *the StrictYAML* library, depending on the point of view, is that it parses only a restricted subset of YAML. One of the reasons behind this decision was to avoid some problematic YAML constructs that can lead to unexpected behavior. Fortunately, in the case of this project, none of these unsupported features is missed.

The job configuration file is validated against the schema when the YAML file is loaded and internally converted into JSON. This schema is written using *StrictYAML* and can be found in `common/schema_job_definitions.py`.

An example of a job configuration file can be found at `tests/job-definitions.yaml`. Here, only the excerpt necessary to describe the format of the file is shown.

```yaml
environments:
  openshift:
    name: openshift-prod
    namespace: khiscahaw
jobs:
  - name: countdown
    environment: openshift-prod
    triggers:
      - topic: "/topic/countdown"
        filter: message["execute"] == "true"
    image: ubi9
    env:
      - name: MESSAGE
        value: "Liftoff!"
      - name: INITIAL_VALUE
        eval: message['initial-value']
    bash: |
      echo 'Starting countdown'
      i="${INITIAL_VALUE}"
      while [ $i -ne 0 ]
      do
        echo "$i"
        i=$(( $i - 1 ))
      done
      echo "${MESSAGE}"
```

At the beginning of the file, execution environments are defined. The only currently supported environment type is `openshift`, which can specify only one environment with one namespace. It could be defined more straightforwardly, but this is meant to be open for the addition of other execution environments. Each environment has some name by which it is referred from the job definition. In this case, it is called `openshift-prod`. Note that

[2] More on this in its documentation: https://pyyaml.org/wiki/PyYAMLDocumentation

the application expects `.kube/config` file to be present on the system, through which it is possible to connect to the OpenShift cluster, so no authentication details are required here.

The following section consists of an array of job definitions. Each job has a name, which determines how it will be referred to in logs and response messages from the API. Then, there is a `triggers` section. There are two types of triggers – `topic`, which is mandatory, and `filter`. `topic` determines messages to which topics trigger the job execution. It is also used by the client so that it knows to which topic to subscribe. `filter` enables the user to write Python expressions that are evaluated to some Boolean values. If it evaluates to `False`, then the message is not executed. Users can use a `message` object to inject a message payload.

Because this job uses an OpenShift environment, it defines an `image`, in which it will be executed. In order to inject variables into the runtime environment, `env` section is used. `name` denotes the environment variable name, and to pass a string, `value` is used. This mimics the way it is done in OpenShift definitions. In order to inject some message data, `eval` key can be used. Again, `message` object contains the message payload.

Lastly, the job definition contains a `bash` key. The script that the user defines here is then executed on the OpenShift cluster using `bash -c <script>` command.

6.2 Generation of Job resources

A new Job resource is generated and sent to the OpenShift API every time a job is triggered. The generation of the resource happens in `_generate_k8_job_resource` method on `OpenShiftJob` object. At first, it loads the basic Job template as a string from a file located at `api/jobs/openshift_job_template.yaml`, it does some basic substitutions and then converts it to a Python dictionary. Then, metadata, script, and environment variables are injected. The library call that creates the resource on the OpenShift cluster accepts the Job object as a dictionary, so no further transformation is needed.

The resulting resource is shown in Listing 2. It is sent to the API in JSON, but here it is shown in YAML, which is the format OpenShift web console prefers to show the resources. The most interesting things to notice here are value injection and labeling. Both the Job object and Pod object, which will be created from the `template` definition, are labeled with `correlation-id` and `job`. Thanks to this, anyone viewing executed tasks in the OpenShift web console can simply filter jobs related to a given message or a job definition. The reason behind labeling it with `job` is that each Job object must have a unique name, which is defined under `metadata` field. Therefore, a random suffix is appended to the job name. This leads to job names not being suitable for filtering, and therefore `job` is added to `labels`.

