BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CONFIGURABLE PARALLEL EXECUTION OF SYSTEM TESTS WITHIN THE STRIMZI PROJECT

KONFIGURATEĽNÁ PARALELNÁ EXEKÚCIA SYSTÉMOVÝCH TESTOV V RÁMCI PROJEKTU STRIMZI

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MAROŠ ORSÁK**
AUTOR PRÁCE

SUPERVISOR **Doc. RNDr. MILAN ČEŠKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2022

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2021/2022

Master's Thesis Specification

Student: **Orsák Maroš, Bc.**
Programme: Information Technology and Artificial Intelligence
Specialization: Software Verification and Testing
Title: **Configurable Parallel Execution of System Tests within the Strimzi Project**
Category: Software analysis and testing
Assignment:

1. Get acquainted with Apache Kafka and Strimzi project to understand distributed systems deployed on top of Kubernetes.
2. Study current system testing of Strimzi project, understand the architecture, principles of execution on different environments and theory of parallelism.
3. Identify bottlenecks related to testing the Strimzi system tests.
4. Design and implement a parallel approach of execution tests that will solve problems from the previous point.
5. Verify the implementation part to find the ideal configuration for parallel tests execution on different sizes environments.
6. Evaluate and summarize your findings and successfully contribute to the implementation into the Strimzi organization.

Recommended literature:

1. Stopford, Ben. *Designing Event-Driven Systems*. O'Reilly Media, Inc. 2018.
2. Narkhede, Neha, Gwen Shapira, and Todd Palino. *Kafka: the definitive guide: real-time data and stream processing at scale*. O'Reilly Media, Inc. 2017.
3. Seymoure, Mitch. *Mastering Kafka Streams and ksqlDB.* O'Reilly Media, Inc. 2021.
4. Poulton, Nigel. *The Kubernetes Book*. Amazon, 2021.
5. Pacheco, Peter. *An introduction to parallel programming*. Elsevier, 2011.

Requirements for the semestral defence:

- Items 1, 2, 3 and the initial proposal for the fourth item.

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

Supervisor: **Češka Milan, doc. RNDr., Ph.D.**
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2021
Submission deadline: May 18, 2022
Approval date: November 3, 2021

Abstract

In recent years, many companies have adopted Kubernetes and the microservices architecture it enables. This technology was opened up many new possibilities not just for large companies, but also for small software developers. Kubernetes is a container-orchestration system and recently a new concept has emerged arround how to orchestrate the containers more efficiently – the Operator pattern. One such operator is developed and maintained under an open-source project called Strimzi. The Strimzi project gathers together several tools, which take care of the deployment of Apache Kafka on Kubernetes. Since Kafka is a complex, horizontally scalable, distributed system, you can imagine that its installation is a relatively complex action. Therefore, one of the biggest challenges of using Kubernetes is how to effectively and quickly test projects such as Kafka and Strimzi and at the same time verify integration with other similar products. The resources needed by Kubernetes are much more demanding compared to the deployment of Kafka on virtual machines or typical container instances. To tackle this problem, we adopt the principles of parallel execution and created a mechanism within Strimzi system tests, which runs tests in parallel against only a single Kubernetes cluster. Furthermore, we proposed a brand new architecture for the end-to-end tests. The improvements aim at *scalability* and *reduction of execution time*. Through several experiments, this paper shows that proposed mechanism with different configurations of the Kubernetes cluster (including *number of Kubernetes nodes*, *number of tests and suites executed in parallel*) significantly accelerated execution of the tests.

Abstrakt

V posledných rokoch mnoho spoločností prijalo Kubernetes a architektúru mikroslužieb, ktorú umožňuje. Táto technológia otvorila nové možností nielen pre veľké spoločnosti, ale aj pre malých vývojárov softvéru. Kubernetes je systém riadenia kontajnerov a nedávno sa objavil nový koncept, ako efektívnejšie organizovať kontajnery – vzor operátora. Jeden takýto operátor je vyvinutý a udržiavaný v rámci open-source projektu s názvom Strimzi. Projekt Strimzi spája niekoľko nástrojov, ktoré sa starajú o nasadenie Apache Kafka na Kubernetes. Keďže Kafka je komplexný, horizontálne škálovateľný, distribuovaný systém, viete si predstaviť, že jeho inštalácia je pomerne zložitá akcia. Preto jednou z najväčších výziev používania Kubernetes je, ako efektívne a rýchlo otestovať projekty ako Kafka a Strimzi a zároveň overiť integráciu s inými podobnými produktmi. Zdroje, ktoré potrebuje Kubernetes, sú oveľa náročnejšie v porovnaní s nasadením Kafka na virtuálne stroje alebo typické inštancie kontajnerov. Aby sme tento problém vyriešili, prijali sme princípy paralelného vykonávania a vytvorili mechanizmus v rámci systémových testov Strimzi, ktorý paralelne spúšťa testy iba proti jedinému klastru Kubernetes. Okrem toho sme navrhli úplne novú architektúru pre end-to-end testy. Vylepšenia sú zamerané na *škálovateľnosť* a *skrátenie času vykonávania*. Prostredníctvom niekoľkých experimentov táto práca ukazuje, že navrhovaný mechanizmus s rôznymi konfiguráciami klastra Kubernetes (vrátane *počet uzlov Kubernetes*, *počet paralelne vykonávaných testov a sád*) výrazne urýchlil vykonávanie testov.

Keywords

Strimzi, Kubernetes, Orchestration, Clustering, Azure, Openstack, AWS, Apache Kafka, Distributed systems, middleware, end-to-end tests, paralelism, multi-threaded execution, race condition, synchronization, scalability, operators

Klíčová slova

Strimzi, Kubernetes, Orchestrácia, Klastering, Azure, Openstack, AWS, Apache Kafka, Distribuované systémy, middleware, systémové testy, paralelizmus, multi-vláknové vykonávanie, súbeh, sychronizácia, škálovateľnosť, operátory

Reference

ORSÁK, Maroš. *Configurable parallel execution of system tests within the Strimzi project.* Brno, 2022. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Doc. RNDr. Milan Češka, Ph.D.

Rozšířený abstrakt

V dnešnej dobe sa čoraz častejšie stretávame s paralelnými programami. Tucet programov, ktoré boli napísané typickým spôsobom pre jednojadrové systémy, nedokáže využiť prítomnosť počítačov s viacerými jadrami. Keď sme chceli urýchliť riešenie problémov, chceli sme vytvoriť niečo, čo by eliminovalo náš čas na výpočty. Tak sme vynašli počítač, ktorý na začiatku nevedel relatívne nič robiť. To všetko sa však po pár rokoch zmenilo a počítač vyriešil problémy, ktoré človeku zabrali veľa dní. V súčasnosti žijeme v dobe, kedy počítače výrazne skrátili čas vykonávania riešením rôznych problémov pomocou paralelizmu.

Pred niekoľkými rokmi *Google* vydal technológiu, ktorá definovala a zmenila našu perspektívu nasadzovania a správy aplikácií. Túto revolúciu spôsobil iteratívny sled malých krokov (t.j., fyzická, virtuálna a kontajnerová éra). *Kubernetes* [1, 17] je systém na správu kontajnerov a v ďalšej mini iterácii priniesol nový koncept, ako efektívnejšie organizovať kontajnery – *vzor operátora* . *Vzor operátora* má za cieľ zachytiť, ako rozšíriť a implementovať úlohy automatizácie nad rámec *Kubernetes*. Jeden takýto operátor je vyvinutý a udržiavaný ako súčasť projektu s otvoreným zdrojom s názvom Strimzi [7, 6]. Projekt *Strimzi* spája niekoľko nástrojov, ktoré sa starajú o nasadenie Apache Kafka [12, 19, 11, 2] na Kubernetes. Komplexnosť, horizontálna škálovateľnosť a distribučný systém; sú všetky atribúty *Apache Kafka*. Žiaľ, tieto atribúty robia systém mimoriadne zložitou entitou na overenie. Preto je jednou z najväčších výziev používania *Kubernetes* efektívne a rýchle testovanie projektov ako *Kafka* a *Strimzi* pri overovaní integrácie s podobnými produktmi. Čo sa týka zdrojov potrebných na nasadenie *Kafka* na virtuálne stroje alebo kontajnery, je relatívne jednoduché porovnať nasadenie Kafku na *Kubernetes*. Napriek tomu to spôsobuje časové problémy pri testovaní nášho projektu *Strimzi*. Na vyriešenie tohto problému sme prijali princípy paralelného vykonávania a vytvorili mechanizmus v rámci *Strimzi testov systému*, ktorý spúšťa testy paralelne iba proti jednému klastru *Kubernetes*.

Súvisiaca práca sa zameriava na zlepšenie celkového času overenia produktu Strimzi. Niekoľko vydaní Strimzi nám dáva empirické poznatky, že testovanie pomocou sekvenčného výpočtového modelu bolo extrémne pomalé. Okrem toho produkt obsahuje asi pätnásť najkritickejších možných kombinácií nasadenia produktu, z ktorých každá trvá viac ako šesťdesiat hodín. Tento sekvenčný výpočtový model nie je odporúčaným kandidátom na overenie takého množstva nasadení.

Napriek tomu, ako súčasť tohto úsilia o coarse-grained paralelizmus pri vykonávaní viacerých nasadení produktov, čiastočne urýchlil celkový výpočet. Tento prístup však nie je horizontálne škálovateľný kvôli našim cloudovým službám, ktoré poskytujú zdroje (t.j., bare metal, virtuálne stroje, kontajnery). Dostali sme sa preto k poslednej príležitosti na zlepšenie výpočtov pomocou vertikálnej škálovateľnosti zdrojov (t.j., pamäte, centrálnych procesorových jednotiek), ktoré nám cloudové služby ponúkajú. Tieto informácie nás motivovali navrhnúť a implementovať mechanizmus fine-grained paralelizmu v našom testovacom rámci.

Experimenty, ktoré sme vykonali nad danou implementáciou ukázali celkové zlepšenie výpočtového času na viacerých podmnožinách testovacích prípadov. Napríklad u method-wide paralelizácií sme mohli vidieť zrýchlenie z troch hodín na takmer dvadsať minút pri využití 12 vlákien (jednalo sa však o podmnožinu testov, ktoré všetky podporovali parallelizmus). Zároveň sme tak mohli vidieť zrýchlenie pri využití reálnej produkčnej vzorky, ktorá obsahovala viac než tristo testov.

Configurable parallel execution of system tests within the Strimzi project

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Doc. Milan Češka, further information was provided by Ing. Jakub Stejskal. All the relevant information sources used during this thesis's preparation are appropriately cited and included in the reference list.

.....
Maroš Orsák
May 5, 2022

Acknowledgements

I would like to thank my supervisors, Ing. Jakub Stejskal, Doc. Milan Češka and Dr. Thomas Cooper for their time. This thesis is realized in cooperation with Red Hat Czech, s.r.o.

Contents

Chapter 1

Introduction

These days, we are increasingly encountering parallel programs. A dozen programs that have been written in a typical way for single-core systems cannot take advantage of the presence of computers with multiple cores. When we wanted to speed up problem-solving, we wanted to create something that would eliminate our time on calculations. Thus, we invented the computer, which knew relatively nothing to do at the beginning. However, after a few years, all this changed, and the computer solved problems that took a person many days. Nowadays, we live in a time when computers have significantly improved execution time by solving different problems using parallelism.

Several years ago, *Google* released a technology that defined and changed our application deployment and management perspective. An iterative sequence of small steps caused this revolution (i.e., physical, virtual and container era). *Kubernetes* [3, 1, 17] is a container management system, and in another mini-iteration, brought a new concept of how to organise containers more efficiently – the *Operator pattern*. *Operator pattern* aims to capture how to extend and implement automation tasks beyond *Kubernetes*. One such Operator is developed and maintained as part of an open-source project called Strimzi [7, 6]. The *Strimzi* project brings together several tools that take care of Apache Kafka [12, 19, 11, 2] deployment on Kubernetes. Complexity, horizontal scalability and distribution system; are all attributes of *Apache Kafka*. Unfortunately, these attributes make the system an exceedingly complex entity to verify. Therefore, one of the biggest challenges of using *Kubernetes* is effectively and quickly testing projects like *Kafka* and *Strimzi* while verifying integration with similar products (i.e., *Prometheus*[1], Grafana[2], Jaeger[3], Keycloak[4]). Regarding the resources required to deploy *Kafka* on virtual machines or containers, it is relatively simple to compare Kafka's Deployment on *Kubernetes*. Nevertheless, this causes time problems for our *Strimzi* project testing. To solve this problem, we have adopted the principles of parallel execution and created a mechanism within the *Strimzi system tests*, which runs tests in parallel against only one cluster of *Kubernetes*.

