UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Resilient System deployment using Kubernetes

Deployment widerstandsfähiger Systeme mittels Kubernetes

Bachelorarbeit

verfasst am
Institut für Softwaretechnik und Programmiersprachen

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Lenard Jensen

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Karam Kharraz, Maria Ostania, Tobias Braun

Lübeck, den 27. Mai 2020

Erklärung

Ich versichere an Eides statt, die vorliegende Arbeit selbstständig und nur unter Benutzung der angegebenen Quellen und Hilfsmittel angefertigt zu haben.

Lenard Jensen
Lübeck, den 27. Mai 2020

Kurzfassung Mit der Zunahme von Software, welche in der Cloud betrieben wird, insbesondere bei Programmen, die sich aufteilen lassen in viele einzelne Microservices, muss es auch Werkzeuge geben, um jene Software zum laufen zu kriegen und sie dabei zu verwalten. Eines dieser Werkzeuge zur Microservice Verwaltung ist Kubernetes, welches schon in vielen DevOps-Pipelines zu finden ist. In dieser Arbeit wird eine Logistikplattform erstellt werden, welche viele verschiedene Anfragen mithilfe des zentralen Microservice Apache Kafka verteilen wird. Dabei werden alle Konzepte, die während des Vorgangs auftreten erklärt und in dem Kontext des Beispiel-Deployments gezeigt. So erhält der Leser einen ersten Einblick in containerisierte Anwendungen durch Docker und wie jene Software durch Kubernetes verwaltet wird. Zusätzlich wird dieses Beispiel-Deployment dann auch noch auf seine Widerstandsfähigkeit gegenüber Softwarefehlern getestet und auch wie gut sich Kubernetes allgemein dabei bedienen lässt. Bei den Tests wird geprüft, ob die Software widerstandsfähig bleibt bei Auslösungen verschiedener Fehlerzustände, als auch ob die Software einer hohen Zahl an Anfragen pro Sekunde bearbeiten kann.

Abstract With the rise of software running in the cloud, especially programs that are split into many microservices, there needs also to be tools to deploy and manage that software. One of those tools for microservice orchestration is Kubernetes, which already is part of many DevOps Pipelines. In this thesis we are demonstrating the deployment of a logistics platform that will distribute many different requests using Apache Kafka as one of its main microservices. All concepts that occur during that process will be explained and shown in the context of that example-deployment. The reader will gain a first insight into containerized applications with Docker and how Kubernetes orchestrates those applications. Additionally, this example-deployment will be tested for its resilience against failures and how well Kubernetes can be used in general. Those tests checked whether the software remains resilient during various failure states, as well as whether the software can handle a high number of requests per second.

Contents

1 Introduction

During the lifecycle of any software, one will certainly encounter the phase of software deployment, especially with services that run on servers instead of being shipped via an executable file to the user. That deployment process should, best case, be rather efficient and uncomplicated, while the software itself should run in a way that it could bear the expected workload and does not need a dedicated developer to restart it after any small failure.

For that use-case, the platform Kubernetes was created to deploy and manage microservices running across any number of servers while keeping the process, once it's set up, simple and efficient. Additionally, Kubernetes supports continuous integration and automated pipelines, which are central parts of the DevOps development style.
With all those claims about this platform, it may be worth to look into Kubernetes and see if those claims hold, or if the needed work required exceeds the effort of manual service deployment.

1.1 Contributions of this Thesis

The main goal of this thesis is to concisely explain the different aspects of Kubernetes, the parts and objects that appear during deployment, and whether it can be used to make the software more resilient. This thesis will not replace any guide about Kubernetes but instead tries to introduce the platform to the reader and help them understand what Kubernetes does exactly. Additionally, one can find here an introduction to DevOps and the motivation of using Kubernetes during the DevOps process.

The reader will also find here an introduction to the concept of containerized software, an explanation of Kubernetes functions, and what resilience means in the context of deployed microservices.

1.2 Results of this Thesis

In the end, Kubernetes upheld its claims and the expectations put into it. Software deployment is made especially easy when the whole cluster is up and running. After that, one can write a deployment manifest once for the whole system, that will be applied to any later versions of the microservices, keeping the overhead of Kubernetes manageable. Because of the level of automation available to the user and integrations by platforms like GitLab, it turned out to be a great tool for DevOps as a part of the automated pipeline.

Regarding the resilience, Kubernetes also turned out as expected. Kubernetes itself cannot prevent any failures of the microservices, as those are a result of something affecting the software itself, like faults in the coding or missing resources. However, instead, Kubernetes manages to uphold the desired deployment state and will always promptly correct everything to resemble that state. Because resilience for this thesis was defined as an ability to recover from failures, it means that Kubernetes succeeded in providing that resilience to the deployed software.

1.3 Related Work

As Kubernetes is a tool created to deploy software in a resilient way, there are many other sources about this subject. While the main documentation of Kubernetes is pretty solid in that regard, one of the main sources that aided during this thesis was the book “Cloud Native DevOps with Kubernetes”[AD19], as it also works through every step of the software deployment with Kubernetes.

Concerning Kafka and benchmarking it, the results of Jay Kreps[Kre14] may be interesting as they load tested Kafka in a similar way done as in this thesis. As there were some mismatches between the results in the blog post and this thesis it might also give additional insight into the behavior of Kafka.

Another interesting talk about Kafka running on Kubernetes can be found with “Kafka on Kubernetes: From Evaluation to Production at Intuit”[Jav18], as the tested setup is very similar to the setup done in this thesis, with the same Kubernetes-Kafka implementation.

1.4 Structure of this Thesis

This thesis is structured in a way that a reader without specific knowledge about Kubernetes will learn about the most basic concepts first, followed by more complex parts that are dependent on earlier sections. Therefore it is advised not to read the chapter independent from one another, but rather read them in the order presented here.

Chapter 2 is mainly concerned with introducing the reader to the different concepts used in this thesis, with a quick overview of Docker, followed by the basics of Kubernetes and the main use-case of Kubernetes in this thesis.

Chapter 3 will provide an insight into the deployment process of a microservice using Kubernetes, while also explaining the methodology used in the resilience tests of that deployment.

Chapter 4 evaluates the resilience tests that were described during the previous chapter as well as the user experience of working with Kubernetes.

2 Preliminaries

In this chapter, we will take a look at the programs and concepts used in this work. This chapter will not explain every detail of Kubernetes, Docker, and other things but look at the relevant parts that will be used later to deploy the Logistics Platform in Chapter 3.

2.1 Docker

Before focusing on the main piece of software that will be looked at in this thesis, we first have to take a look at Docker and its provided functions. Docker is an open-source software made to create images of containerized software. These lightweight virtual machines run another software while keeping track of the needed dependencies so that the user has to install almost nothing on his machine. The main appeal of Docker and its containers is that the containerized software can run on every machine where the Docker Engine is installed, without the need to install any dependencies of the software, as they are already installed inside the container VM. Additionally, the containers themselves are entirely isolated from the host machine, reducing any interaction with other programs to network communication, as if the containers were running on different machines inside the same network. Therefore, if the software was able to run on the programmer's machine, it should almost always run the same way after deployment on any other machine while reducing the clutter of dependencies on the deployed server, where they otherwise may conflict with each other.

With Docker, one can simply take his compiled software and create a so-called DockerFile containing instructions about any dependencies, such as external libraries or even the OS inside the container[SD19]. After container creation, it will then be saved as an image that can easily be uploaded to either Docker Hub or any private repository and deployed. Additionally, one can even tag the images with extra information such as the version number or the release name for later update deployment.

It is easy to imagine why one would choose to use Docker containers to deploy any type of microservice with its different functionalities and ease of use. That is also precisely why Kubernetes takes advantage of containerized microservices instead of running programs directly, as we will see in Section 2.3.