Comparing the generated Job resource with the original job definition, the resource substituted `value: '7'` for `eval: message['initial-value']`, effectively injecting message data into the Job definition. The data is extracted from the message using `eval` function, which is discussed in more depth in Section 6.4.

```yaml
---
apiVersion: batch/v1
kind: Job
metadata:
  name: countdown-wstb
  labels:
    correlation-id: cf56f921-e7b9-4246-a064-71ed5a99bb3b
    job: countdown
spec:
  ttlSecondsAfterFinished: 600
  backoffLimit: 3
  template:
    metadata:
      name: countdown-wstb
      labels:
        correlation-id: cf56f921-e7b9-4246-a064-71ed5a99bb3b
        job: countdown
    spec:
      containers:
      - image: ubi9
        name: countdown-wstb
        command:
        - bash
        - "-c"
        - |
          echo 'Starting countdown'
          i="${INITIAL_VALUE}"
          while [ $i -ne 0 ]
          do
            echo "$i"
            i=$(( $i - 1 ))
          done
          echo "${MESSAGE}"
        env:
        - name: MESSAGE
          value: Liftoff!
        - name: INITIAL_VALUE
          value: '7'
      restartPolicy: OnFailure
```

Listing 2: An example of generated Kubernetes resource that is then created on the OpenShift cluster.

6.3 Logging

In a microservice architecture, extensive logging has to be in place so that the debugging of the system can be effective. For services communicating with messages, it is essential for each logging print to reference the corresponding message. Only then it is possible to filter through logs effectively. In the case of Red Hat's message bus, the identifier used is a `correlation-id` header, which is unique for every message.

Another best practice that should be followed is logging messages at the processing boundaries. Logs should contain information about when the service receives a message, sends it, or some significant execution step is finished. This way, messages can be tracked as they travel through the infrastructure. Furthermore, any errors should be logged as well. The error data can then be aggregated, and alerts can be set up to notify when the application does not function properly.

My solution implements these best practices. Apart from the basic logging, both client and API log the message on receive, and then they validate the message headers. Once they make sure that those headers contain a `correlation-id`, they create an instance of `MessageLoggingAdapter`. This adapter is a subclass of `logging.LoggerAdapter` and its constructor expects a `correlation-id` as an argument. The most important part of the adapter is its process method, which can be seen in the code snippet below. Every time message is logged through this adapter, the `process` method is called, injecting `correlation-id` into the message. Therefore, the format of the logging messages is unified, and as long as the programmer is using the adapter, they do not have to think about message identifiers when logging. Note that at first, the message has to be validated that it contains the `correlation-id`, and only after that can it be logged with the `MessageLoggingAdapter`.

```python
class MessageLoggingAdapter(logging.LoggerAdapter):
    """
    Any application log related to a message has to contain its
    correlation-id.
    """

    def process(self, msg, kwargs):
        return (
            f"('correlation-id': {self.extra['correlation-id']}) {msg}",
            kwargs
        )
```

The `process` method keeps the formatting that is set up on the application startup of both client and API and only changes the message part. The logging format and level are specified in variables defined in the `common/config.py` file. Also note that when working with Flask, the logger has to be accessed through the Flask object, as shown in the code snippet below.

```python
app = Flask(__name__)
app.logger.setLevel(config.LOG_LEVEL)
```

6.4 Arbitrary code execution

One of the key considerations is how to enable users to specify filters for jobs they define. Among the most user-friendly approaches is to have filters defined as Python expressions or functions returning Boolean values. To do that, Python offers `exec` and `eval` functions for code execution. `exec` function executes the statements and returns `None`, and therefore, the results of the execution are only seen as side effects. In contrast, `eval` returns the result of the executed expression. Because we are interested in the results of the expressions, `eval` is used. However, there are some problems to be aware of, namely:

- `exec/eval` introduces a vector of attack for code injection.
- Poorly written code can cause a segmentation fault of the whole application.
- It is slower than writing if-else statements because it leads to the recompilation of relatively small pieces of code during the execution of the program.