[1]Prometheus – open-source metrics-based project. Moreover, it provides an alerting system with incredible features, in case of interest https://prometheus.io/

[2]Grafana – open-source project, which primary responsibility is to show user interactive visualisation to track crucial parts of the system via the great user interface. (https://grafana.com/)

[3]Jaeger (Jaeger Tracing) – is a product which finds and helps troubleshoot problems in distributive systems. (https://www.jaegertracing.io/)

[4]Keycloak – open-source project for securing applications (authentication and authorization). (https://www.keycloak.org/)

Key Contributions Related work focuses on improving the overall verification time of a Strimzi product. Several releases of Strimzi give us empirical insights that testing using a sequential computational model has been extremely slow. Furthermore, the product contains about fifteen of the most critical possible combinations of product deployment, each of which lasts over sixty hours. This sequential computational model is not a recommended candidate for verifying such numerous deployments. An attentive reader could see the entire test time approaching one thousand hours, which is approximately one and a half month. Nevertheless, as part of this effort for coarse-grained parallelism in performing

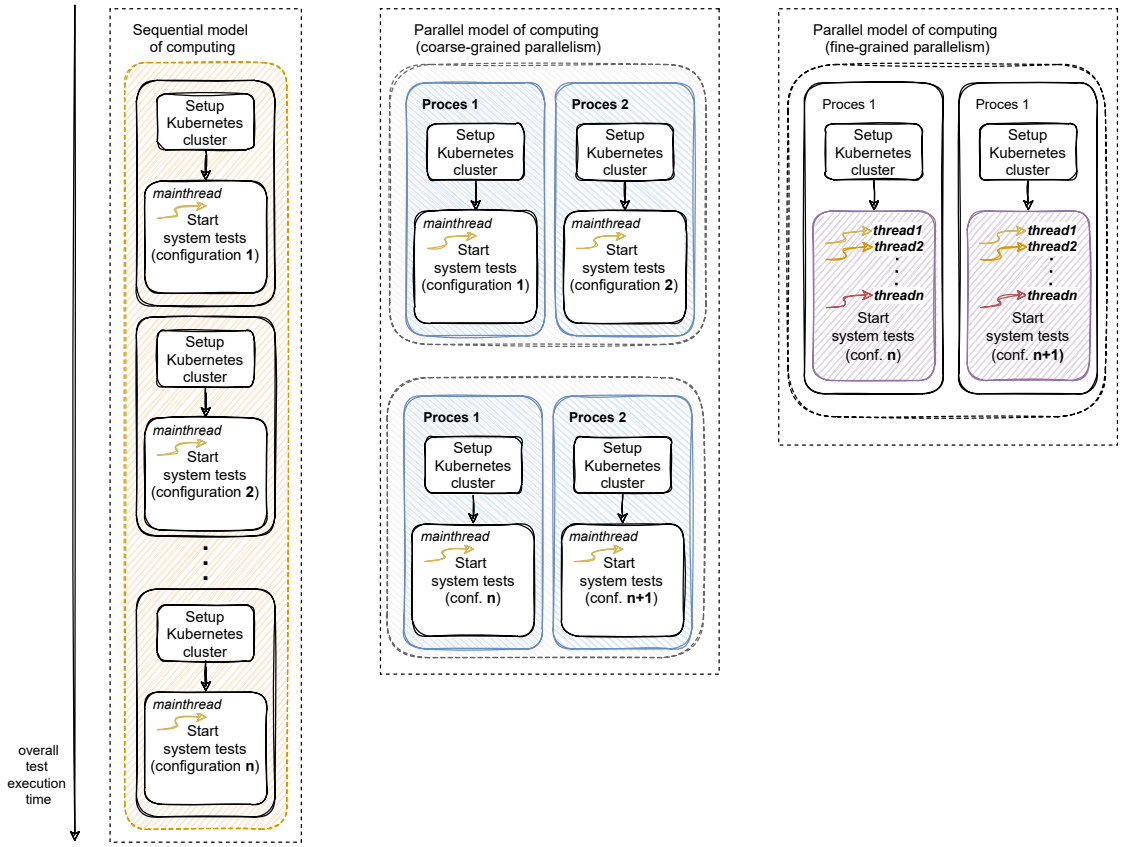


Figure 1.1: Evolution of our test framework execution

multiple product deployments, it partially accelerated the overall computation. However, this approach is not horizontally scalable due to our cloud services that provide resources (i.e., bare metals, virtual machines, containers). Therefore, we got to the last opportunity to improve the computation using the vertical scalability of the resources (i.e., memory, central processing units) that the cloud services offer us. This information motivated us to design and implement a mechanism of fine-grained parallelism in our test framework. Figure 1.1 shows the overall evolution of our test framework and summarises the previously mentioned sentences. The experiments on the implementation show that the given parallelization can significantly improve the execution time. The author contributed the given code to the open-sourced project Strimzi, available on Github[5], which also makes it possible to inspire

[5]Strimzi Github repository – https://github.com/strimzi/strimzi-kafka-operator

other *kube-native*[6] products to implement such solutions. The comprehensive benefit of this work is the acceleration of the verification process.

The structure of the diploma thesis The author decomposed the whole work into seven chapters together with an introduction. In Chapter 2, the reader learns about the theoretical background to understand the overall thesis (i.e., Kubernetes, Apache Kafka, Strimzi). Subsequently, we explain the fundamental concepts of parallelism (i.e., Amdahl's law (3.1), Shared memory (3.2), Process and Thread (3.3), Synchronisation (3.5) in Chapter 3. Chapter 4 presents bottlenecks in the current approach to testing the Strimzi product and proposes a brand-new computational architecture that solves many issues. Moreover, in Chapter 5, we describe the implementation of the proposed architecture. In the penultimate part of this thesis (Chapter 6), we summarise the results from many experiments with a deep analysis of the thesis implementation. Finally, we conclude the entire diploma thesis with the knowledge that has been acquired in Chapter 7.

[6]Kube-native – it is a product that has been moved from the standalone world to the Kubernetes world. Moreover, it provides a communication interface (i.e., Kubernetes REST API) with which it manages individual components (i.e., Apache Kafka is a standalone application, and Strimzi is a *kube-native* product because it encapsulates Apache Kafka and provides a communication interface for the user.

Chapter 2

Preliminaries

This chapter provides the fundamentals of the technologies used across the whole thesis. Notable technologies used are Kubernetes[1], Apache Kafka[2] and Strimzi[3], which are described in details in the following sections. Note that high-level descriptions of these technologies were already published in bachelor's thesis [13] written by the same author as this thesis. In this chapter, the author aims to explain the technology in more technical depth. Furthermore, some ideas related to Kubernetes were taken from the *The Kubernetes book* [17]. Section 2.4 describes the *e2e (end-to-end)* Strimzi tests that run on the top of the Kubernetes cluster. Also note that author described this topic in series of blogs posts *Introduction to system tests* [15] and *How system tests work* [14].

2.1 Kubernetes

In 2014, Google came up with a new concept of container management. This concept has opened the door for many products to simplify their management of applications deployments. This technology defined a set of primitives, which collectively provide mechanisms that deploy, maintain and scale applications based on CPU, memory, or custom metrics. Moreover, it does not create a virtual machine but uses the kernel of the physical computer. Also known as the lightweight approach compared to virtual machines. Kubernetes follows the leader and follower architecture. The leader node controls Kubernetes resources, and the follower node is responsible for resource creation. The definition of these resources is given in a declarative way using YAML[4] formatted files.

2.1.1 History

So far, we have developed four approaches to managing applications on the top of the operating system [3]. In each direction, we have eliminated certain disadvantages based on empirical knowledge.

1. **Running a physical machine** — The first phase of how to deploy applications was to execute the program on the physical computer. This approach was not as practical

[1] Kubernetes – orchestration system created in 2014 by Google (https://kubernetes.io/)

[2] Apache Kafka – distributed messaging system initially created in 2013 by LinkedIn (https://kafka.apache.org/)

[3] Strimzi – collection of operators for deploying and managing Apache Kafka on top of the Kubernetes (https://strimzi.io/)

[4] YAML – human-readable serialization format (https://yaml.org/)

as it seemed at first. The main issues were scalability, hardware management, security, and price. Besides that, sharing memory between five running applications in an identical environment is not ideal. Moreover, to isolate the applications from one another, one has to buy five physical servers, significantly increasing costs.

2. **Virtualisation** — The next phase has solved problems like scalability, security, and also price. This allows an application to run on a single machine without sharing memory, which means it is isolated and encapsulated from other applications. Furthermore, one can run many of these virtual machines on a single physical server, and the only limitations are the server resources. These virtual machines are independent of each other, and therefore the security is much higher. However, resource consumption is still high since each virtual machine includes an entire operating system. At the same time, the management of these entities is not accessible if we imagine production with hundred virtual machines. Another limitation is that sometimes applications need to share information, and the intense isolation of VMs makes this problematic.

3. **Containerisation** — In the last phase, containerisation is considered a lightweight alternative to virtualisation. The difference between these two phases is that virtualisation is using hypervisors[5] to manage all the virtual machines which have operating systems. The container shares the operating system with the server. Similar to virtualisation, they have their filesystem, memory, and space. Containerisation has become the most popular technology due to the several benefits it offers:

   - **Isolation** – predictable application performance,
   - **Observability** – gathering of information, providing metrics, logs,
   - **Portability** of distribution in the cloud and OS – runs on basically all available OS, public clouds, and so on,
   - **Agile approach** – easy to create and manage smaller container images instead of using virtual machine images, which are usually much larger.

   Unfortunately, containerisation still has several shortcomings, such as managing more running containers simultaneously and making debugging challenging.

4. **Container orchestration** — The phase of the present. Let us imagine a situation where we run several containers and want to know the container's current state or metadata information. It is not straightforward to get such information because we have to look at each running container separately and analyse it. Kubernetes brings us a solution to this problem. While in containers, we have to search each one individually, so in Kubernetes, we all have it simultaneously. Figure 2.1 illustrates and summarises the phases of managing an application on top of the operating system, starting in 1950 when the first computer, ENIAC, was assembled—moving to the virtualisation era, which started in the early 70s. IBM Cambridge Scientific Center began the development of CP-40, the first operating system that implemented complete virtualisation. However, what is very important to note is that the first known virtualisation software was VMware, created in 1997. Afterwards, the lightweight era came with an idea whose functionality was based on containerization [10]. Finally, we have a manager who takes care of the overall management of the individual containers

[5]Hypervisor - It is a software that manages virtual machines, for instance, VMware or VirtualBox.

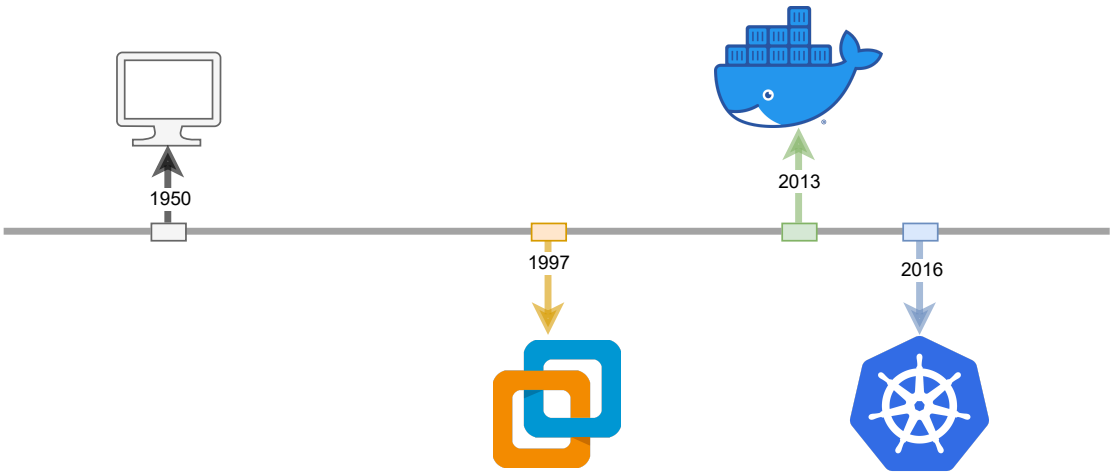


Figure 2.1: Evolution of virtual technologies

and guarantees their reliability, scales them effectively and more. This is what we call a container orchestration system [1]. It has the following properties:

(a) Deployment, StatefulSet, ReplicaSet, and Custom resource definitions (CRDs).
(b) Service and Load balancing (Service discovery).
(c) Storage (Storage orchestration).
(d) Secrets (Secret and configuration management).

2.1.2 Essential components of Kubernetes

The Linux hosts can be virtual machines, bare-metal servers in the data center, or private or public cloud instances. Production environments typically have more than one master node running because of the need for High Availability[6]). Kubernetes services from the

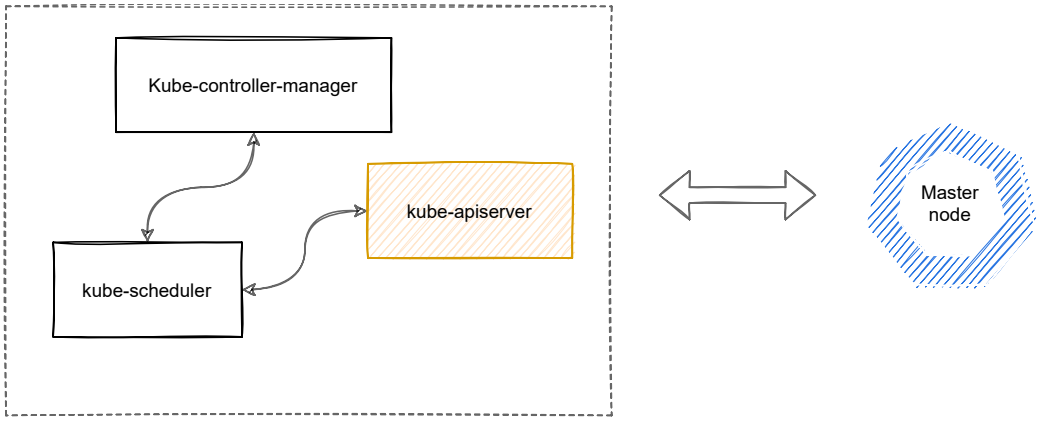


Figure 2.2: Representation of the Master node

most significant cloud providers, such as Azure Kubernetes Service (AKS), Amazon Elastic

[6]High availability (HA) – is the characteristic of the system to run without failing for some period of time.