2.2 Automation and DevOps

One of the main benefits of using Kubernetes, or any container orchestration platform for that matter, is the number of automated processes it provides. For instance with Kubernetes one does not need to manually delete old microservices and add new ones to every server inside the cluster, Kubernetes updates the deployment information and automatically adjust everything to the new parameters, e.g., five microservices of version 1.0 get replaced by 10 new pods containing the 2.0 version of the program via a rolling update.

With that in mind, would it be a good idea to look into the concept of DevOps and how its aspects could help while working with Kubernetes and other similar platforms?

DevOps is a rather new concept that was introduced in 2009[LKK$^+$19] that is still missing a definitive definition in the industry[WFW$^+$19], but there are many definitions attempts that agree with one another. We will take a look at some key parts of the different definitions that are relevant for the work with Kubernetes. In DevOps, software developers will also look over the deployment and take on the responsibilities of the operations team. Hence the name DevOps being a portmanteau of development and operations.
[WFW$^+$19] defines, amongst other aspects, that DevOps is a concept where, during the product's life cycle, there will be many software versions with fast revisions during a continuous integration process. The development will be aided by the use of automated tool pipelines for testing, deployment, monitoring, and updating. On the non-automated side of things, DevOps also describes how user feedback is directly given to the developer team that will handle everything internally without the need to communicate with other departments if not necessary. DevOps, with its focus on automation and communication structure results in a highly efficient software life cycle. While the key aspects of the DevOps method can be observed in one form or another in many different companies[LKK$^+$19], it is important to note that every developer team is free to define what their version of DevOps is due to it being more of a suggestion or guideline rather than a strict set of rules, making it applicable to different types of software development.[WFW$^+$19]

One may now have already noticed how the DevOps concept also fits into Kubernetes and its primary use cases. While Kubernetes does not cover the collection of user feedback or the coding of microservices, it helps with the deployment, monitoring, and version controlling. Developers no longer need to account for manual restarts or general resource management, as Kubernetes will try to keep everything running inside its scalable cluster according to the specified deployment. Kubernetes is also able to deploy rolling updates, which is an update that might only be deployed to a certain percentage of users at a time in order to keep the software running if there is an error with the new version. In case of an update to a bugged version, one can roll back the cluster to the older version without much work needed.

With the observed benefits of DevOps, as seen in [LKK$^+$19], it is therefore important to always keep the key aspects of it in mind while working with Kubernetes. Instead of viewing the software as a web of replicated nodes and connections, one can view it as an abstract cluster of services that can communicate with each other. As Kubernetes can also be inserted into a continuous integration pipeline, it can automatically update and deploy the software without any input by a developer, therefore keeping the whole development process more efficient.

2.3 Kubernetes

We will now look at the main piece of software for this project, which will enable the capability of higher resilience. Kubernetes is an open-source platform designed to deploy, scale, and update applications that were containerized using Docker. Its main appeal is the ability to create deployments, objects that describe a state of the application that Kubernetes should maintain or try to achieve. An example of a deployment may be that the application should run on two servers, each running four instances of the microservice. If something unexpected were to happen and one of those instances crashes, Kubernetes will try to restart it to maintain the desired state.

It is now important to see how Kubernetes achieves what it does and what components play a role in this. Deployments are not just vague descriptions of the application but rather describe how the *Pods* running on the server should be working. *A Pod is the Kubernetes object that represents a group of one or more containers*[AD19], is one of the easiest explanations regarding Pods. When one is creating a deployment, the main thing that happens is that Kubernetes creates a group of Pods containing the containerized microservices that will then be distributed among the servers, each running independently with the ability to communicate with other Pods. Microservices that share the same pod also share their memory. As a rule of thumb, if the containers still work as intended distributed among different machines, one should put those containers inside multiple Pods.[AD19]

After understanding the nature of Pods, we will start to look at the structure between servers and how Kubernetes can manage those. Kubernetes will group the provided servers into a cluster, where each server acts as a node that will carry out the necessary work like running a group of microservices. Those nodes will then be organized by the master-node, a single server tasked with keeping the nodes running according to the current deployments and the main server that is reached when

using the `kubectl` interface. While the master is supposed to restart any crashed or malfunctioning server, it has no other server monitoring the uptime. That means in case of a crashed master that the whole cluster can not receive new information regarding the deployments and any node crashing will not result in a quick restart or redistribution of resources, but the nodes will still run their microservices in accordance to the last commands of the master.
Due to the importance of the master node, one should not run any heavy services on it and fully dedicate that specific server to being the master. It may be possible to run multiple master nodes to increase the resilience of the whole system, but that case will not be tested in this thesis due to the provided server setup as described in Section 3.6. With multiple masters, it would be possible to always have one master that keeps the other one alive in case of a failure, just like Kubernetes generally tries to keep the whole cluster alive.

With every pod running and replicated, we are just faced with the issue that there still needs to happen some kind of communication between the different servers and pods. The solution to that problem are the Kubernetes services. Note that the term "service" in this context stands for a specific type of interface used by Kubernetes for pod communication and not the deployed software itself. To communicate with the replicated microservices on the different nodes with their dynamic IPs it is necessary to use those services that will reroute any traffic to its correct target.
Instead of discovering the microservice on a certain node that needs to be reached, one can simply send the traffic to a specific service that knows the relevant IPs of the nodes/pods and just simply connects to the correct pod, as depicted in Figure 2.1. Pod-to-pod communication can, therefore, be viewed on a more abstract level by just imagining a microservice-to-service communication with a constant complexity regardless of replication factors. Services can be configured to run either on internal or external IPs, e.g., the most basic service a "ClusterIP" is just an internal IP reachable by other pods inside the cluster, while a "NodePort" uses the external IP of a specified node in addition to an opened port to redirect the traffic.
A Kubernetes service could be described like a DNS, e.g., instead of finding the IP address of the third instance of microservice X running on node 2, one can define a service that will connect to any microservice X instances and using that service's name instead of the dynamic pod IP.
Additionally, those services also act as the main way to balance the load between the different pods. Per default, any service will try to route the traffic to a random pod matching the service description[1]. If that approach is not considered optimal, or a more refined behavior is wished, Kubernetes allows the user to implement a custom load balancer or the usage of an implementation build by an external cloud host like Google or AWS.

[1]Since Kubernetes version 1.10 one can also replace the random default algorithm with a round-robin or least-used selector on a system wide level.

With the main components of a Kubernetes cluster out of the way, let us also have a quick look over the different applications that will enable us to create and run a Kubernetes cluster, Kubectl, Kubeadm, and Kubelet.

- Kubectl: Our main interface with the Kubernetes master that enables the deployment of pods, listing of resources, and the general management of the platform. Due to its complex nature with its many applicable arguments, there will not be a specific explanation of Kubectl in this thesis.
- Kubeadm: By running Kubeadm on a server, it is possible to create and new Kubernetes cluster and also promote that server to a master node that will keep the whole system up and running. If there is already a running cluster, kubeadm also enables the server to join the cluster as a new node.
- Kubelet: This is the main background process that will control any node according to the directions of the master node, like creating new pods or updating running services.

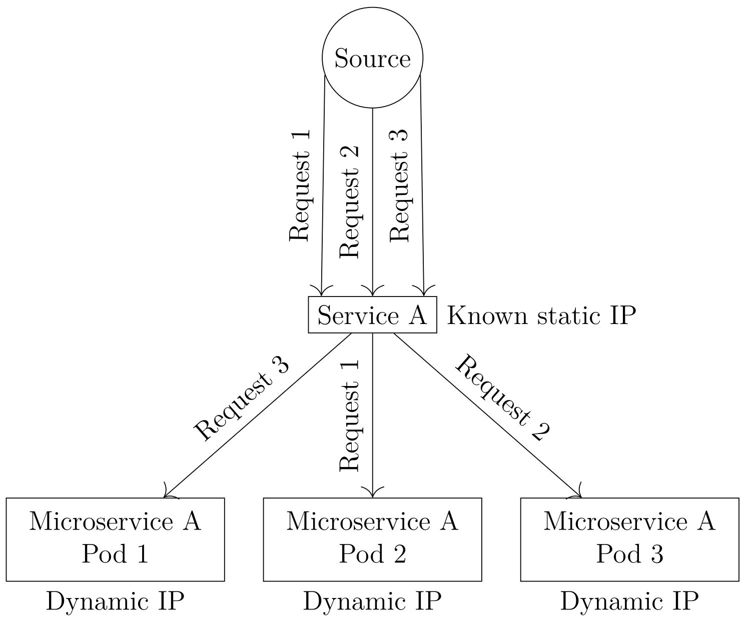

Figure 2.1: Kubernetes service handling traffic with random distribution

2.3.1 Docker Swarms in Comparison

While Kubernetes is the software we will use in order to manage the containerized microservices, it is not the only platform designed for that purpose. Docker itself has its own container orchestration platform called “Docker Swarm”, another open-source platform that is developed by the same company that is also working on Docker. This raises the question of whether Docker Swarm is better or worse than Kubernetes and why it was not chosen as the main platform.