While there is some deny-listing one can do to limit what functions can be executed inside the `eval` calls, there is always some workaround[3]. Considering the use case of my framework, users can not only write filters but also define scripts that are launched if the message passes the filters. These scripts are executed with similar or even more permissions than the deployed task execution framework, so if they wanted to cause some damage intentionally, they could encode the evil logic into scripts in job definitions. Therefore, the primary protection is to have all job definitions code reviewed.

Also, if the user is not intentionally trying, a segmentation fault does not happen easily in Python. Users may write code that results in throwing an exception, but this exception is caught in the code that calls `eval`.

So, the main disadvantages are that the code executed by `eval` will recompile every time[4], worse testability of the written code, and it may make the application harder to debug.

People criticizing the use of `eval` say that it is usually a sign of poor design and there is usually a better way to do it, but it seems reasonable in this case. The only alternative is using a different language for filtering, such as using CEL, but this would mean losing one of the advantages of this design.

6.5 Flask and error handling

The simplest way to work with a Flask application is to create `flask.Flask`, which acts as its central object. Once created, it is used for registering view functions, URL rules, template configuration, and much more.[5] However, in order to make the application more extensible in the future, *blueprints* are used. A `Blueprint` object works similarly to `Flask` object, and it is used for extending an application. It is great for splitting large applications into components or registering blueprints on URL prefixes.

In the final application, blueprint `api_v1.py` is created and then registered on Flask application object using construct

[3]This blog post describes the dangers in more depth: https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html.

[4]But if there were any performance problems, this code can be compiled beforehand and then reused, see https://lucumr.pocoo.org/2011/2/1/exec-in-python/.

[5]Documentation is at https://flask.palletsprojects.com/en/2.2.x/api/#flask.Flask.

```
app = Flask(__name__)
app.register_blueprint(api_v1.py)
```

Anytime during the execution of the Flask application, an exception may be raised. To override the default behavior, an error handler can be registered[6]. An error handler is a function that returns a response when a certain error is raised. It is passed an instance of the error that is being handled. The following is a code snippet from `api/api_v1.py` that shows how is the error handling done in a final solution. Notice that `ResponseBaseError` is passed into the handler. It is a class that other custom errors should extend so that there is only one error handler for all of them.

```
@api_v1.errorhandler(ResponseBaseError)
def return_errors_as_json(e):
    response = {
        "name": e.name,
        "description": e.description,
    }
    return response, e.code
```

This error handler is registered on `api_v1` blueprint. While handlers registered on the take precedence over those registered globally, not all requests reach the blueprints. For example, 404 routing errors happen before the blueprint can be determined. Therefore, there is an additional error handler defined in `wsgi.py` that takes care of such errors.

Another cool thing about Flask framework is that it offers a testing client. The following snippet show how it is initiated and then it is used in tests as a fixture. The `app` here is a `flask.Flask` object.

```
@pytest.fixture
def client(app):
    return app.test_client()
```

The client instance can then be used to directly create requests in unit tests like this

```
resp = client.get("/status")
```

6.6 stomp.py library

Working with *stomp.py* library is relatively straightforward. At first, a connection object is created. Because message bus supports Stomp v1.1, `stomp.Connection11` is used. As an argument, the constructor takes a list of URL endpoints. When connecting, the list is traversed until a connection can be made.

Then, a connection listener is registered on the connection object. A connection listener is a subclass of `ConnectionListnener` class that implements many helpful methods such as `on_connected()`, `on_message()` or `on_error()`. Multiple connection listeners can be registered, each with a different purpose, such as logging or statistics collection. The client uses only one connection listener that takes care of everything, including logging, sending messages to the API, and reconnecting.

[6] Flask documentation about error handling: https://flask.palletsprojects.com/en/2.2.x/errorhandling/.

Once registered, the client connects and subscribes to the queues it loaded from the job definitions file and waits for incoming messages using `Event().wait()` statement, which blocks the current thread.