Kubernetes Service (EKS), and Google Kubernetes Service (GKS), have five controller nodes, which are replicated in case of any failure. The master node 2.2 contains several components such as *kube-controller-manager*, *kube-scheduler* and *kube-apiserver*. These components are also called the „control plane“. The *kube-controller-manager* takes care of all controllers where each of these controllers runs as a separate process. The *Node controller's* responsibility is to control and respond to the current status of the node. In other words, do a health check of nodes. There is also the *Endpoint controller* for Service and Pod objects, *Job controller* for Job objects, etc. All these controllers follow algorithm 1.

Algorithm 1 Generic algorithm for each Kubernetes controller

```
1: desired_state ← controller.obtain_desired_state()
2: while True do
3:     desired_state ← controller.obtain_desired_state()
4:     current_state ← controller.observe_current_state()
5:     if current_state ≠ desired_state then
6:         controller.reconcile()
7:     desired_state ← current_state
```

The *kube-apiserver* works like the controller of API calls and communicates with the *kube-scheduler*. It makes sure that every created Pod is assigned to run there. It is worthwhile to mention that we also have a component called *etcd*, which works as a backup for cluster data. Slave node components 2.3 such as *kubelet* have taken care of containers running inside the Pod. *Kube-proxy*, which reflects all the services defined in the kube-apiserver. In the following Figure 2.4, one can see relation between master and slave nodes.

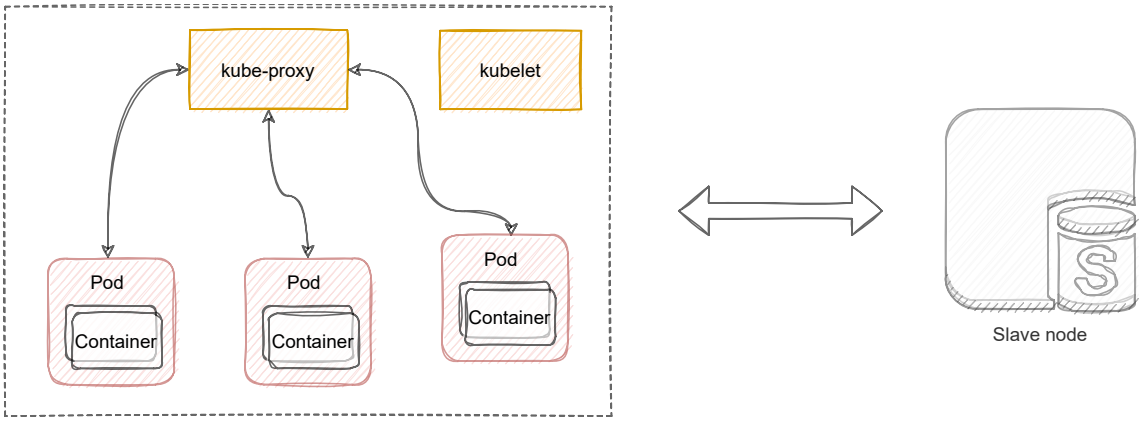


Figure 2.3: Representation of the Slave node

2.1.3 Common objects

1. **Pod** – is the atomic unit of Kubernetes. For instance, in the VMware environment, the atomic unit is a virtual machine, and in Docker, it is a container. The term Pod originated from the Docker logo. If we think about it, Docker has one whale on his logo, and we call a group of such whales Pod or, in other words, Pod of whales. Deductively, we can find out the property of the Pod, that is, that one or more containers can run in it. These containers share storage, network, and specification

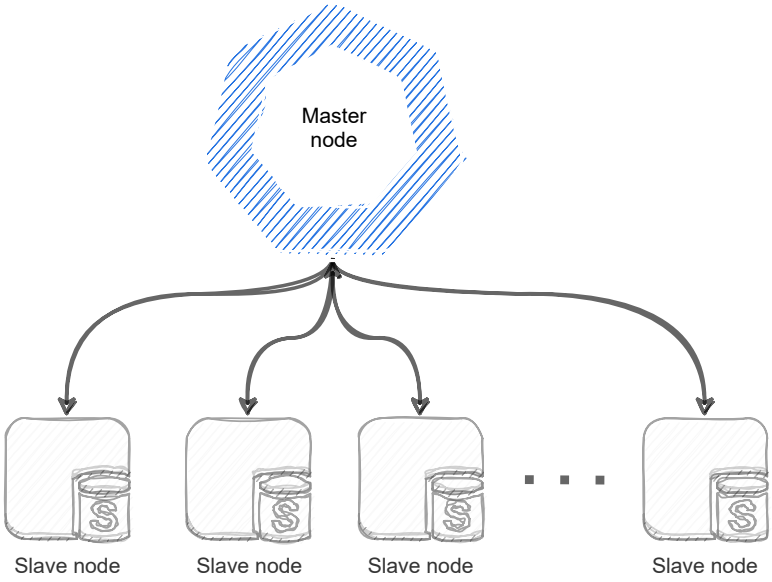


Figure 2.4: Relation between master and slave nodes

of how to run the container. If the container wants to communicate with the other container, this can be achieved using the localhost interface. One of the disadvantages of these resources is their lifecycle. If the Pod crashes or is deleted, it will no longer be possible to copy this Pod. Instead, Kubernetes will create a new Pod with a unique ID and a new IP address assigned.

2. **Service** – represents how particular components communicate. Services provide reliable networking for a set of Pods. If Pod fails and Kubernetes creates a new Pod, its IP address is changed. Moreover, operations like scaling up or scaling down do the same. This is where Services come into play. They provide reliable names/alias and IP addresses. Furthermore, the Kubernetes service has its DNS name and port. It is a stable network abstraction, which provides TCP and UDP load-balancing across a dynamic set of Pods. By default, a service in Kubernetes has a type of *ClusterIP*, which means that communication can be established only inside of the Kubernetes cluster. The way one can expose an application outside of the cluster is to use the following type of service which Kubernetes offers:
    - **nodeport** – exposes the service to be accessible via node IP with a specific port. For instance, one wants to expose an HTTP server to be publicly accessible on a specific port.
    - **load balancer** – exposes the service externally using a cloud provider's load balance. The load balancer is shown in the definition. *.status.loadBalancer* field, where one can find a real IP address. For example, if demands are high and one wants an application that requires more ports on specific IPs, then the usage of load balance is a wise choice.
    - **ingress** – the previously mentioned types of how to expose a service were service types, but ingress is an entry point for the cluster. *It lets you consolidate your routing rules into a single resource as it can expose multiple services under the same IP address* [5].

3. **Namespace** – this concept of namespaces was introduced in order to run numerous virtual clusters inside a physical one. *It is great for applying different quotas and access control policies. On the other hand, it is not suitable for strong workload isolation.* By default, Kubernetes starts with three initial namespaces:
   - **default** – the objects which do not have another namespace belong to the default namespace,
   - **Kube-system** – namespace for objects created by the Kubernetes system, i.e. Pods, Kube-proxy, Kube-DNS. Furthermore, the service account in this namespace is used to run the Kubernetes controllers.
   - **Kube-public** – *this namespace is created automatically and is recognizable by all users (including those not authenticated). In other words, there is a situation we need to have shared resources across the whole cluster; then we have to make sure that these resources are inside this namespace* [4]
4. **Volume** – is data storage. The Volume is a separated object which binds to a Pod. The main ideas behind volumes are: at first, assume a scenario when Pod crashed, and the application will lose all its data, and one would like to retrieve it secondly if one wants to share the same data between more Pods. The answer to these problems is the *Kubernetes Volume abstraction.*

2.1.4 Controllers

1. **ReplicaSet** – is the controller that is responsible for the correct number of running Pods. Furthermore, ReplicaSet plays a significant role in the Deployment controller, supplying a self-healing mechanism and scale operations. The self-healing mechanism guarantees that the Pod is running, and in the event of any error or termination of the Pod, a new one will be created immediately. Scale operations guarantee an easy way to increase the number of Application Pods if necessary in a heavy load. The same applies even if the given number of Pods is already high (we use a scale-down operation). ReplicaSet also has responsibility for the Rolling Update and Rollback operations available to Deployment.
2. **Deployment** – it is one of the most widely used application management controllers in the Kubernetes environment.

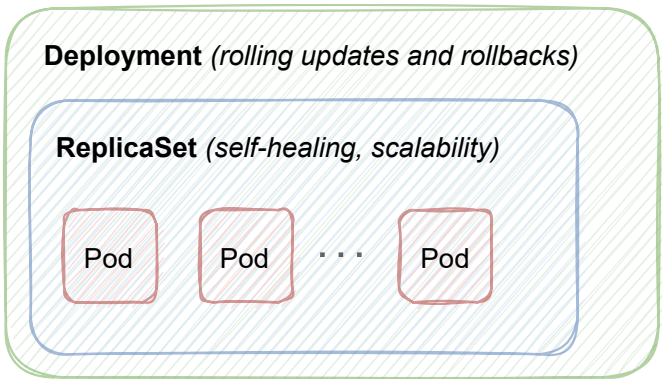


Figure 2.5: Hierarchy of Deployment, ReplicaSet and Pod inspired by The Kubernetes Book [17]

Based on our knowledge, the skilful reader will realise that Pod as an atomic unit will not be sufficient. This is mainly since Pod has no self-healing mechanism, does not support scale operations; Rolling Update[7] or Rollback. Deployment has all these features at its disposal. Importantly, this controller manages the ReplicaSet, which manages self-healing and scale operations. This means that the ReplicaSet checks whether the desired state is equal to the current state, such as the number of replicas being equivalent to the current state. Additionally, Deployment supplies the remaining properties, Rolling Update and Rollback. Since Deployment is a fully-fledged object in the Kubernetes API similar to Service, Pod, or Volume, that gives us the ability to define such an object in YAML files, such an object can then be edited, which will trigger a Rolling Update. Figure 2.5 shows us the hierarchy of mentioned the controllers.

3. **StatefulSet** – The last major controller is StatefulSet. This controller has many features in common with Deployment, such as the reconciliation loop described in 1, scaling operations, and a self-healing mechanism. The difference between Deployment and StatefulSet are as follows:

   - **storage** – with the Deployment controller, it is possible to specify PersistentVolumeClaim, which is shared between all Pod replicas. On the other hand, in the case of StatefulSet controllers, each Pod has its own PersistentVolumeClaim. For clarity, one can use Deployment in the case of a stateless application, where each node does not need a unique identity, and in the case of StatefulSet, one can use it in the form of databases (i.e., Cassandra, MySQL) where each node has its unique storage.
   - **unique identity to Pods** – in case of failure remains the same (Deployment will create a new Pod with a completely new name). Moreover, StatefulSet guarantees that Pods are created/deleted in order (Deployment does ensure order).
   - **scaling operation** – ensures that each new Pod is installed only after the previous one is ready and running. This process is repeated until we reach the number of replicas required. Figure 2.6 illustrates a scaling up scenario, where firstly *Pod_1* is being deployed and after a while when *Pod_1* is running and ready, the *Pod_2* is being deployed.

   In Figure 2.6, we see that architecturally StatefulSets has a different self-healing and scaling operations mechanism compared to the Deployment. In addition, Volumes play a significant role in the StatefulSet. When the Pod is created, the Statefulset immediately creates an associated volume and attaches this Volume to the Pod. This guarantees that the Pod can keep all its information even in an unexpected failure.

2.2 Apache Kafka

This section describes and explains the basics of the Apache Kafka system. The description is based on two books: *Designing Event-Driven Systems* [19] and *Real-Time Data and StreamProcessing at Scale* [12]. Moreover, the Kafka streams subsection is based on *Mastering Kafka Streams and ksqlDB Building real-time data systems* [11]. We also used Kafka's

[7] **Rolling Update** – is the process when one updates the Deployment configuration, and this update trigger replacements of the Pods with the new desired configuration

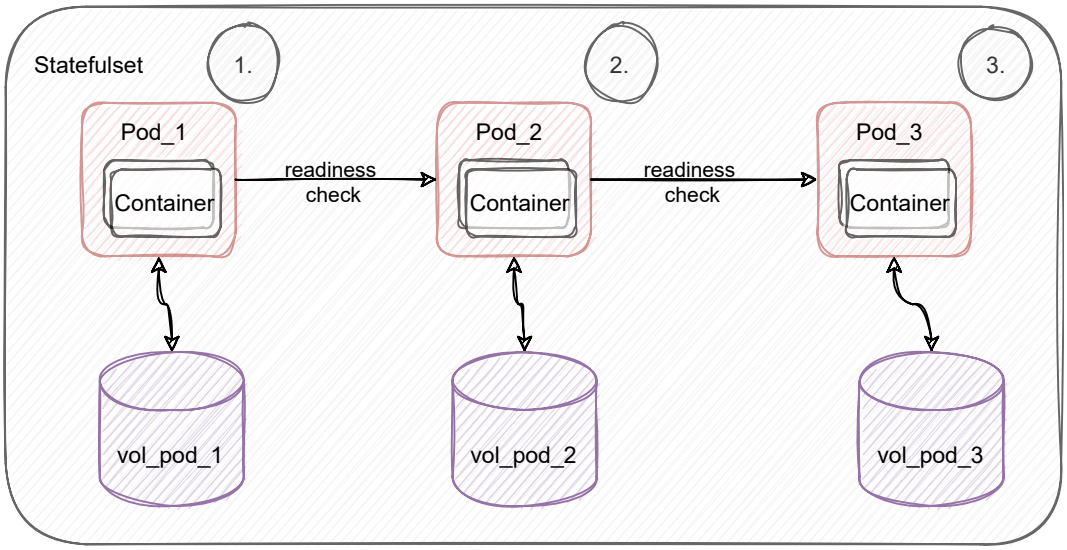


Figure 2.6: StatefulSet ordered creation of Pods

documentation [2] as the most up-to-date reference. In these books and documentation, a more detailed explanation of Kafka itself can be found.