While both Kubernetes and Docker Swarm offer the same services, being a resilient and scalable deployment of containerized software as a cluster, one can still find differences between them. With both Kubernetes and Docker Swarm being first released in 2014, they both managed to stay in a relative range to each other, with neither one being the absolute best choice. For instance, it may be harder to install and create a Kubernetes Cluster in regards to a Docker Swarm, the cluster managed by Kubernetes will be stronger and more convenient in the long run with features such as auto-scaling and in-build monitoring. On the other hand, Docker Swarm will offer a simpler installation, while also being faster at container deployment and scaling of the cluster[SD19]. Kubernetes is, in the end, the more popular choice due to it offering more with its complex nature and features, making the hard installation one of the only caveats[AJBB^{+}19].

2.4 Kafka

Because Kafka will be the software mainly used in the resilience tests of Kubernetes in Section 4.2, it is necessary to have a quick overview of some functions Kafka can provide for the Logistics Platform. As stated before, there will not be an in-detail explanation of Kafka like we have seen with Kubernetes, as this thesis is more about the general deployment of software instead of the deployment of Kafka.
Nonetheless, as there will be mention of certain functions of Kafka in Section 4.2, its important to generally describe them in order to avoid confusion. A brief overview of the software components needed to run a Kafka server can be found in Subsection 3.3.1, where the deployment of Kafka was explained.

The main function of Kafka is to provide a platform that can accept a stream of messages published by other sources, like other components of the software, and share those messages with anyone subscribed to a specific topic in Kafka. Something that is generating messages is called a producer, while something reading the messages is a consumer, both can be either a client or another microservice. Producers and consumers both interact with so-called *topics*, those are predefined during the setup and are used to categorize the messages sent to Kafka. Any single topic can run on multiple Kafka nodes, also called *brokers*, where they are replicated and kept synchronous. One can define on how many Kafka instances the topics are replicated for redundancy and how many partitions are split between those replicated copies. The partitions are used to split up the writing and reading process from a topic, like in Figure 2.2, where messages are written onto different partitions instead of just a single block of data. Those partitions may also be split up between the replicated brokers to balance the load generated.
One thing to also keep in mind is that the messages written and consumed by Kafka

will be grouped into a batch and compressed for a more efficient data transfer. If a message exceeds the batch size, it will be split up into multiple batches, but if a batch still has some space left, Kafka will try to fill that batch before sending it. The batch sizes and the compression rate may also have an impact on Kafka's performance and should, therefore, be tweaked accordingly. However, as the options to tweak those things were missing in the used software, those test cases could not be made.
While multiple producers and consumers can publish to a single topic, or be subscribed to the published data, one has to note that producers can only append data to a topic and not insert it arbitrarily, while consumers can read the data from anywhere in the topic as seen in Figure 2.3. The impact of replication and partition numbers can also be found in the evaluation in Section 4.2.
Those replications with their different partitions have to be kept in sync, as they would otherwise have an incomplete amount of data about any single topic. That synchronization task is not done by the brokers themselves, but rather another program called Apache ZooKeeper that will handle the traffic to the brokers and coordinate them to keep the data replicated and in the right order on every node. Because ZooKeeper is another program that runs alongside the main Kafka brokers it complicates the whole Kafka setup, which is explained in Subsection 3.3.1.

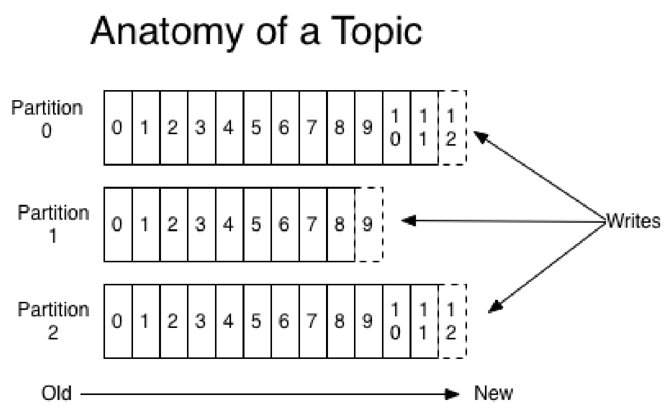

Figure 2.2: Topic being split into 3 partitions with multiple producers writing onto the same topic data
Source: https://kafka.apache.org/intro

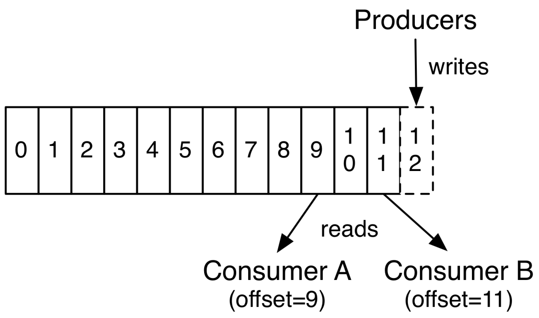

Figure 2.3: Different consumers reading the same partition with multiple producers writing data
Source: https://kafka.apache.org/intro

2.5 Resilience

While the preceding sections were about types of software or a specific development process, it is essential to look into the aspect of resilience itself. As the concept of "resilience" is a broad term ranging from security concerns to resilience against wrong user-input and so on, we first need a specification of what resilience means for this thesis.

For this project, the resilience should be found in the capability to have "the ability to recover from failures commonly encountered in the cloud"[HRJ+16], while staying reachable at all times.

For instance, if we have a microservice replicated on n servers, we should only ever observe $n-1$ downed servers running the replications during the software's runtime. In a best-case scenario, one can then still reach and interact with the deployed software, while the $n-1$ servers are starting back up. A service with multiple replications is, therefore, inherently more resilient to any kind of random failures occurring during runtime, as the probability of that failure happening on every replica or server in cluster is rather small. While there is undoubtedly a focus on having the service withstand a big workload generated by external users, one has to also pay attention to any software failures and errors that can render a specific server unusable until it gets fixed. In this case, the workload present would be a high number of accesses per second with only a few bytes sent per request, rather than single large packs of data. There may even be the possibility of having additional servers that are usually not concerned with running the software but could be used as a backup strategy in case of any server outage inside the main cluster of workers.

The observed resilience will also be the primary benchmark if Kubernetes could fulfill its expectations and will, therefore, be one of the main topics of Chapter 4.

3 Methodology

This chapter will cover the methodology used regarding Kubernetes and its microservices in the implementation of the ISP Logistics Platform. The platform will be described in Section 3.1, but the chapter's main focus will be on a more abstract level due to the focus on Kubernetes rather than the platform itself.