Chapter 7

Testing and Evaluation

This chapter describes the approach to testing and the technologies that were used. It provides some numbers on the system's performance and discusses scaling options. Lastly, it compares the solution against the current setup.

7.1 Unit testing and continuous integration

Unit testing aims to split the code base into individual units and ensure these units work independently. Each unit test is executed in an isolated environment, so integration with other components does not affect its result. Apart from code validation, it also has a documenting function that strictly defines the contract for a given piece of code.

For this project, *pytest* is used as a unit-testing framework. Tests are located in `tests` directory, which is split into subdirectories that group tests for each top-level module. It also contains the `job_definitions.yaml` file with a sample of job definitions used within the tests. The test suite focuses primarily on testing the application logic, any external dependencies, such as OpenShift, are mocked. This means unit tests can be executed without worrying about the environment setup. For mocking, `MagicMock` object is used extensively by invoking its constructor directly or using a `@mock.patch` decorator. This class is part of *unittest* testing framework, which is imported as a dependency. From every unit test run launched by `make test`, a code coverage report is generated using *pytest-cov* module, which configuration can be found in `.coveragerc` file.

A continuous integration pipeline is triggered on every commit in order to minimize the number of regressions introduced into the code, speed up the process of merging new changes, and follow the best practices of modern software development. It is defined in a `.gitlab-ci.yml` file and uses GitLab CI/CD tool. The pipeline consists of one stage with two jobs. Both of them use a `ubi9-minimal` container image with all Python dependencies installed as an execution environment. The first job executes the unit tests described above using `make unit-test` command. The second job is doing static analysis of the code. It uses *flake8* linting tool to catch programming and stylistic errors. The second utility it launches is *black*. It is used for code formatting to increase consistency and readability and to reduce diffs when committing changes into the version control system.

End-to-end testing has to be done manually. *sender.py* utility is provided, which sends two messages to the ActiveMQ message broker upon launching. These messages should trigger task execution, and as a result, two Jobs should be executed on an OpenShift cluster. Further manual testing was carried out to ensure that both client and API can

handle the failure of components they communicate with, such as the OpenShift cluster, message bus, or any of the two mentioned parts of the application. All the testing was carried out with applications running on Red Hat Enterprise Linux 8.7 that were connecting to Openshift with Kubernetes 1.24.11 and Kustomize 4.5.4. Messages were delivered by ActiveMQ version 5.18.1.

Performance was also tested manually by examining the logs. However, the solution has not yet been deployed so the numbers may vary for the production deployment. The solution was tested by launching the development environment, which means an instance of ActiveMQ, client, and API was running locally on my laptop. The Flask API was running on a debug WSGI server that is part of the package and was making requests to the OpenShift cluster, which was running somewhere in the cloud. *sender.py* utility was configured to launch 10 `countdown` jobs from `tests/job_definitions.yaml`. The test was performed three times. On average, it took 0.198 seconds to execute the whole logic of launching a Job and acknowledging it by the client. This meets the requirements by a sizeable margin. However, analyzing logs for one of these runs in more depth, the execution spends 93% of time in a `openshift_job._create_kubernetes_job` function.

This might be caused by a long round trip time caused by the distance between my machine and the data center hosting the OpenShift cluster. Therefore, it is recommended to deploy the solution on the same OpenShift cluster that will execute the tasks or at least in the same data center. Also, the library creating the resource could be analyzed for speed, but from my quick tests, performing a health check with `curl` command takes longer than creating the resource, so I think it might be pretty efficient with the number of requests it needs (`curl` starts by performing a TLS handshake). To increase the system's performance, the client could make requests asynchronously. The same could be said about the API, but Flask is a synchronous web framework, so there are some limitations when it comes to asynchronous programming[1]. Within a single request, multiple asynchronous coroutines can be launched. Nevertheless, when there are many requests, the worker still executes them one by one, which does not bring any benefit in this case. Therefore, the API should be scaled horizontally.