Apache Kafka is an event streaming platform that offers many features like high performance, distribution, commit log service[8], and more. It offers a publish/subscribe system to record streams similar to a message queue or enterprise messaging system. Additionally, it stores record streams in a robust, fault-tolerant way. Kafka also creates real-time data flows that reliably capture data transferred between systems or applications. Kafka is widely used by many big companies like LinkedIn, Spotify, Netflix, and Uber.

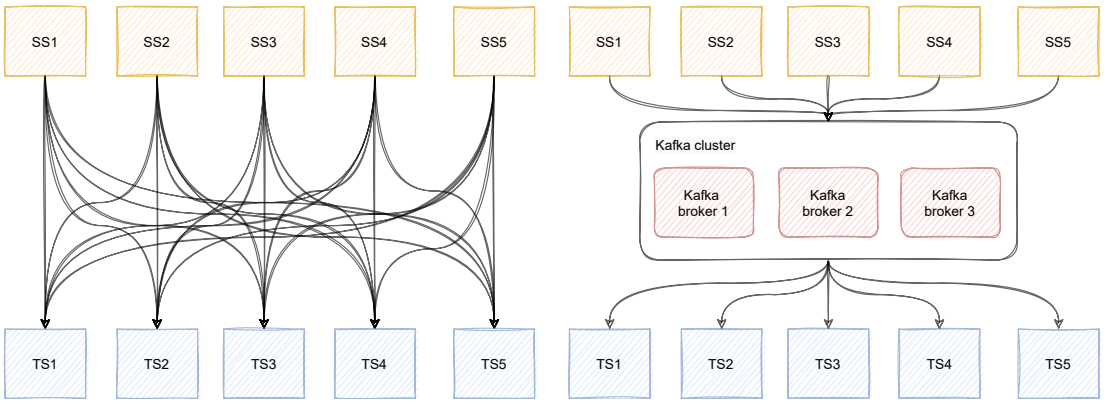
2.2.1 Motivation [13]

Companies had applications or systems that shared large amounts of data in the past. Usually, these applications would provide valuable information to another application. So, there was one source system and one target system. Nevertheless, what about adding more source and target systems? Assume an example where one has five source systems and five target systems. Each source system needs something from each particular target system.

The system without Kafka depicted in Figure 2.7a has twenty-five links, which is not scalable (quadratic complexity). That is one of the main reasons Kafka was invented. Let us illustrate the same example with ten systems and Kafka in the middle serving as Middleware[9], which is placed in the middle of these systems. Each source system only binds to the Kafka broker, and a single link delivers all data. One can see the updated system in Figure 2.7b.

[8] **Commit log** — is a type of data structure that stores ordered sequences of events.

[9] **Middleware** – Software, that acts as the middle man between two systems and guarantees interoperability between them.



(a) Source and Target systems without Kafka (b) Source and Target systems with Kafka

Figure 2.7: How to make system more efficient with Kafka

2.2.2 Fundamental concepts

In this subsection, we describe fundamental concepts of Apache Kafka such as Producer, Consumer, Kafka broker and Kafka cluster. The description is based on the *Real-Time Data and StreamProcessing at Scale* [12] and Kafka documentation [2].

1. **Kafka broker/cluster** – it is a server application that manages messages that are sent by producers and at the same time obtained by consumers. In other words, it takes care of storing the data and the order of the data. Sometimes, we can see a Kafka broker with Kafka server or Kafka node names. These names are synonymous with Kafka broker. Kafka broker was designed to be horizontally scalable to create a Kafka cluster (two and more Kafka brokers). Within a Kafka cluster, there is a single cluster controller. The cluster controller takes care of fundamental operations such as assigning partitions to brokers or monitoring for the failure of Kafka brokers. One broker in the Kafka cluster always owns the topic partition. This broker is called the leader of this topic partition. Of course, this topic partition can be replicated into several Kafka brokers, which will result in its replication and thus data redundancy. On the other hand, if the leader Kafka broker fails, the one who has the replicated topic partition will take control and become the new partition leader. Figure 2.8 illustrates this type of scenario, where two Kafka brokers shared data between each other and partitions of the topic are replicated.

2. **Producer** – is one of the types of clients that Kafka provides. They produce new messages that are sent to a specific topic. In general, the client does not need to know which partition it is necessary to send messages. It simply sends messages divided among several partitions. Thus, producers represent the entity that creates the data in the Kafka system. Kafka also provides the implementation of these clients in several languages such as Java, Go, C++, Python, and many others. Kafka also provides a higher-level abstraction, which means that it is no longer necessary to create the producers themselves, but those entities are encapsulated in the client. These are, for example, Kafka Streams for stream processing or Kafka Connect API for data integration.

3. **Consumer** – unlike a producer, a consumer or group of consumers tries to consume messages. It is necessary to specify the topic from which the consumer will read in the consumer configuration. However, the consumer can also read from a group of topics. The consumer maintains an internal offset value that represents a position from where the consumer should read the data from the topic. The method that consumers use to read messages is called polling[10]. The consumer group behaves as a single logical unit. Kafka does not support reading from one specific partition to more than one consumer simultaneously. The reason why this concept was created is based on a straightforward question - *How are we able to consume data concurrently?* Likewise, what is worth mentioning is that we *can not* have more consumers than partitions because, in that type of example, some of them are inactive. This concept differs from other messaging solutions and describes why Kafka is so flexible compared to traditional messaging based on AMQP protocols like ActiveMQ or RabbitMQ.

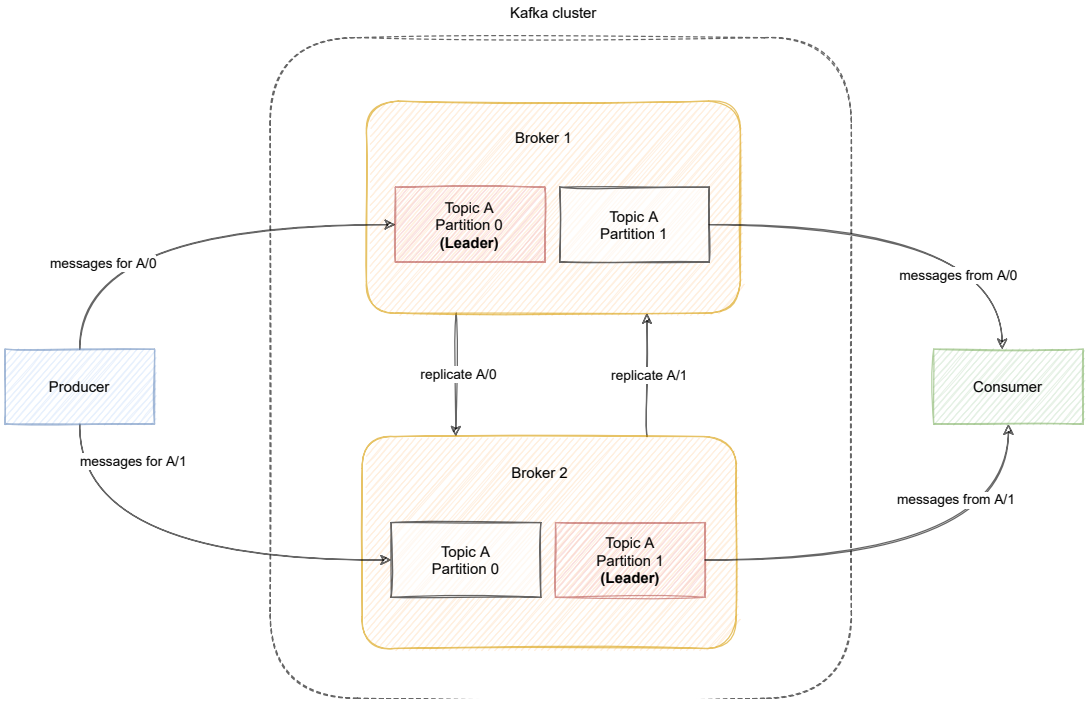


Figure 2.8: Kafka topic partition replication scenario in Kafka cluster inspired by *Real-Time Data and StreamProcessing at Scale* [12]

4. **Kafka Topic** – is not a simple concept and includes several parts such as the replication factor, partitions, and more. Kafka topic is equivalent to database table as one can see in the Figure 2.9.

   Messages are being stored on a specific topic. A replication factor is a number which defines how many replicas will be available on the other brokers from the Kafka cluster. Imagine the following scenario – we have a Kafka cluster with three Kafka brokers. We create a new topic with a unique name using an administration client. (In Section 2.3, we will talk about alternative ways of creating resources.) The question can be *what happens if we set higher replication factor then we have available Kafka*

[10] **Pooling** – periodic querying to the server, in that case, to the Kafka broker

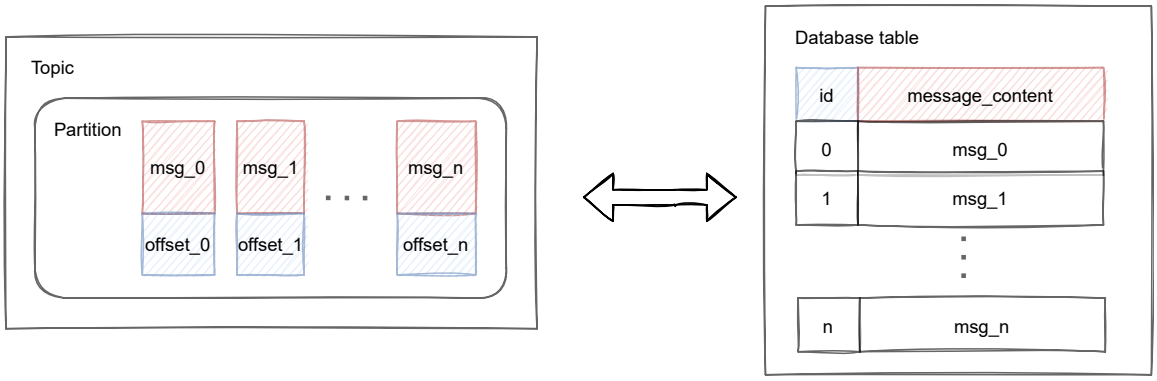


Figure 2.9: Equivalence of Kafka topic and database table

brokers. We are notified that the Topic can not be created because we do not have enough accessible Kafka brokers. More about this in **??**. Partitions are entities that split *KafkaTopic* into separate parts. It means that in each partition, we have different data; using this feature, we allow the consumer to fetch data in a concurrent[11] way. A partition contains offsets, which serve as ids for the specific messages. An *Offset* is an integer value assigned to each consumer indicating the following message, which will be read. Consider the scenario when we have one Kafka broker and one Topic with a hundred messages. According to *Offset* implementation, the maximum offset value is 100 because it reflects the position of the last message in the Topic. If we configure consumers to subscribe to that Topic, it uses the polling method and starts with offset zero. The first poll gets twenty messages, so offset moves on to nineteen. The Figure 2.10 illustrate this scenario. In general, we can understand offset as the message index.

Figure 2.10: Partition offset

2.2.3 Kafka Streams

It is a stream processing tool created by the Kafka community that does expose the low level of the Consumer API and Producer API. These client APIs are very flexible, and the user can create the data processing logic he wants. However, there is a tradeoff, and it is writing many lines of code. Unfortunately, we cannot classify these APIs as stream processing APIs because they do not contain primitives that would classify them there, such as *Local* and *Fault-tolerant* state and a set of transformers that work with data (a transformer is an operator that transforms data).

In 2016, Kafka introduced the *Kafka Streams API*, which solved these problems. Inexperienced users in Kafka Streams would think it is just a matter of sending messages to and

[11] Consumes more than one message at the specific period.

from Kafka. Instead, we can see that Kafka has a part of Producer and Consumer, where it offers a wide range of libraries for data transformation. Kafka streams also support two crucial operating characteristics:

1. **Scalability** – In Kafka Streams, the smallest unit of work is a single partition. If we want to scale the Kafka Streams application, we have to divide *KafkaTopic* into several partitions. Practically speaking, one uses the Kafka Streams API to deploy multiple instances of an application, each of which will handle a subset of the work. For illustration, one *KafkaTopic* has sixteen partitions, and it is up to us how we scale it. One scenario could be to deploy two instances, and each of them would trade eight partitions. Figure 2.11 shows example with three partitions.