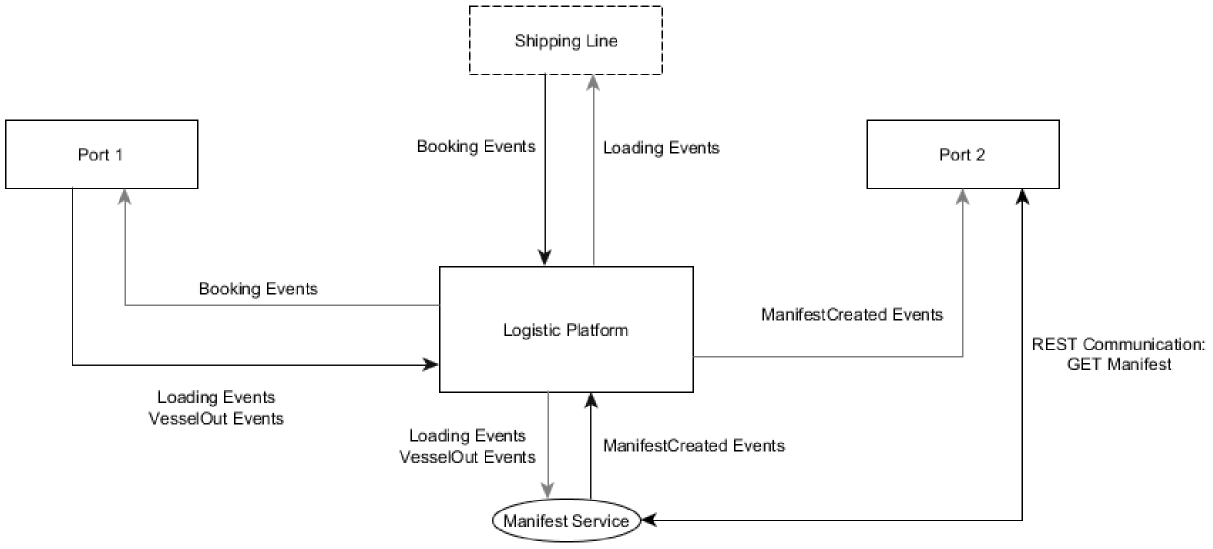

Figure 3.1: Abstract overview of the Logistics Platform's connections

3.1 The Logistics Platform

While the Logistics Platform and its specification is not the main focus of this thesis, it is what prompted the usage of Kubernetes and provided the general goals. Therefore we will take a quick look at the platform, and its idea. In this case, the goal was to create a resilient cluster of microservices that will send and receive information to and from different users. The users here will be different logistic ports that all have their processes and databases, as depicted in Figure 3.1. In case Port A wants to communicate something to Port B, like a ship leaving the port with an estimated time of arrival X, they would have to know each other's preferred way of communication while also using the correct data format.

The project's motivation was to create a centralized hub where information in the shape of events will pass through and get redirected to the correct target while removing the heavy lifting from the users. For that approach, it was decided to create a scalable cluster of microservices that are capable of working with the provided data and sending it to the right port. For that was Kubernetes chosen as the main platform to create and host the microservices that will parse and send the data, while also hosting a Kafka Server as the persistence layer that distributes the parsed information. Those messages sent to and from the platform will only consist of JSON objects that carry information about the various SQL databases found on the user's side, therefore reducing the messages themselves to only a few kilobytes per request. The data in this platform will only be stored temporarily until another microservice consumes that data, lifting the necessity of having large data archives that may take up server space. Kafka itself will not be explained in detail, as its scope might be on the same level as Kubernetes, but we will look over its deployment onto the Kubernetes cluster with Helm later in Section 3.3.

3.2 Deployments with Kubernetes

Now we will look into Kubernetes' features used in this project, what the idea was behind them and how it was done in the end. While the main idea is later to deploy a whole set of microservices inside the cluster, for this thesis, only two groups of microservices will be deployed, a microservice that collects data from a database, and a Kafka cluster for temporary storage and sorting of the data. Both groups were deployed differently, with the microservice using a handwritten manifest .yaml file and the Kafka cluster using an external Helm chart. We will also look in Section 3.3 into the details of Helm and the advantages of using it.

First, we take a look at the containerized microservice used to collect database entries and send them to the Kafka cluster. To create the deployment, one first needs to generate a Docker container containing the application. In this case, the program was already compiled into a JAR file, making the containerization rather simple. However, regardless of the programming language used for the microservice, the result will always be a container acting like a black-box making the following steps of the process similar regardless of the libraries or other extras used. In order to turn the container into a pod, it is now necessary to write a manifest file, a .yaml file that specifies the created pod and its deployment.
While the term "deployment" used throughout this thesis in conjunction with general pod creation, it is necessary to state that its possible to create and deploy a pod without a "deployment". A pod without one will still function correctly and execute its functions, but the main difference and disadvantage is that Kubernetes

will not try to fix any crashes or other malfunctions. In the case of a crashed pod without a deployment, Kubernetes will not restart or scale the pod, as there are no specific instructions given.

Here are two pods that were deployed onto the Kubernetes cluster with varying complexity, a Kafka cluster client used to directly issue commands to the Kafka cluster and the SQL-to-Kafka connection microservice. Note that the Kafka client is just a pod without a deployment, while the microservice is contained within a deployment.

```
apiVersion: v1
kind: Pod
metadata:
  name: kafka-client
  namespace: my-kafka-namespace
spec:
  containers:
  - name: kafka
    image: confluentinc/cp-kafka:5.0.1
    command:
      - sh
      - -c
      - "exec tail -f /dev/null"
```

Source Code 3.1: Pod containing the Kafka client

As we can see, the kind of manifest we have in Sourcecode 3.1 just a pod, the name and namespace will help find and categorize the pod later on, while the spec contains what image is put into the pod and the commands in the last few lines specify what commands are issued to the container after creation. The deployment shown in Sourcecode 3.2 is a little bit more complex than just a simple pod. First, one has to specify the deployments name after that it is necessary to state how many replicas of the same pod should exist at a time, followed by the normal pod declarations. This specific pod also needs some environment variables, like the database and Kafka IP, as well as some more or less private data, e.g., the database login credentials. Because it would be bad to breach security and simply display something like a password in plain text, one can also create local files containing such information. Lastly, it may also be necessary to provide some kind of authentication token for the Docker image registry, because not just everyone should be able to pull one's code.
For that case, Kubernetes allows the creation of secrets where one can store private information inside the temporary storage of the kubelet application for later usage, but be aware that while secrets are saved locally and usually should not leave the cluster, they are still only just encoded in base64 and not hashed. The encoded information can always be accessed and converted back into plain text, while this

may be insecure against a human attacker with root access, any pod not authorized to see the secret should not be able to get its contents, as Kubernetes will not provide the secret to those pods.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sql-producer-deployment
  labels:
    app: sql-producer
spec:
  replicas: 1
  selector:
    matchLabels:
      app: sql-producer
  template:
    metadata:
      labels:
        app: sql-producer
    spec:
      containers:
      - name: sql-producer
        image: docker-image-url/my-docker-image
        env:
        - name: KAFKA_SERVERS
          value: my-kafka-server-ip
        - name: SQL_SERVERS
          value: my-sql-server-ip
        - name: DATABASE
          value: my-database
        - name: USERNAME
          value: my-username
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-pass
              key: db-pass.txt
        - name: JDBC_TYPE
          value: mysql
      imagePullSecrets:
        - name: regcred
```

Source Code 3.2: Deployment of the sql-to-kafka microservice

3.3 Helm

As previously mentioned, not every microservice running in Kubernetes needs to be deployed by handwritten manifest files. If, for instance, one wishes to deploy an external microservice, written by someone else, for their platform, then they might not want to read the complete documentation about any dependencies and special parameters. Instead, one can opt for looking for an existing deployment template, called a Helm chart, to simply install an deploy the specific microservice.

Helm is a package manager for Kubernetes that can create and install charts which specify everything a specific software might need to run.[AD19] After pulling a chart from a repository, one can then tweak certain flags put in place by the chart creator to adjust the deployed software to one's specific requirements and then simply deploy the software. Behind the scenes, Helm uses a pre-written manifest file that will then be filled with the information from the flags or config files and then handed to the Kubernetes platform just like any other manifest would be. Those flags could be as permissive or strict as the chart's author wishes them to be. E.g., one could write a helm chart with just a flag for a debug mode, or on the opposite side, have a helm chart where almost any value can be edited from the default one. In the end, the main difference between manual deployments and Helm chart installations are that someone with more insight about the to-be-deployed software has already filled out the forms necessary for a seamless deployment, lifting unnecessary work from the developer and keeping them focused on the main tasks.

3.3.1 Kafka deployment

For this project, the software that needed to be deployed was Apache Kafka, a streaming platform used to receive, store, and provide information used by the microservices. Kafka itself runs as a cluster with many nodes that need to communicate with one another, while also temporarily storing data on the hard drive as a stateful service. Those microservice clusters then also need to sync their storage data by using another microservice called Apache ZooKeeper that runs as another cluster of replicas besides the main Kafka brokers. Even ignoring the fact that a Kafka is not just one monolithic microservice, but also requires persistent volume storage means that a manual deployment via normal manifest files can quickly end in a logistical nightmare that ends up taking valuable time away from the project.
However, with the usage of Helm suddenly, the whole deployment became a rather trivial thing, where only some flags needed proper adjustment for the logistics platform. Helm created a deployment for the Kafka and ZooKeeper nodes, made the appropriate services for communication, and also filed volume claims that told Kubernetes to allocate the correct storage volumes to the nodes.