7.2 Comparison with similar tools

There is no industry standard around the task execution that the solution created as part of this thesis could be directly compared to. Close to it is the Jenkins setup described in Chapter 4. While it provides many features out of the box, it must still be heavily customized to provide the needed functionality. This is done using plugins that enable support for configuring the whole system and job definitions in code, which I would expect to be part of the standard Jenkins installation. It also needs a plugin that handles communication with the message bus. In the end, it might be argued that from the point of view of customization, the final solution of this thesis is not much different from the Jenkins setup. Both solutions use some off-the-shelf software as an execution environment. The current solution uses Jenkins agents to execute jobs in virtual machines, while the new one uses OpenShift, which launches the jobs as containers. This thesis comes up with a client that connects to the message bus, building on top of the *stomp.py* library. Again, this is quite similar to customizing a plugin, which Red Hat must maintain.

[1] Flask documentation on async is at https://flask.palletsprojects.com/en/2.2.x/async-await/

However, the solution presented in this thesis is still more customized. The client relies on the message bus to store unprocessed messages, eliminating the problems of missing messages caused by the failing client. It is open for extension of other execution environments (e.g., Ansible). It is more flexible to adapt to future needs while not introducing any extra complexity caused by bending one-size-fits-all solutions. While these general tools offer a wide range of features that might support a particular use case, such a workflow may not be aligned with the central vision of product teams and, therefore, might be more challenging to set up (see Section 5.6.1 for more discussion of how Tekton fits into this).

Similarly to much off-the-shelf software, Jenkins aims to support various use cases and therefore has many features that might not be strictly necessary to meet the requirements but can be helpful. In the case of the existing solution, these nice-to-have features include a graphical user interface or alert email integration. OpenShift also provides a graphical user interface, but a large part of the workflow is accessible only via terminal and API calls. Buying a general solution is usually cheaper than implementing all the functionality if no or only a little customization is needed. However, hidden costs might be associated with it, which may come in the form of recurring subscription renewals, upgrades, and mentioned customizations. Sometimes, companies start with off-the-shelf software to quickly bring the product to the market. Once the product succeeds and its bottlenecks show up, the custom-tailored solution is built to address the performance needs, either by picking better technologies or making the architecture more scalable.

Another aspect worth mentioning is that both Jenkins and Kubernetes are open-source projects. Therefore, anyone can download and use their source code for free according to the permissible use specified in their licenses. However, enterprises often do not use these projects directly but pay for subscriptions or use them as a service. This can grant them customer support or the ability to focus on the added value of their products without worrying about the underlying infrastructure. And even though it is not a primary concern here, because engineers' time is often more expensive than the subscription price, it might be better for companies like Red Hat to invest in the offerings they actively develop and sell rather than into their own competition.

Also note that for scheduled jobs, this thesis suggests using Kubernetes CronJobs as an off-the-shelf solution because there is no added value in implementing anything custom.

Chapter 8

Conclusion

The goal of this thesis was to develop a task execution framework that would listen to messages sent on a message bus and react to them by executing jobs on OpenShift. As a result, two applications were created that facilitate this workflow – a client application communicating via Streaming Text Oriented Messaging Protocol with the message bus and a Flask application creating Kubernetes resources on the OpenShift cluster as a response to these messages.

At first, this thesis discussed the basic concepts of distributed computing, workload isolation with containers, and container orchestration platforms Kubernetes and OpenShift. It also mentioned Tekton as one of the possible technologies for implementing part of the functionality and weighed its trade-offs. It described basic concepts for understanding the intricate details of messaging, its ecosystem, and protocols.

The created solution aims to replace an existing one currently running on Red Hat's release pipeline. As explained in a chapter describing the Jenkins setup in more depth, its features include triggered and scheduled job execution. While the functionality of scheduling tasks was initially in the scope of the designed solution, I quickly realized that it would be better to leverage Kubernetes CronJob resources for this workflow.