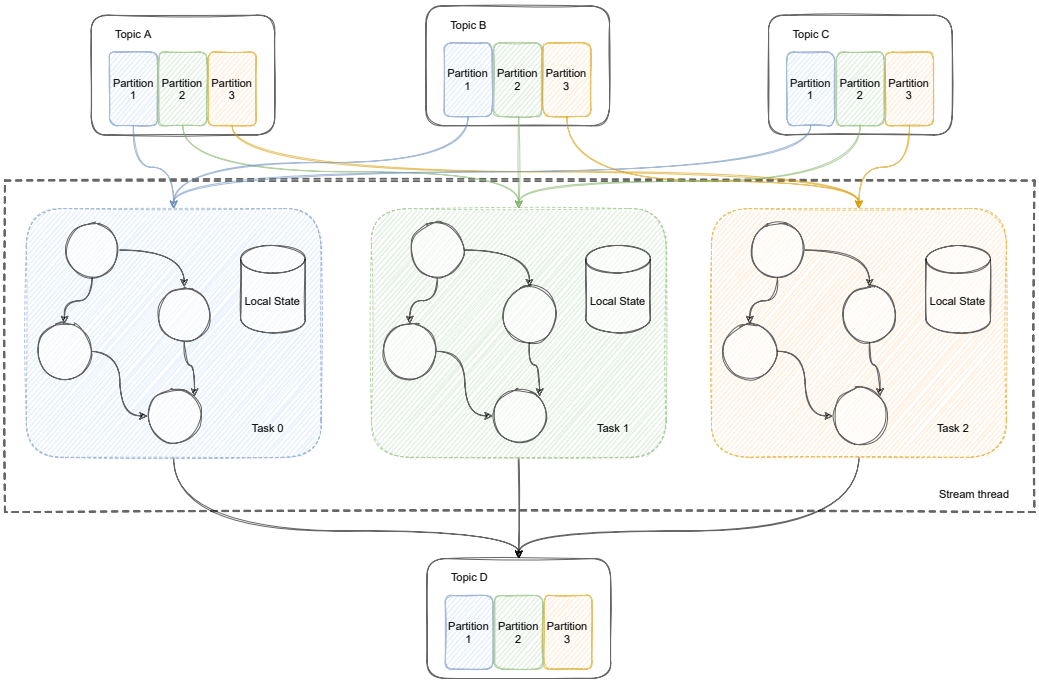


Figure 2.11: Kafka Streams with local state stores inspired by Kafka Documentation [2]

2. **Reliability** – If an error occurs on any node, Kafka automatically distributes the load to other nodes. However, we must realise that if the node that crashed is the last, we may lose the data if we do not use some Volume or other external storage. At the same time, when the node returns the given error is corrected, Kafka will rebalance again.

One of the main differences between other similar systems is the processing model that Kafka Streams offers. These systems, such as *Apache Spark Streaming*[12] or *Trident*[13], use micro-batching, which occurs very much in machine learning where work is divided into several batches. These groups are then loaded into memory and emitted at a pre-selected interval (typically $1s$ or less). Figure 2.12 shows a micro-batching strategy, where one can see that events are coupled into groups. By contrast, Kafka Streams offers us event-at-a-time processing, where events are processed as soon as they arrive. This approach gives us

[12] **Apache Spark Streaming** – is a extension of Spark API with many transformation methods.

[13] **Trident** – high-level abstraction for stream processing based on the Apache Storm. It provides multiple transformation methods such as filters, grouping, and aggregations.

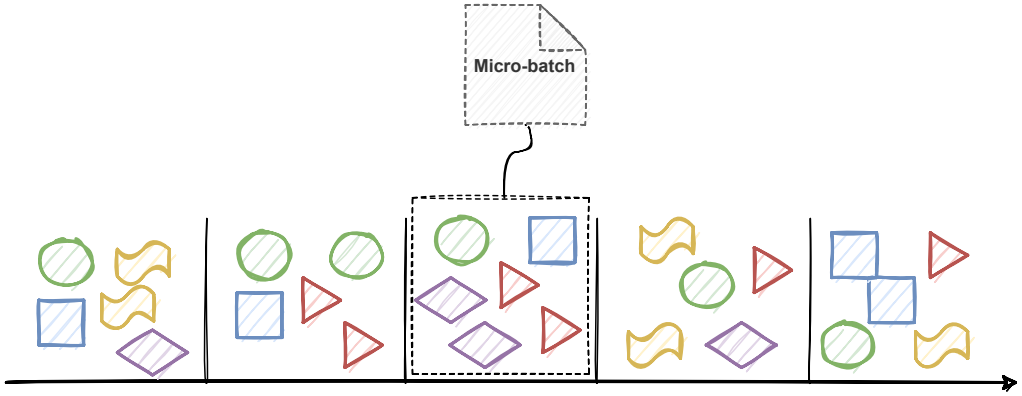


Figure 2.12: Micro-batching processing (typical for different systems) inspired by [11]

low latency and is considered true data streaming. Figure 2.13 illustrates the event-at-a-time processing strategy.

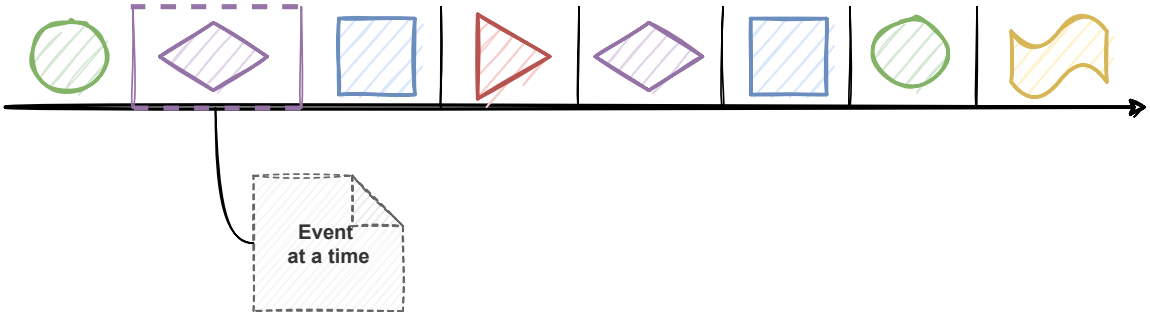


Figure 2.13: Kafka Streams uses event-at-a-time processing inspired by *Mastering Kafka Streams and ksqlDB Building real-time data systems* [11]

Kafka Streams is thus a set of libraries that offer developers incredible power over data processing. Additionally, it has a model of parallelism, where the smallest logical unit is partition. Easily scalable by either increasing or decreasing partitions, and lastly, Fault tolerance is rooted in Kafka itself (dependent on Topic replicas). This collection of characteristics makes it the perfect choice for today's data-intensive applications. These types of applications could be, for instance:

- email tracking, monitoring,
- chat infrastructure (Slack), virtual assistants, chatbots,
- machine-learning pipelines (Twitter),
- smart home (IoT sensors).

There are many such types of applications. However, what brings together all the examples is real-time data processing.

2.2.4 Kafka Connect

One of the most critical questions that every data engineer has is: „*How to move data from Kafka to a datastore or vice versa?*“. Moreover, how to create data pipelines that

connect several systems, for instance, by selecting data from Twitter and then sending it to Elasticsearch or other external storage. Of course, Kafka will play a middleware role in this data transfer. We can answer the previous question and solve the data integration problem thanks to the *Kafka Connect* component.

Kafka Connect offers many features that are transparent to the users. These include configuration, parallelisation, error handling, and much more. Moreover, for data integration, Kafka Connect offers two types of connectors. Connectors are already predefined templates. These connectors need metadata information to work. We give this connector information such as the names of one or more Topics to follow. In addition, these are attributes such as the connector class, the number of tasks executed in parallel, and the connector URL. The first type of connector is Kafka Connect Source, which obtains the data from the datastore. Information about what datastore and other metadata are provided in the connector configuration files. If the data in the datastore are changed, the data is automatically sent to one or more Topics. The second type is the Kafka Connect Sink, analogous to the Source connector. In the connector configuration, we define which datastore it should add data to and from which Topic it should monitor changes. When Topic changes his state, this data is automatically pushed into the given datastore. The simplest examples of connectors already mentioned above are the *FileSource* and *FileSink* connectors.

However, to properly understand Kafka Connect, it is necessary to know how the following fundamental mechanisms work:

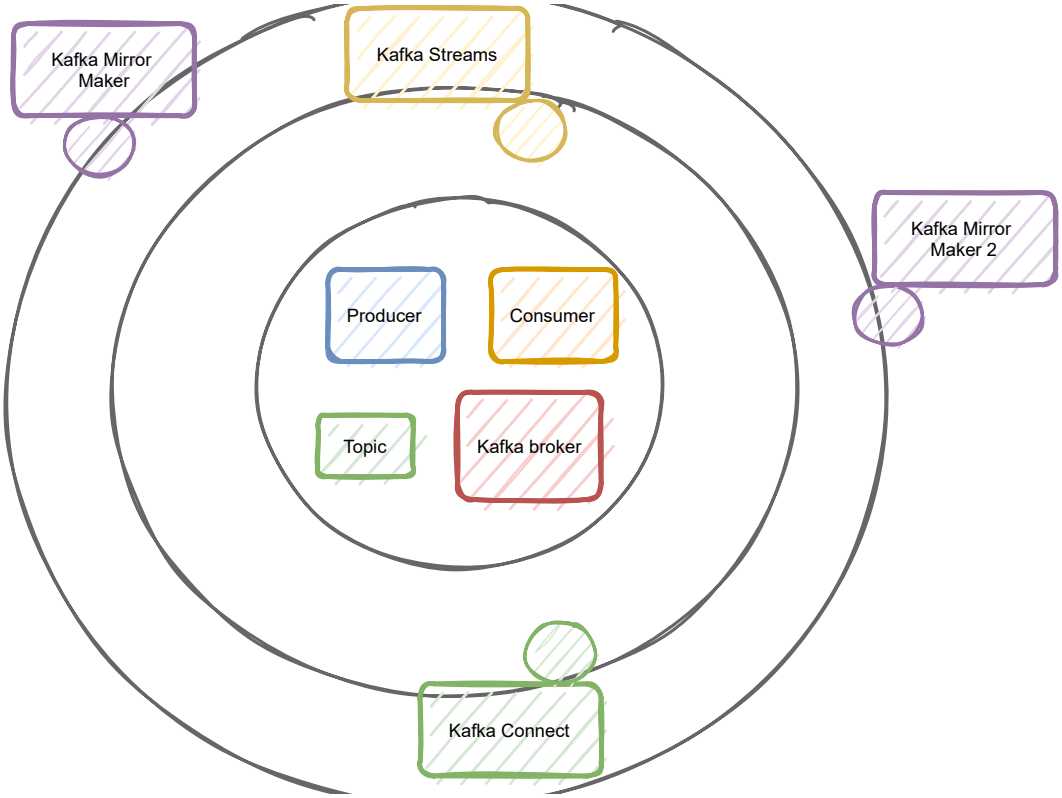


Figure 2.14: The entire Apache Kafka ecosystem.

1. **Connector** – As mentioned above, the connectors are used to transfer data to and from Kafka. Among the essential responsibilities of connecting connectors to a given datastore, it maps the data structure that the external storage has at its disposal and decides how many tasks (threads) will run simultaneously during the transformation.

2. **Worker** – This entity is responsible for the REST API available to Kafka Connect. They check REST API requests and respond accordingly. If a worker error occurs, the other workers in Kafka Connect will know this information as soon as possible and then perform rebalance and redistribute the work.

3. **Data model and converters** – Kafka Connect API contains endpoints of data objects and the scheme. These objects can be database tables, JSON, XML, and AVRO schemas. Converters transform this schema to a Connect Schema object. Subsequently, this Connect Schema object is sent to the target system. There are currently many such converters available.

All the mentioned Kafka components can be divided into three stages. The first milestone was the emergence of a new messaging system with basic functionality and no enterprise libraries. These included Kafka Broker, Topic, Consumer, and Producer components. The lack of libraries and the writing of vast amounts of code in data processing brought Kafka Streams. Kafka Connect solved data integration problems between other systems. Finally, the Kafka Mirror Maker 2 concept came along, which improved the Kafka Mirror Maker predecessor with many capabilities. It was a way to move data from one Kafka cluster to another. The whole Kafka ecosystem is not trivial. Figure 2.14 shows these stages, starting with the Kafka Broker, Producer, Consumer, and Topic. Many other parts, such as Kafka Quotas or Kafka Rebalance features. Nonetheless, in the thesis, we do not deal with Rebalance, Mirror Maker, or Kafka Quotas, and therefore it is not necessary to explain them in detail. However, in case of interest, we recommend the previously mentioned literature.