```
Persistent Volumes of Cluster
NAME          CAPACITY  ACCESS  STATUS   CLAIM
pv-volume-01  4Gi       RWO     Bound    datadir-k-1
pv-volume-02  4Gi       RWO     Bound    datadir-k-0
pv-volume-03  4Gi       RWO     Bound    datadir-k-2

Persisten Volume Claims of Pods
NAME         STATUS  VOLUME        CAPACITY  ACCESS
datadir-k-0  Bound   pv-volume-02  4Gi       RWO
datadir-k-1  Bound   pv-volume-01  4Gi       RWO
datadir-k-2  Bound   pv-volume-03  4Gi       RWO
```

Source Code 3.3: Overview of peristent volumes and their claims by the Kafka brokers

As Kafka needs allocated disk space inside the different nodes one has to create placeholder volumes on the cluster that will be bound and used by pods that filed a "persistent volume claim". In the example found in Sourcecode 3.3, we can see the three different volumes that each allocate 4Gi on their respective nodes, which are then claimed by the pods to store any data that needs to be retained between restarts.
In the end, the only manually written .yaml files were the ones creating the storage spaces, as the helm chart did not create them, but that single step compared to the work done by Helm is way more bearable for a developer not 100% familiar with Kafka. Additionally, almost all parameters tweaked in the resilience test could be easily found in the config file of Kafka's Helm chart, as it contained a parameter for almost everything that would have been otherwise found in the deployment manifest. E.g., an important setting provided was the anti-/affinity between pods, as it is recommended to prevent the different Kafka pods from staying on the same server and group them with their respective ZooKeeper Node. Affinity is mainly used to spread the workload of the microservices between the different servers, while also grouping services together that would benefit from communicating quickly with one another. It also prevents the case where a single server failure takes down a complete set of replicas, if they were all grouped on that specific server.

```
## Replication of the deployment
replicas: 3

## The kafka image repository
image: "confluentinc/cp-kafka"

## Configure resource requests and limits
resources: {}
  # e.g.,
  # limits:
  #   cpu: 200m
  #   memory: 1536Mi
  # requests:
  #   cpu: 100m
  #   memory: 1024Mi

## Pod scheduling preferences
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: app
          operator: In
          values:
          - kafka
      topologyKey: "kubernetes.io/hostname"
  podAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
     - weight: 50
       podAffinityTerm:
         labelSelector:
           matchExpressions:
           - key: app
             operator: In
             values:
               - zookeeper
         topologyKey: "kubernetes.io/hostname"

## Persistence configuration.
persistence:
  enabled: true
   size: "4Gi"
```

Source Code 3.4: Parts of the Helm chart's config file for Kafka

3.4 Resilience Testing

A main part of this thesis next to the software deployment is also the part about its resilience. The final results of the resilience test can be found in Chapter 4. In this chapter, we will focus on the setup and the reasons behind it. Due to Kafka being already deployed on the Kubernetes cluster as a service planned to be used later in the production environment, made it the prime candidate to resilience test it. Next to Kafka, there were only microservices running mockup or demo versions that were never designed to be tested thoroughly in terms of resilience, as they are going to be replaced by their proper counterparts in production. The good thing was that Kafka will probably also be the component in the final product receiving the main part of the incoming workload, as a central message hub between other microservices. That is why the resilience of the Kafka service will greatly influence the resilience of the whole platform.

With resilience being something that focuses on the tolerance of a system against any failures, there needed to be a way to introduce failures into the system without having to hope for a randomly occurring one. In the end, two ways were chosen to generate them. The first one was to simply overload the system with too many inputs per second, as it should result in software crashes due to missing resources in the system. The other way was to simulate a pod/server malfunction by simply terminating the currently working pods or shutting a server down. Any impact of those failures during the resilience tests can be found in the Section 4.2. Therefore, while load testing and resilience testing are two different things, was the load testing in some way also a tool to trigger failures for the overarching resilience test.

3.5 Load Testing

In order to load test Kafka, some things were necessary to run those tests. First, a tool for generating a significant number of messages per second to generate the workload. Additionally, another tool had to be found for test automation, as it would be infeasible to run a command line for each test manually and to download the whole test environment for each new version.

For the load generation, the tool Apache JMeter[1] was used. JMeter qualified itself for its ease of use by allowing the user to work with a GUI, good documentation, a large userbase, and the available number of plugins. With JMeter, one can easily create test-cases for Kafka load generation while also being highly portable with no installation requirements. After test-case creation, one can just send the whole

[1]https://kafka.apache.org/

directory containing JMeter and the tests to any other server and just run the program.

The test automation was solved via Jenkins[2], a program easily deployed via Docker on an external server that will be hosting the load generation. Jenkins is a DevOps centered tool, where the main goal is to automate the continuous integration process during development. As one may recall, tool-chain automation is one main aspect of the DevOps process, making such a program invaluable to the whole project, once it is up and running.
Just like JMeter, one main advantage of Jenkins is the available web-interface where the developer can define tests and their triggers. E.g., one could run a specific test any n-days or every time a new software version is pushed onto the production server. For this thesis, such specific build-triggers were not necessary due to the nature of the tests, but they can make the development much easier and efficient.

While the tools for load testing were configured correctly, with the provided setup there could be no true stress testing, as only one server had to generate the messages against a cluster of multiple servers. The exact setup can be found in the following Section 3.6, while the load testing results are in Section 4.2.

3.6 Testbed Setup

Only knowing what programs to use and how the tests themselves function is not enough. One first has to set up a testbed where the tests can be executed and monitored. For simulating the logistics platform the following configuration was chosen: The tests themselves were executed on a remote server that was not inside the Kubernetes cluster, shown in Figure 3.2. If we were to place the load generators themselves inside the cluster, it would only result in a slowdown of the workload as soon as the system would get overwhelmed. In the worst case, it could be possible to crash the Jenkins or JMeter installation, making the results unusable as they would not reflect the expected behavior.
With a remote server, it is possible to simulate a user that is sending their workload to the Kubernetes cluster with no regard of the current resource usage of the cluster, with the potential to cause an overload of events per second to the Kafka server.
The cluster itself was deployed between three different servers, as shown in Figure 3.3, each one functioning as a normal node except for a single server that also acted as the master node for the whole cluster. While it is not a good practice to run the master on an also working node, as stated in Section 2.3, it was the setup provided for this project as at least three servers are needed to create a cluster that would reasonably appear in a production environment.

[2]https://jenkins.io/

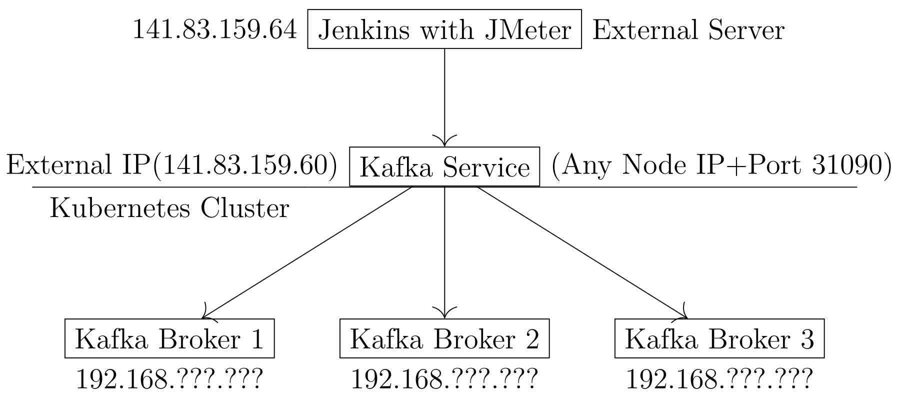

Figure 3.2: Jenkins and Kubernetes setup

On that external server would run a Jenkins installation, easily deployed with its respective Docker image, where any tests could be launched via its web interface. The tests themselves are executed by pulling the test-cases together with its portable JMeter installation into the Jenkins workspace, where a shell command could start everything.