Therefore, the core of this work focused on triggering task execution based on messages coming from the message bus. I collected the requirements for the system, designed the solution, and implemented it. The final solution meets the requirements, including handling error cases such as client failure or temporary disconnection of one of their components. Also, extensive logging is in place to make any problems easier to debug. In order to increase the quality of the software and reduce the number of regressions, the codebase was covered by unit tests. Scenarios for manual end-to-end testing were created to showcase the resulting functionality and test its performance. The resulting design was then compared to the alternative approaches and weighted the advantages and disadvantages of custom-tailored solutions.

Deployment of the solution was out of the scope of this thesis and, therefore, will be the natural next step. The Flask application will need a production-grade WSGI (Web Server Gateway Interface) server because the one that is part of the Flask package is insufficient. While the current setup can handle the required workflow just fine, HTTP requests could be made asynchronously to prepare for possible future performance needs.

Bibliography

[1] Anil, N. et al. *Communication in a microservice architecture* [online]. 2022 [cit. 2023-03-27]. Available at: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture.

[2] Harrington, B. *OpenShift and Kubernetes: What's the difference?* [online]. 5. February 2019 [cit. 2023-03-30]. Available at: https://www.redhat.com/en/blog/openshift-and-kubernetes-whats-difference.

[3] Hohpe, G. and Woolf, B. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. 1st ed. Addison-Wesley Professional, 2003. 139 p. ISBN 9780321200686.

[4] IBM. *IBM: What is container orchestration?* [online]. [cit. 2023-04-11]. Available at: https://www.ibm.com/topics/container-orchestration.

[5] Kaplarevic, V. *Understanding Kubernetes Architecture with Diagrams* [online]. 12. November 2019 [cit. 2023-04-11]. Available at: https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams.

[6] Lord, D. *Flask's documentation: Application Structure and Lifecycle*. 10. February 2023 [cit. 2023-04-02]. Available at: https://flask.palletsprojects.com/en/2.2.x/lifecycle/.

[7] Ochoa, G. *Tekton documentation: Getting Started with Triggers* [online]. 29. January 2023 [cit. 2023-04-16]. Available at: https://tekton.dev/docs/getting-started/triggers/.

[8] Stomp Spec Group. *STOMP Protocol Specification, Version 1.1* [online]. 22. June 2010 [cit. 2023-03-31]. Available at: http://stomp.github.io/stomp-specification-1.1.html.

[9] Stratton, M. et al. *UMB Appendix* [online]. 3. May 2021 [cit. 2023-04-23]. Available at: https://source.redhat.com/groups/public/enterprise-services-platform/it_platform_wiki/umb_appendix.

[10] Stratton, M. et al. *UMB Client Guide* [online]. 15. April 2021 [cit. 2023-04-23]. Available at: https://source.redhat.com/groups/public/enterprise-services-platform/it_platform_wiki/umb_client_guide.

[11] Uhrig, T. *Queues vs. Topics vs. Virtual Topics (in ActiveMQ)* [online]. 22. May 2017 [cit. 2023-03-27]. Available at: https://tuhrig.de/queues-vs-topics-vs-virtual-topics-in-activemq/.

Appendix A

CD Contents

- **umb-tasks/*** - source code of the applications developed as part of this thesis
  - **api/**- source code of API application
  - **client/** - source code of client application
  - **common/** - modules shared by both applications
  - **tests/** - unit tests for both applications and shared modules
  - **.gilab-ci.yml** - definition of a CI pipeline
  - **Makefile** - file with helper scripts
  - **README.md** - file with information about development setup, configuration, etc.
  - **client.py** - launcher script of client application
  - **sender.py** - utility script used for testing the solution
  - **wsgi.py** - launcher script of API application
  - other files not mentioned here are described in README.md
- **thesis/** - source code and other resources for generating this thesis
- **xticha09-thesis.pdf** - final version of this thesis