2.3 Strimzi

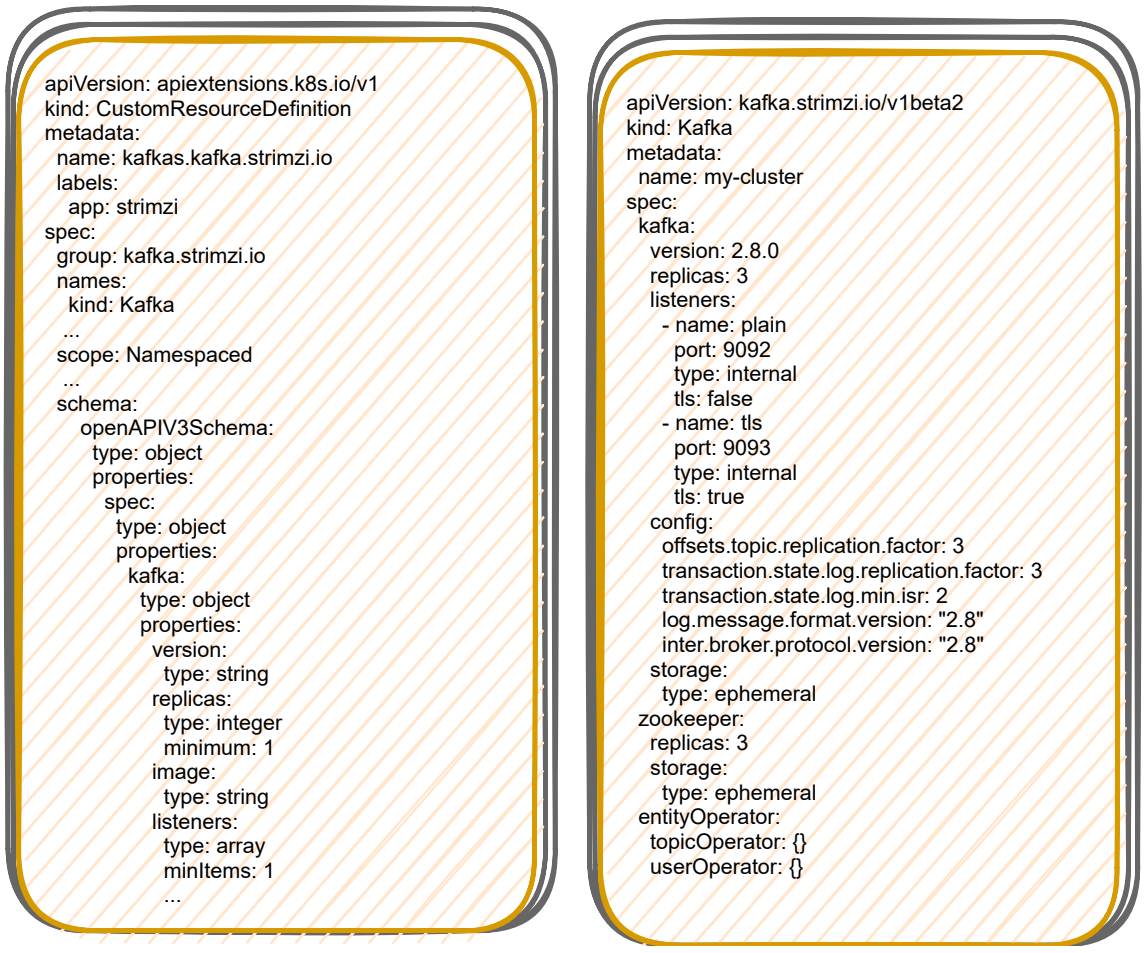
This section describes the fundamental parts of the Strimzi project. Moreover, it explains the whole architecture with all Operators (i.e., Topic, User, Cluster). The description is based mainly on Strimzi documentation and blog posts[7, 6].

The information described in Sections 2.1, 2.2 was a precursor to a complete understanding of the Strimzi system. Strimzi is an Apache Kafka orchestrator in the Kubernetes environment. Therefore it is a collection of operators that simplify working with Kafka. The Operator in Kubernetes is a component that is always in one of the following three states:

- **Observe** – gain the desired and current state,
- **Analysis** – compares these two states and finds the differences,
- **Act** – subsequently, if the given differences were found, it will do a reconciliation that will make the current and desired state identical

One can understand these Operators as a superset of the Deployment controller, which, like other controllers, followed the 1 algorithm. The main difference is that the Operator oversees *Custom Resources - CR*. A Custom Resource is an extension of the Kubernetes

API. These CRs define application objects in the Kubernetes environment. Moreover, this is associated with the *Custom Resource Definition*, which declares what values and types a given Custom Resource can acquire. We can also imagine that Custom Resource Definition is a template comparable to classes in the Object-Oriented programming world, and Custom Resource is an instance of the class. Strimzi defines a Custom Resource Definition for each Kafka component we described in section 2.2 except for clients. For example, for the KafkaBroker component, Strimzi has Kafka Custom Resource Definition.



(a) Example of Kafka Custom Resource Definition (Unnecessary parts omitted for brevity).

(b) Example of Kafka Custom Resource

Figure 2.15: Kafka Custom Resource Definition and Kafka Custom Resource (Class and Instance)

Figure 2.15 illustrates the mentioned Custom Resource and Custom Resource Definitions. In Figure 2.15 (left side), one can see Kafka Custom Resource Definition that shows several essential parts:

- **labels.app.strimzi** – every Kafka Custom Resource in Kubernetes contains this label, and with that, it is easier to find these resources
- **spec.names.kind.Kafka** – by this attribute, we specify how the Custom Resource type will be uniquely named. In this case, the label is Kafka.

- **spec.scope.Namespaced** – type of environment scope. It distinguishes between Custom Resource, which works multi-namespace or single-namespace. Because Kafka Custom resource has value Namespaced (single-namespace), it can work in one namespace. On the other hand, we also know the Custom Resource can have the scope set to cluster (multi-namespace), which means they will observe all the namespaces that the Kubernetes cluster has.
- **spec.schema** – this is the whole declaration of the Custom Resource Definition. In the child nodes, we can see what types the individual attributes must comply with and the restrictions on the given types. For example, the attribute *replicas* has a restriction that it must be at least one and similarly for other attributes (it can not be zero).

On the other hand, we have Kafka Custom Resource (Figure 2.15 - right side), which includes parts worth mentioning:

- **apiVersion** – This is the REST API offered by the Custom Resource Definition. The prefix must also match the value found in Kafka Custom Resource Definition in spec: group.
- **metadata.name** – Custom Resource name,
- **spec.kafka.version** – version of Kafka to be used,
- **spec.kafka.replicas** – number of Kafka Pods to be deployed,
- **spec.kafka.listeners** – types of listeners to be supported by a given Kafka instance. In this case, we see two types, one with plain communication listening on port 9092, and the second listener with encrypted communication using TLS technology and listening on port 9093.
- **spec.kafka.config** – these are additional configuration features that are added to Kafka (i.e., broker.id, log.dirs, zookeeper.connect, compression.type, cleanup.policy, delete.retention.ms),
- **spec.kafka.storage** [18] – the storage type. Kubernetes supports two storage types. In Figure 2.15, it is ephemeral storage. Ephemeral storage is usually a directory somewhere in the operating system on our Kubernetes node. It works the same as a temporary directory. There are also risks associated with this; if the Kubernetes node crashes, then the data stored in the ephemeral storage will be lost. The same thing will happen if we get a running Pod that will use ephemeral storage. In case of a restart, together with the new Pod, empty storage will be created, not containing the previous data. The second type of storage is Persistent, which eliminates these risks.
- **spec.zookeeper.replicas** – number of Zookeeper Pods to be deployed,
- **spec.entityOperator** – configuration for Entity Operator.

2.3.1 Architecture

Strimzi's architecture consists of two central units where the first unit is the Kafka architecture and the other components with which it communicates. The second unit is the Operators architecture, consisting of a Cluster Operator, an Entity Operator, a Topic Operator, and a User Operator. These Operators each have control loops, which control the already defined Custom Resources. (i.e, Kafka User, Kafka Topic, Kafka and Kafka Connect, Kafka Bridge, Kafka Mirror Maker, Kafka Mirror Maker 2, Kafka Rebalance)

The Kafka Architecture consists of several components, each performing specific tasks. Zookeeper is one of the most significant dependencies for Kafka and limits it in several areas, scalability, metadata management and Deployment itself. The answer to these problems came in a 2020 Kafka Improvement Proposal (KIP)[14]. As a result, Kafka 3.0 should run without Zookeeper. Its responsibilities include, for example, leader election of partitions or storing the status of Kafka Brokers or Consumer offsets. Clients in Figure 2.16 are classically Producer and Consumer as we know from the section 2.2, so their objective is clear. On the other hand, HTTP clients communicate with the Kafka Bridge (a component provided by Strimzi) and thus connect the Kafka cluster and the clients themselves. It

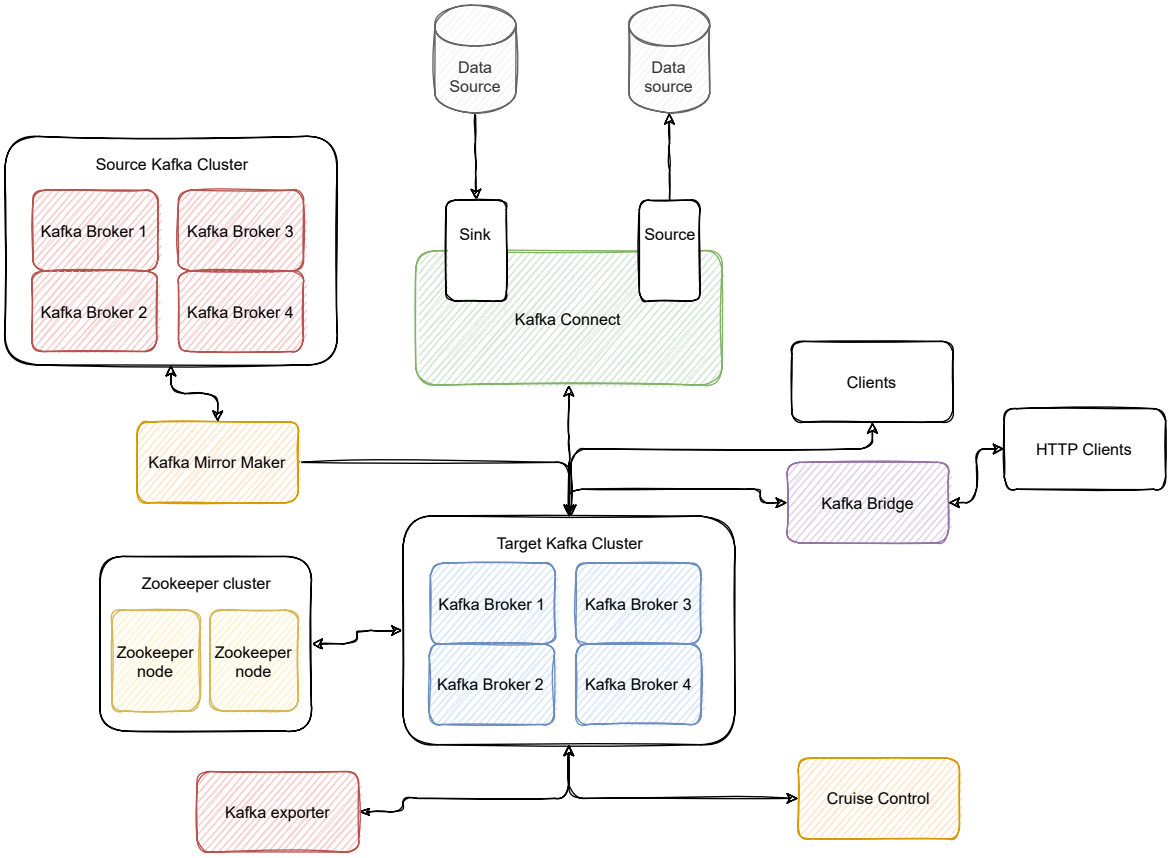


Figure 2.16: Strimzi Kafka architecture

communicates by default via the REST API, and the user can create, delete, and update Consumer, Producer, Topic and similar resources that Kafka Bridge offers. So Kafka Bridge is nothing more than an HTTP proxy that integrates HTTP clients with a Kafka cluster. Another part of the Kafka architecture is the Kafka Exporter, which is used to extract additional metrics and supply them to Prometheus[15]. Then we have Kafka Connect and Kafka Mirror Maker, where we described the meaning of these components in Section 2.2. The last essential component, especially for the overall balancing of the Kafka cluster, is Cruise Control. This component collects data on CPU usage, partitions status, and many

[14]KIP-500 – removal of Zookeeper with replacing him with self-managed metadata quorum https://cwiki.apache.org/confluence/display/KAFKA/KIP-500

[15]Prometheus – open-soured metrics-based project. Moreover, it provides an alerting system with incredible features, in case of interest https://prometheus.io/

other metrics. Cruise Control creates a workload model and analyses it when necessary to perform balancing and rearrange the load across the Kafka cluster. Everything we have described is shown in Figure 2.16.

The second part of the Strimzi architecture is the collection of Operators. In the beginning, we described what such an Operator does (reconciliation/control loop). Strimzi contains three Operators, where hierarchically, the highest is Cluster Operator. This manages Kafka, Kafka Mirror Maker, Kafka Mirror Maker 2, Kafka Connect, Kafka Connector, Kafka Rebalance, and Kafka Bridge Custom Resources. Furthermore, since Kafka Custom Resource encapsulates the Entity Operator (Topic and User Operator running within the same Pod but in different containers) and Zookeeper, the Operators mentioned above are also deployed with each Kafka Custom Resource deployment. Figure 2.17 illustrates whole Strimzi ecosystem, for which the Cluster Operator is responsible.