```
./bin/jmeter.sh -n -t ./'Test Plans'/'Kafka Request-repl3.jmx'
         -l ./output.jtl
```

Source Code 3.5: Example command for JMeter execution

As we can see in the Sourcecode 3.5, Jenkins is using Jmeter just like any user would do on his own command line. One can think of Jenkins just like a normal developer running some commands in their terminal during specific times or after a specific case is fulfilled.

After the test was executed, one can look at any output files generated, which can be parsed from Jenkins by installing the specific plugin. In our case, the output.jtl file could give some insight into the average latency, the percentage of errors encountered, and how many bytes of data were transmitted to Kafka. One of the main benchmark values used in this thesis, however, the number of messages per second transmitted, was sadly not included in the output file of JMeter, but rather something read from the console output of the specific test-case. The reason for choosing that value specifically can be found in Chapter 4.

It is also important to consider the reliability of the results from the different tests. As it turned out, tests that were done during the same day with at maximum only being separated by a few hours were consistent enough when being re-done as to claim that any observed change in the messages per second observed had to be the result of the explicitly changed environment variables during the test.
Some tests were also being completely re-done during multiple days, with any observations being consistent with the previous days, to validate if the previously observed behavior would stay the same. E.g., while the general speed between two

days varied, the message rates increased or decreased still accordingly.

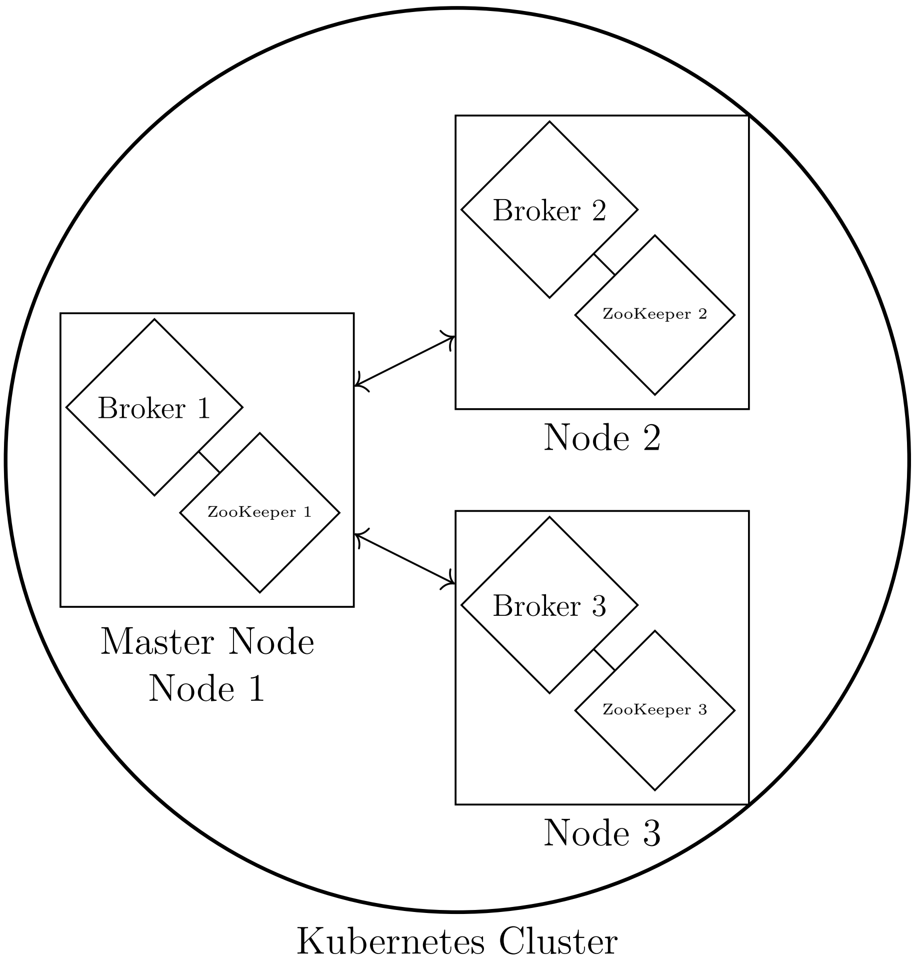

Figure 3.3: Kubernetes cluster node setup with Kafka pods

4 Evaluation

In this chapter, we will look into the results produced by the resilience tests of the deployed software, as well as a performance overview of Kubernetes. During Section 4.1, we can see how hard it was to get Kubernetes to work correctly and what obstacles need to be considered. Section 4.2 will go a little in-depth into the JMeter test-cases and how the software reacted in certain conditions.

4.1 Functionalities of Kubernetes

First, we will have a look at Kubernetes itself. While it may be important to have some statistics for the deployed software, it is also important to have a deployment platform that can be used without having some insight into old and cryptic knowledge about it.

Please note that the server setup and getting them into a specific cluster was not something that could be evaluated, as they were provided in an already functional state. It is always possible to use a preconfigured Kubernetes cluster from the many readily available cloud providers, such as Google or AWS. However, if no such provider is used for one's project, it may be necessary to combine some unconfigured servers into a cluster, where the needed overhead may vary.

Let's get one of the main concerns out of the way, the usability and user experience with Kubernetes. Kubernetes as a platform is generally more complicated than its competitors like Docker Swarm [SD19] but has a low barrier of entry once some basic concepts are learned. While there is no direct comparison to something like Docker Swarm in this thesis, as only Kubernetes was actively worked with, in a vacuum, it was not that hard to use.
One can containerize their first piece of software and quickly deploy it, as seen in Sourcecode 3.1. At its core, the manifest files containing the deployment instructions can always easily be read, while writing them is also rather painless, especially if a template was provided.
As also seen in this thesis, there are, of course, some pieces of software where the deployment was a "logistical nightmare", but that may also depend if the software was written by someone else. If, for instance, there was something different to be deployed that was coded by a dev team member, they may be able to instruct the

Kubernetes team how the software can be deployed with all the needed components. They may also be able to simply write a Helm chart that can later be used for a simple and automated deployment for that microservice.
At the beginning of this project, Helm was just something to maybe look into, but it definitely evolved to one of the core components that helped in the deployment of more complex structures like Kafka.

Before heading into the resilience aspect of Kubernetes, let's also have a quick look into the scalability of the deployed software. While horizontal scalability was not something to be concerned about for this thesis, as the provided setup only contained three server nodes to work with and no tests could be done about the speed or stability of highly scaled services, it was still something that Kubernetes just does out of the box. One could just state in their deployment specification how many copies of a specific microservice were preferred, and Kubernetes tried to create and manage that number of replicas. Scaling the microservices up or down was as simple as changing an integer value in either the Kubernetes dashboard or the deployment manifests, while Kubernetes created or deleted any number of deployed pods and always routed the traffic to the correct targets.
Vertical scalability, on the other hand, was being tested on a smaller scale, within the bounds of the provided servers. Instead of scaling the microservice horizontally across multiple servers, vertical scalability is concerned about the resources and computing power available to the microservices. As shown in the test done in Section 4.2, Kubernetes also allows to scale the resources of any pod to a specific threshold, done via values in the deployment manifests.

4.2 Resilience Testing of Kafka

For the resilience tests, as described in previous chapters, the focus was on testing a Kafka Server deployed on the Kubernetes Cluster and subjecting it to a continuous workload of randomly generated messages. This section will deal with the different test-cases and changes done to the Kafka Server between and in each test.

The main value that was taken from the different tests to evaluate them were the messages per second produced onto the different Kafka brokers. Another valuable measurement that maybe could have enhanced the observations would have been the CPU and memory usage of the Kubernetes nodes to gain insight into the resources used by the different tests. Sadly the metrics software that was supposed to run alongside Kubernetes and measure those resources could not be installed and used correctly on the provided servers due to unknown reasons with errors that could not be fixed properly.
Nevertheless, the tests still produced some usable data, with the only metric being

the messages per second. First, we take a look at why that specific metric was chosen. In the beginning, the messages per second produced should have been only a measurement if the Kafka brokers may crash during a workload of x messages per second. But as it turned out, rather than crashing after a certain threshold is reached the producer matches the maximum speed that Kafka can handle at that moment in order to prevent any crashes.
A nice side effect of that behavior was that a drop in the messages per second would always mean that the Kafka server had to throttle itself due to some loss in resources, like having to use fever computational power when other processes had to run alongside it. If, for instance, multiple users were to write onto the Kafka server and some speed loss could be observed from either one, then that would mean that too many users could, in theory, use up the complete bandwidth and effectively crash the service.

Please note that, if not otherwise specified, the parameters were as follows, 8 threads are creating the workload onto a Kafka topic running 3 replications with 3 partitions and Kafka being able to claim as many resources as needed. The CPU of the external server only allowed the use of 8 threads. Any two servers had a high enough bandwidth of 5 Gbits/s that the bytes/s produced never came close to that value, while there was also a ping latency of <1ms between them.
The latency found in the following tables is not the ping latency, but rather the latency between a sent message and its respective acknowledgment, which may take longer than a ping due to it also facting in the time the request is written and parsed. Some test cases may also include if the behavior was also replicated by other benchmarks such as [Kre14].

1. **Effect of replication:**
   The effect of having multiple partitions and replications of a Kafka topic:
   While the number of partitions has no effect on the performance, replicating the topic causes a slowdown. This behavior was also observed in [Kre14], where a replication factor of 3 would slow down the throughput from 800.000 msg/s to 400.000 msg/s.

Replication	1	2	3
Avg msg/s	5000	4200	3300

2. **Effect of broker failure:**
   What happens if any broker, or server running a broker, fails during a running message stream:
   Terminating any number of brokers with at least one broker remaining during runtime has no effect of the messages, and there is no message loss.
   Terminating the last running broker results in an unreachable Kafka service that is quickly restored as Kubernetes restarts the pods.

While running tests with an active consumer on a topic with different replication factors, it was possible to observe a short service downtime due to crashes on the different nodes, while having a replication factor of 1 or 2. Any Messages sent during that downtime were lost unless one implements a producer with redundant message generation. Those observed downtimes were noted down, as they would better reflect a genuine broker failure instead of the case where one would manually terminate them.
Note that the downtime was significantly shorter the more brokers a topic was assigned to, as the probability of any single pod finishing its restart went up, with more available pods. On the other hand, the likelihood of a failure in all pods at the same time also goes down with more pods.

Topic replication	Messages lost during downtime
1	290
2	30
3	no downtime observed

3. **Effect of topic consumers:**
Does starting a consumer during runtime have any effect on the performance: Creating a consumer on a topic with three replicas will just result in a slowdown, with more consumers resulting in further linear slowdowns on the producer side.
But creating a consumer running on a topic with only one or two replicas results in a pod failure after some messages. Kubernetes just quickly restarted the failed pods, resulting in some intervals of downtime but no complete outage. Presumably, the load was balanced enough with three replicas that no crash could occur, while two brokers could not handle the load.
However, there was no producer slowdown observed in [Kre14], instead the rate of messages per second stayed around 790.000 msg/s.

Replication	msg/s before Consumer	msg/s after Consumer
1	5000	4000
2	3500	2500
3	2600	2300

Additionally, running two consumers:
Speeds depicted are observed by the producer, not the consumer.

0 Consumers	1 Consumer	2 Consumers
3800 msg/s	3300 msg/s	2900 msg/s

4. **Effect of topic producers:**
   What happens if multiple producers are running:
   Running multiple producers on the same or different topics has no impact on performance.
   Additionally, two producers with a halved number of running threads resulted in the same performance as a single producer running on the full 8 threads.
   According to [Kre14] one should have observed a higher throughput with multiple producers, but the testbed had a bottleneck at the threads usable for production, therefore one could not have run those tests.

   Speed of single producer with 8 threads: 4000msg/s
   Producers in the table are running on 4 threads:

Topic	1 Producer running	2 Producers running
same topic	2000 msg/s	1700 msg/s
different topics	—	1700 msg/s

5. **Effect of external components:**
   Does running an external consumer results in a different performance:
   An external consumer still results in a performance loss, just like the consumer running on the same server would.

6. **Effect of reduced resources:**
   Do pods have a slowdown or crash with fewer resources available to them:
   Decreasing the memory or CPU available to the Kafka broker pods results in a linear decrease of performance, with a halved performance at either halved memory or CPU available. Additionally, decreasing the CPU results in increased latency, while decreasing the memory to a certain point causes the brokers to become unstable and prone to crashing.
   In the case of pod failures due to too few memory, Kubernetes still restarted them in a short timespan.

CPUs available	msg/s	latency
0.5 CPU	2000	4 ms
1.0 CPU	4000	1 ms
1.5 CPU	5000	1 ms

Memory available	msg/s	Crashes observed?
512 Mi	2000	Yes
768 Mi	2500	Yes
1024 Mi	4700	No
1536 Mi	4700	No

7. **Effect of slower producers:**
   Does decreasing the producers speed has any impact on performance, if the pods are throttled by their resources:
   Running a producer with halved threads on a throttled pod with halved CPU available results in the same performance as a producer with full threads on the same pod.
   It appears that the broker lowers the cap of messages per second to a point, that even a slower producer will outspeed it.

8. **Effect of containerization:**
   Does Kubernetes has any impact on the performance of Kafka:
   A server that is just running a non-containerized Kafka server without Kubernetes shows a significantly higher performance. Additionally, there was a latency of at least 1ms present while running the Kubernetes tests, while the non-Kubernetes server showed a latency of near 0ms.

Kubernetes used?	msg/s	Latency
No	8200	0 ms
Yes	5700	1 ms

9. **Effect of disk usage:**
   Is there any observable impact of using a topic filled with many messages:
   After filling a topic with data for around 24 hours with approximately 8GB of data transferred it was tested if any slowdown would occur.
   While the topic behaved when writing data onto it similarly in either a filled or empty state, a higher slowdown was observed when running a consumer. Additionally, the brokers restarted multiple times in the 24-hour window due to crashes, but due to their replication, no messages were lost. It is also important to note that while only around 8GB of messages were sent to the topic, the servers themselves were completely filled up with logs produced by Kafka causing a disk usage of almost 100% with the downside of generally slowing the servers down. Therefore, it is recommended to either reduce the rate of incoming messages, have a large enough disk to save the logs, or reduce the general retention time to the Kafka topic to a reasonable minimum time frame.

Data on topic	No Consumer running	Consumer running
Empty topic	2400 msg/s	2200 msg/s
8 GB written	2400 msg/s	1700 msg/s

One can see that the tests themselves were not always concerned with introducing a failure into the running system, as the failures themselves could just be broken down to “stop pod from executing”. The goal was also to run a general performance test of Kafka to uncover some causes of failures, which certainly appeared like the instability of pods with too few resources.

5 Conclusion and Outlook

With the resilience tests done on the deployed microservices, it is time to conclude this thesis and review everything. Here one can also find suggestions for additional tests and possible difficulties with Kubernetes.

Comparison with a similar benchmark:
First of, originally the plan was to compare the results of load testing Kafka with the results from a similar test done in [Kre14], but as those instructions to replicate the tests provided in their GitHub page were either outdated or incomplete it was not possible to run the tests on this specific server setup. Therefore, the values had to be taken from the source directly, where there might be a vast difference in the hardware.