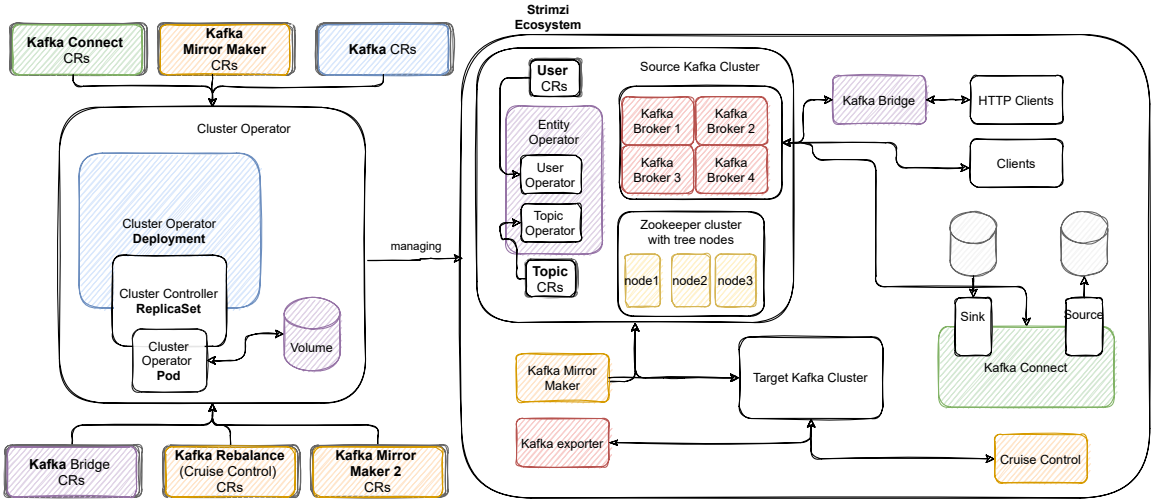


Figure 2.17: Strimzi Operators architecture with Strimzi ecosystem

The Topic Operator takes care of creating, deleting and updating individual Topics. It is also necessary to mention that the Topic Operator ensures synchronisation between the Custom Resource Topic and the Topic located inside Kafka and keeps them in sync. Strimzi documentation says - *For instance, assume the scenario where the user changes different topic properties in Kubernetes but simultaneously in Kafka itself. Also, imagine another scenario where one changes topic property simultaneously. The first action is considered allowed, and the solution for this is a 3-way diff (more about this method in section 2.19). In general, this method constructs these two differences' union and finds out where the intersection is not empty. The second one is treated as incompatible change. It must be deterministically selected by some winner policy implemented inside Topic Operator.*

The User Operator is responsible for the Kafka User Resource, which specifies authentication and authorisation for individual components. It can be, for example, the Producer that can not change data in a Topic with a particular name or prefix. In other words, we can define read and write rules for Topics. In addition, we can create different types of Kafka Users, which support authentication such as TLS or SCRAM-SHA. Nevertheless, if we use SCRAM-SHA authentication, we must also configure one of the Kafka Broker listeners. When one creates Kafka Custom Resource, then immediately the User Operator creates an associated Secret with the credentials. These credentials are then submitted to

the Consumer or Producer configuration. Credentials ensure that the Producer or Consumer can connect to Kafka Broker and send or receive messages. Several components can also be used in authorisation, such as ACLs (access control lists). There is support for the Keycloak or Hydra authorisation server for more complex rules. Another exciting feature is User quotas, ensuring that one client will never overload the entire Kafka Broker, and the total load will be limited.

2.4 Strimzi system tests

This section describes the basics of the Strimzi system tests. At first, we will go through a short description of how we test Strimzi. Then in Section 2.4.1 we explain the fundamentals of JUnit5 and how tests are discovered and executed. Lastly, in Section 2.4.2 we explain Strimzi system test management and execution flow.

Overall testing begins, as we know from textbooks with unit testing. Subsequently, if this phase is successful, we move on to integration tests and then to system tests. Of course, the most time-consuming is the system tests, which in our case take about 40 hours to complete. The testing phases are dependent on each other in the order in which they

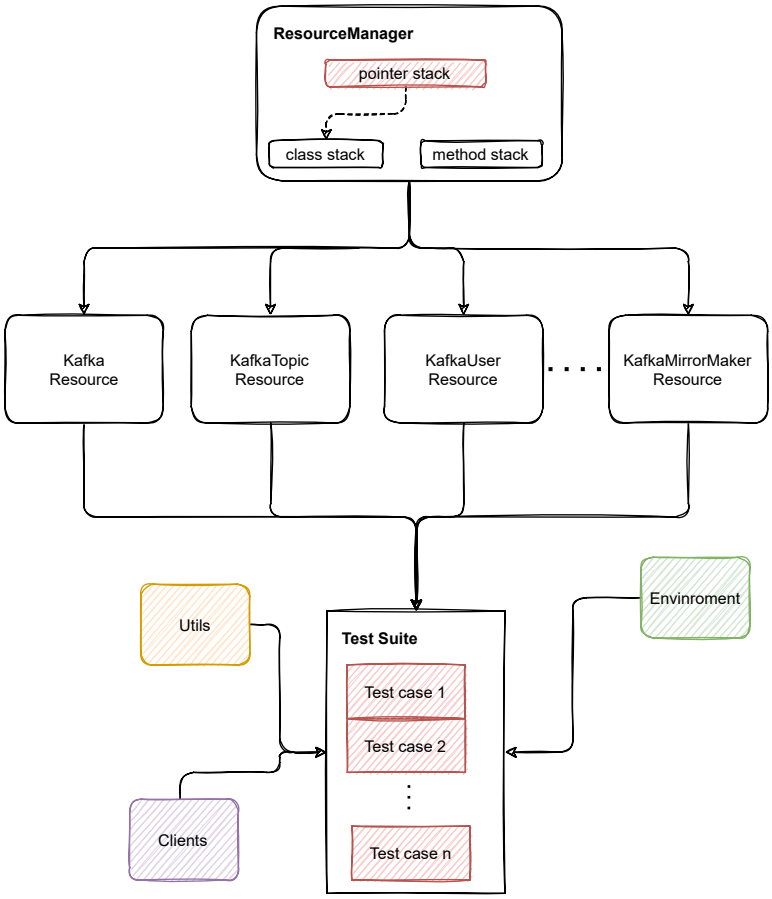


Figure 2.18: Strimzi system tests top-level component architecture

are executed. For instance, integration tests will not run if unit tests fail, similarly to integration and system tests. Furthermore, system tests run on multiple infrastructures such as *Openstack*, *Microsoft Azure* or *Amazon Web Services*. There are certain limitations

to the set of tests on each of these infrastructures. Since these are Kubernetes system tests, it is essential to realise that the total load on the computing resource is enormous. At the same time, the preparation of resources and their cleaning is time-consuming. Therefore, our system tests have two essential parts. The first is resource classes that provide the user interface for creating, retrieving, deleting, and updating these resources. Moreover, we have three independent stacks that serve as resource storage. These stacks are responsible for storing all resources based on the test case. Furthermore, the deletion of these resources is transparent for the user just as if it is a resource created in a *@BeforeAll*[16] annotated method. The second fundamental part is auxiliary classes such as Utils[17], Apache Kafka clients for external communication, Kubernetes client offering an API for communication with the Kubernetes cluster and finally classes such as *Constants* and *Environment*. This can be seen in Figure 2.18.

2.4.1 JUnit5 relation and execution of test cases

Junit5 Engine handles the entire implementation and management of the test lifecycle. The Engine facilitates the discovery and execution of tests for a specific programming model. In other words, it is the entity in charge of discovering and executing tests. Discovering can be thought of as scanning all the classes and methods in specific directories. The Engine has specified in advance which signatures to include in the test tree. In the case of the Junit5 Engine, it is a sequence of chaining methods, which gradually add all classes (test suites) and methods (test cases) to the test tree. They also add the test types defined by them (i.e., *@TestFactory*, *@ParametrizedTest*, *@TestTemplate*). Everything is depicted in Algorithm 2.4.1.

Algorithm 2 Junit5 Engine: Discovery selector resolver

```
1:  procedure RESOLVESELECTORS(DiscoveryRequest request, Descriptor descriptor)
2:      EngineDiscoveryRequestResolver.<JupiterEngineDescriptor>builder()
3:      .addClassContainerSelectorResolver(new IsTestClassWithTests())
4:      .addSelectorResolver(c → new ClassSelectorResolver(classFilter, config))
5:      .addSelectorResolver(c → new MethodSelectorResolver(config))
6:      .addTestDescriptorVisitor(c → new ClassOrderingVisitor(config))
7:      .addTestDescriptorVisitor(c → new MethodOrderingVisitor(config))
8:      .addTestDescriptorVisitor(c → TestDescriptor::prune)
9:      .build();
10:     .resolve(request, descriptor);
```

Once the resolver is created, we can run the following algorithm, using the resolver and creating the already mentioned tree of *TestDescriptors*. Here is a detailed description of how the algorithm works:

1. Enqueue all selectors in the supplied request to be resolved.
2. While there are selectors to be resolved, get the next one. Otherwise, the resolution is finished.

[16] **@BeforeAll** – is JUnit5 annotation, where one specifies what must be executed before all tests in the test suite.

[17] **Utils** – type of class that consists of static methods, which in general dynamically wait for a specific event. For instance, waiting for Rolling Update, if one changes Kafka configuration

(a) iterates over recorded resolvers in the directive that they were recorded in and discover the foremost one that yields a resolution other than unresolved().

(b) If such a resolution exists, enqueue its selectors.

(c) For each exact match in the resolution, expand its children and enqueue them as well.

3. Iterate over all registered visitors and let the engine test descriptor accept them.

The second phase after the correct scan of test cases that the user wants to perform is execution. In this case, TestEngine already has a TestDescriptor in which all the information needed to run is available. At this stage, the TestEngine must always notify the Junit5 platform of the success or failure of the test case. Moreover, Engine instantiates the *SameThreadHierarchicalTestExecutorService* class, which ensures that each test is performed sequentially.

2.4.2 Strimzi system test management and execution flow

In the previous Section 2.4.1, we described the intricate parts of loading and the type of tests performed. In the case of the Strimzi part, adding several mechanisms (i.e., creation of Kubernetes cluster, communication with Kubernetes cluster, management of Kubernetes resources, waiting for conditions) is necessary. We solve all these parts in Strimzi. We create Kubernetes clusters in several ways as we test the product on several infrastructures. For example, on Microsoft Azure, we create a Minikube (a subset of the Kubernetes cluster, one-node cluster) with approximately eight CPUs and 16GB of RAM. In Openstack, we typically create a six-node cluster with three master nodes and three worker nodes. Each has eight CPUs and 16GB available (similarly to Amazon Web Services).

Communication with the Kubernetes cluster is guaranteed by the Fabri8 Kubernetes client `https://github.com/fabric8io/kubernetes-client`. This client provides a Java client with many methods that communicate directly via the Kubernetes REST API. Most methods are designed to create, update, delete and retrieve a given resource. In practice, we will also encounter the term CRUD methods. To illustrate, we can imagine getting all the namespaces on a given Kubernetes cluster. All namespaces are obtained using the command *client.namespaces().List();*.

The overall orchestration of Kubernetes resources is handled by the *ResourceManager* class and its additional resource classes. As we wrote at the beginning of Section 2.4, it includes three stacks where the main/pointer stack points to the method or class stack based on context. For example, suppose the execution is located in *@BeforeAll* or *@AfterAll* annotation, we add elements to the class stack. In other scenarios, such as in the test case or *@BeforeEach*, we add elements to the method stack. This data structure will guarantee the correct order of resources deletion at the end of each test or test class. This is because we want to delete resources in the order they were created. So if we create first Kafka, Producer, and lastly Consumer, then in the clean-up phase, we will first delete Consumer, Producer, and finally Kafka. Thus, the user who creates the test cases does not have to delete individual resources created for the entire test. In other words, the clean-up phase is transparent to the user. However, if one wants to delete the resource explicitly, it is possible via the following command *ResourceType.delete(name)*. Algorithm 2.4.2 defines clean-up phase.