The benchmarks themselves were not designed with Kubernetes in mind, but rather using the built-in functionalities of replicating a Kafka server. The goal was to compare if the observed behavior of Kafka during the resilience tests will also occur in a normal Kafka installation, in order to either validate the test results or gain additional insight into a replicated microservice. Without being able to replicate those tests the numbers in the article could not be verified, but one can compare the behavior of Kafka to the local tests.
The slowdown observed when using multiple replicas of a topic also occurred, but while there also appeared a slowdown when running a consumer or multiple producers in Section 4.2 the Kafka instance running in the article showed a linear increase in throughput with more producers or consumers. The throughput may have not increased due to some bottleneck the testbed that causes a single producer or consumer to already occupy all available resources, but the exact reason is unknown.
Either way, those tests still provided some ideas what to look out for when testing Apache Kafka and may spark the interest of the reader to further look into optimizing Kafka in a highly scaled environment.

Kafka testing and deployment on Kubernetes:
Another thing regarding Kafka, while the resilience and loads tests were performed using JMeter, the Kafka Plugin for JMeter turned out to be rather basic with not much room to tweak any values like the batch size of messages or how the connection should be handled. In the final stage of this thesis the Plugin “PepperBox” was considered as an alternative, as it provided more configuration details regarding Kafka and a generally better performance. Sadly the Plugin was discovered so late

into the thesis that there was no time to completely redo every test with the new setup. Therefore it may be recommended to reconstruct the tests found here again with PepperBox.
As it was shown in Section 4.2, Kafka had a general decrease in performance when running on Kubernetes, even without any additional replicated topics. One should therefore consider if the added resilience and automatic maintenance of running Kafka in Kubernetes is worth the speed loss. Additionally, while an increased number of replicas might help to keep the data redundant in case of server failures, one would have to take even more performance losses. In my opinion, having a slower but more automated setup that provides resilience to the deployed software would be the better choice in most cases, unless one were to build a platform specifically with a high request throughput in mind.

Kubernetes' general experience:
While the whole setup of Kubernetes and the testbed went rather smooth, one of the main problems encountered during this thesis were errors resulting from wrong network configurations. Every time two components did not seem to function correctly between each other, there was always the question if the components themselves were not working or if there were some connection problems.
E.g., during the setup of the first mockup, an external database was to be used for data generation, but as the database was placed on a server with no possible route to the testbed, nothing worked until that networking flaw was discovered. Another example would be the Kafka service reachability of the Kubernetes cluster from the Jenkins server, while the service ran on the port of a single node. Every node needed to have their same respective ports opened to connect with the Jenkins server, even though there was never a connection established between Jenkins and the other two nodes.

Resilience with Kubernetes:
Nonetheless, Kubernetes did meet the expectations in terms of provided resilience and also surprised with a rather good usability. The resilience tests showed that Kubernetes upheld the definition of resilience “the ability to recover from failures”, during testing. Kubernetes will, however, not try to prevent those failures in the first place, as their root cause may be found in the microservice itself or just the server missing resources to meet a specific workload. While an increased number of pod replicas may shorten the downtime between failures or even nullify them due to a better load balance between servers, are those aspects of scalability and not resilience.

Kubernetes regarding scalability:
While at the topic of scaled software, it may also be interesting to see how Kubernetes can handle things when scalability, rather than resilience, is at the center of attention. For this thesis the software was only ever deployed on three different nodes, which can be considered a rather minimalistic setup.
While it was shown that Kubernetes can keep a system resilient and running at that level, what would happen if the replication number increased tenfold, or if a sudden increase in the workload means that the software has to scale quickly during runtime? Could Kubernetes keep up, or is there a certain point where maybe the whole system gets unstable and everything fails? Additionally, one could use the scalability server setup to create a better stress test than used in this thesis, as only a single server had to create the workload. But with more servers provided, one could easily scale the load generation alongside the microservices to reach a critical mass of requests per second.

Kubernetes as a DevOps tool:
Either way, Kubernetes turned out as a great tool to integrate in a DevOps pipeline, with that development style also being one that was looked at during this thesis. The ways Kubernetes can be automated and integrated into a platform like GitLab, can really lift some work from the developers and make the development process more efficient. While for this thesis there were only deployments of externally written services, there was still some insight gained into the available tools for continuous integrations with Kubernetes. Those tools may not be interesting enough for research, but are definitively something that should be considered by every developer that wants to deploy their projects.
It seems that a pipeline with more automated tools is, in the end, more efficient than having multiple smaller pipelines. While one may want to approve each small phase of the deployment and testing manually, it seems that if one has correctly configured every tool to process the software written, there would be no need to approve those sub-phases. With a properly set up logging method, one can completely automate the process after the source code is pushed onto the repository via the DevOps pipeline.

List of Figures

Source Code Listing

Bibliography

[AD19] ARUNDEL, John ; DOMINGUS, Justin: *Cloud Native DevOps with Kubernetes*. O'Reilly Media, 2019

[AJBB$^+$19] AL JAWARNEH, Isam M. ; BELLAVISTA, Paolo ; BOSI, Filippo ; FOSCHINI, Luca ; MARTUSCELLI, Giuseppe ; MONTANARI, Rebecca ; PALOPOLI, Amedeo: Container Orchestration Engines: A Thorough Functional and Performance Comparison. In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)* IEEE, 2019, S. 1–6

[HRJ$^+$16] HEORHIADI, Victor ; RAJAGOPALAN, Shriram ; JAMJOOM, Hani ; REITER, Michael K. ; SEKAR, Vyas: Gremlin: Systematic resilience testing of microservices. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* IEEE, 2016, S. 57–66

[Jav18] JAVADEKAR, Shrinand: Kafka on Kubernetes: From Evaluation to Production at Intuit. (2018)

[Kre14] KREPS, Jay: *Benchmarking Apache Kafka: 2 Million Writes Per Second (On Three Cheap Machines)*. `https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines`, 2014. – Accessed: 2020-05-07

[LKK$^+$19] LWAKATARE, Lucy E. ; KILAMO, Terhi ; KARVONEN, Teemu ; SAUVOLA, Tanja ; HEIKKILÄ, Ville ; ITKONEN, Juha ; KUVAJA, Pasi ; MIKKONEN, Tommi ; OIVO, Markku ; LASSENIUS, Casper: DevOps in practice: A multiple case study of five companies. In: *Information & Software Technology* 114 (2019), 217–230. `http://dx.doi.org/10.1016/j.infsof.2019.06.010`. – DOI 10.1016/j.infsof.2019.06.010

[SD19] SHAH, Jay ; DUBARIA, Dushyant: Building modern clouds: using docker, kubernetes & Google cloud platform. In: *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)* IEEE, 2019, S. 0184–0189

[WFW$^+$19] WIEDEMANN, Anna ; FORSGREN, Nicole ; WIESCHE, Manuel ; GEWALD, Heiko ; KRCMAR, Helmut: Research for practice: the

DevOps phenomenon. In: *Commun. ACM* 62 (2019), Nr. 8, 44–49. `http://dx.doi.org/10.1145/3331138`. – DOI 10.1145/3331138