Algorithm 3 ResourceManager generic deletion algorithm

```
1: procedure DELETELATER(MixedOperation<T, ?, ?, ?> operation, T resource)
2:     switch(resource.getKind()) {
3:         case Kafka.RESOURCE_KIND:
4:             pointerResources.push(() →
5:             operation.inNamespace(resource.getMetadata().getNamespace())
6:             .withName(resource.getMetadata().getName())
7:             .withPropagationPolicy(DeletionPropagation.FOREGROUND)
8:             .delete();
9:             waitForDeletion((Kafka) resource);
10:        );
11:        break;
12:        case KafkaConnect.RESOURCE_KIND:
13:        case KafkaMirrorMaker.RESOURCE_KIND:
14:        ... (other resource)
15:            // similar to Kafka resource
16:        default:
17:            pointerResources.push(() →
18:            operation.inNamespace(resource.getMetadata().getNamespace())
19:            .withName(resource.getMetadata().getName())
20:            .withPropagationPolicy(DeletionPropagation.FOREGROUND)
21:            .delete();
22:        );
23:    }
24:    return resource;
```

By contrast, when creating any resources, the user has at his disposal, for example, KafkaResource, KafkaTopicResource, and the like. Each of these classes contains predefined templates that include specific configuration settings. A typical example is Kafka, which can be seen in Listing 2.1.

```
private static KafkaBuilder defaultKafka(Kafka kafka,
            String name, int kafkaReplicas, int zookeeperReplicas) {
    return new KafkaBuilder(kafka)
        .withNewMetadata()
            .withName(name)
            .withNamespace(ResourceManager.kubeClient().getNamespace())
        .endMetadata()
        .editSpec()
            .editKafka()
                .withVersion(Environment.ST_KAFKA_VERSION)
                .withReplicas(kafkaReplicas)
            .endKafka()
            .editZookeeper()
                .withReplicas(zookeeperReplicas)
            .endZookeeper()
        .endSpec();
}
```

Listing 2.1: Default Kafka Custom Resource in KafkaResourceċlass

Another part of the Strimzi system tests is the wait for methods mechanism. It is used primarily in scenarios where it is necessary to wait for an event to occur. An example could be waiting for a Rolling Update to occur when Kafka's original Statefulset changes. The

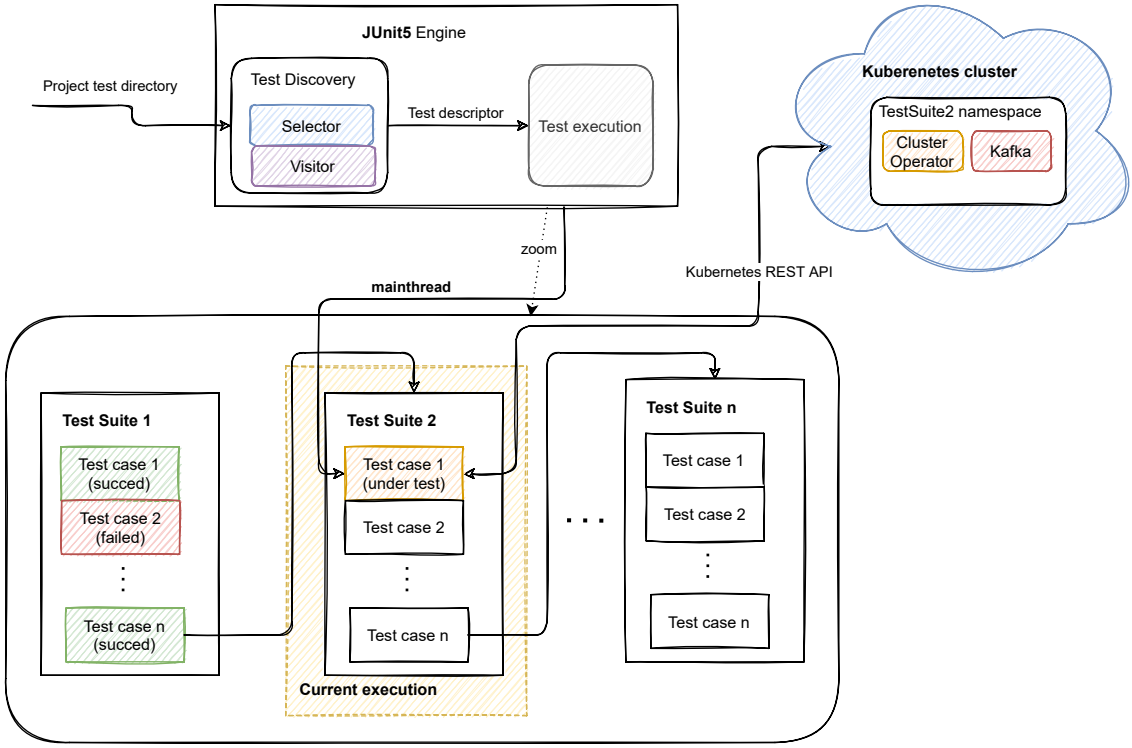


Figure 2.19: Strimzi system tests execution flow

second example could be while waiting for a particular pessimistic scenario (i.e., the Cluster Operator Pod will switch to the CrashLoopBack state, and the KafkaBridge Deployment Status will contain the text in the message).

So if we summarise everything we have learned. It all starts with scanning the test directory, which provides a tree of TestDescriptors. This is the primary responsibility of TestEngine, which uses selectors to filter out all test cases and the visitors who accept the individual test cases. As soon as we have a tree available consisting of TestDescriptor nodes, TestEngine starts execution. This execution is sequential for each test case. At the same time, thanks to our management and defined resources, we can communicate with the Kubernetes cluster. For example, in Figure 2.19 we can execute n Test cases where Test Suite 2 is currently executed and specifically Test Case 1. The attentive reader will realise that the execution model is sequential due to Java's main thread, the primary thread identifier.

Chapter 3

Theory of parallelisation

This chapter describes the fundamental theory of parallelisation (i.e., Amdahl's law (3.1), Shared memory (3.2), Threads and Processes (3.3), Mutual Exclusion (3.4), Synchronization (3.5), Asynchronous tasks (3.6)). Moreover it is based on the following books *An Introduction to Parallel Programming* [16] and *The Art of Multiprocessor Programming* [9].

In the past, computers did not have an operating system. They could only execute one program at a time. The programmers of the time were as respected as the virtuoso in music. Writing such programs has been highly challenging. This problem was solved by developing operating systems that can run several processes (programs). At the same time, processes use the so-called variant of *coarse-grained communication*. Coarse-grained communication includes primitives such as sockets, signals, semaphores, shared memory, and files. These primitives allow them to communicate with each other using signals, files or shared memory. Processes were virtually von Neumann computers, which contained their own memory space that included instructions and data. Subsequently, the processes executed these instructions according to the semantics of the assembly language. The last part was a set of I/O operations to communicate with each other. Thus, if we combine all the elements, we will have a model called *Sequential*. Most of today's programming languages use this model. Hence, the sequential programming model is intuitive, and it creates a sequence of operations that follow each other, thus making the expected result. However, it has limitations on performance and time consumption on specific tasks. During the twentieth century, technological advances brought a regular increase in the processors's clock speed so that the speed of operations "accelerated" itself over time. Nevertheless, this scenario is not repeated in the twenty-first century. Today's advances in technology bring about a regular increase in parallelism but only a slight increase in clock speeds. The use of parallelism is one of the significant challenges of modern software engineering.

3.1 Amdahl's law

If we imagine ourselves as a user that would like to migrate from a single-processor program to a multi-processor program, it would be helpful to know that if we begin with the parallelisation of such a system, it will eventually pay off. Moreover, many people often believe that if we build a multi-processor program from a one-processor program and run it on 3-cores, the overall acceleration will be three times. This is an illusion, and we will never get such a result. The main problem is the division of labour which is not uniform for all parts. For clarity, we will illustrate with an example. Imagine that one has to construct

a table. In this case, it is a sequential approach. Adding four identical tables (so there will be five) will take five times more time than one. Suppose four friends come to help (we assume they are just as skilled and start simultaneously). The acceleration for such identical tables will be five times. Nevertheless, everything gets complicated if the tables are not the same. For example, the second table might be more complicated to build and take more time than the others. Furthermore, the first will be smaller, and thus the total time will be lower. These discrepancies imply that the acceleration will not be close to 5-times, but it will probably be only 3-times. This kind of analysis is crucial for concurrent computation, and thanks to Mr. Amdahl [8], we have a formula for such calculation. It is called Amdahl's law, which can be seen in Equation (1).

$$S = \frac{1}{1 - p + \frac{p}{n}} \tag{1}$$

The formula defines the acceleration S, which depends on the quantities n and p. n is a non-zero positive number that represents the number of concurrent processors performing the same job. p is a non-zero positive number that defines how much work is done in parallel. The sequential part that cannot be parallelised is defined as the difference between the total work and the work that can be parallelised (*1 - p)*. The parallel component is expressed as the ratio of the parallel part and the number of competitors by the processor (p / n). So if we sum up these two parts, we get the total time performed by parallel computation (1 - p + p / n). Then, finally, we have to put the ratio between the sequential (single-processor) time and the parallel time, and we get the already mentioned Equation 1. If we apply this formula to the previous example with five friends who want to build five tables, we get such an Equation (3).

$$S = \frac{1}{1 - \frac{3}{5} + \frac{\frac{3}{5}}{\frac{5}{1}}} = 25/13 = \sim 2x\ acceleration \tag{2}$$

Before we dive into the overall terminology and discuss the *Critical section*, *Mutual exclusion*, it is necessary to know the program's correctness. The correctness of the program consists of two essential properties. The first is the safety property, which states: „*Bad thing never happens*". To illustrate, imagine the concurrent program never ends up in a deadlock[1]. The second is the liveness property, which tells us: „*An excellent thing will happen eventually*". So, for instance, the program always terminates. Thus, if we combine these two properties, we say that the program is correct.

3.2 Shared memory

One needs to understand how memory is organised and how a computer accesses individual data. The speed of memory in a computer is usually much slower than the speed at which the processor operates, and if one processor overwrites data in memory, the others must wait. All processors access the same memory in the global address space in this type of memory.

[1] **Deadlock** – is one of the possible situations that occur in an environment where two or more threads/processes operate with shared memory. Specifically, this situation occurs when Process/Thread A and Process/Thread B enter a wait state because a given shared resource is held by another Process/Thread that is waiting for another resource held by another waiting Process/Thread.

Definition 1 ***Shared memory*** *– is a type of memory where all CPUs has access to the same address space.*

So if one processor makes a change to the data, all the other processors will know about it. The shared memory architecture is classified as UMA (Uniform Memory Access) and NUMA (Non-uniform Memory Access). This classification tells us how the individual processors are connected to the memory and how fast the data can be accessed. The wise reader might realise that memory access will be the same for all processors in Uniform memory access architecture. While at Non-uniform Memory Access, the time will be different. Each processor has its cache memory in the Uniform Memory Access, storing the most frequent data. However, if the processor uses cache memory, there is a very high risk for cache coherence[2]. Fortunately, this cache coherence is handled by hardware in multicore processors.

3.3 Processes and Threads

If one imagines a shell script with a predefined set of instructions (bash commands), the moment someone runs it, it becomes a *process* running in the Operating System.

Definition 2 ***Process*** *– is a dynamic entity, which has its own global address space (set of instruction and data).*

We can also imagine it as a static entity (written shell script) and a dynamic entity (shell script execution). The Process contains program code, its data, and status information. Each Process is independent of the other and has its own address space in memory. On the other hand, there is also a subset of the Process, and it is a thread.

Definition 3 ***Thread*** *– is a lightweight variant of the Process that has an independent execution path and shares code and data within a specific Process.*

Each thread must be part of a process. Thus, the data we work with is shared with all threads inside the Process. Furthermore, each thread has an independent path of program execution. Therefore, one can imagine a thread as a lightweight variant of the Process. It is well known that threads take up less memory. Moreover, the operating system can switch faster between individual threads than between processes (context switching[3]). In general, threads can be in one of four states:

1. **New** – If the main thread spawns a new thread, that thread will be in the *New* state. Moreover, the descendants of the main thread can further create a tree hierarchy of new threads.
2. **Runnable** – If one creates a thread, it automatically acquires the *New* state. Subsequently, in order to change to the *Runnable* state, it is necessary to run the thread explicitly.

[2] **Cache coherence** – this is a situation where one of the processors obtains a value from shared memory and tries to make a change in its cache memory but fails to do so. For example, update to shared memory (while the other processor reads a value that has not yet been updated and will work with the wrong value)

[3] **Context switching** – it is a situation where the Process scheduler finds out that some processes have spent a fair share of their time on the processor and swap it with the different Process. When this happens, the Operating system stores the state of the Process or thread and then loads the state of a different process.

3. **Blocked** – If a thread needs to wait for an event, it switches to the *Blocked* state. This is very useful in terms of resource utilisation. If the event occurs, the operating system assigns the CPU time and returns the thread to the *Runnable* state.
4. **Terminate** – The thread switch to the *Terminate* state if it was previously aborted abnormally (i.e., using inter-process communication) or complete its execution.

3.4 Dependencies and Protection

One of the main challenges in parallel programming is detecting dependencies between threads. Imagine a situation where two threads access the shared variable x. Then, *Thread A* reads a value from the shared variable x and starts execution. Subsequently, the scheduler switches the context, and *Thread B* reads the value of the shared variable x. Then *Thread B* modifies the value of $x = 10$. The scheduler switches the context again, and *Thread A* is currently operating with the wrong value. This is one of the possible faults that can occur in parallel programming. With this example, we have described the Data race failure.

Definition 4 ***Data race*** *– is a situation where two or more concurrent threads access the same address space, and one of these threads has changed it.*

Fortunately, as programmers, we can eliminate such errors. The process begins with the detection of critical sections in the code.

Definition 5 ***Critical section*** *– section of code, where two or more concurrent threads have write-access (simultaneously), and at least one of them can write to it and can produce erroneous behaviour.*

As can be seen from Definition 5, the programmer must look for such places. It can be a simple increment of a shared variable or a complex structure or object change. If these places are detected, it is necessary to perform the next step – use *Mutual Exclusion* (Mutex).

Definition 6 ***Mutual exclusion*** *– two threads are excluded from being in the critical section at the same time.*

By using a mutex, we guarantee that only one thread will access the shared resource at a time. One will have to acquire a lock whenever one wants to modify a thread or read from a shared resource. Then one modifies the source and finally releases the lock. Acquiring a lock is an atomic operation performed as single action and cannot be interrupted by other threads.

We know several lock implementations, but not all of them guarantee us the liveness property. As a reminder, the liveness property tells us that: *„A particularly good thing will happen eventually“*. For example, a program never „hangs“. However, they usually guarantee the safety property, and the attentive reader would undoubtedly notice that Mutual Exclusion has a safety property. Some of the leading implementations of lock are the following:

- **Reentrant lock** – This type of lock can be locked unlimited times. Nevertheless, the important thing is that if we want to unlock the lock, we have to do it the same number of times. The use of this type of lock can be seen, for example, in recursive functions, when we lock the lock several times and unlock it the same amount of times.

- **Try lock** – Non-blocking version of the classic lock, if the Mutex is available, it acquires the lock and returns instantly true at the same time. Otherwise, it returns false. This behaviour is beneficial if the thread can do other things than the critical section. Therefore, it will not be blocked as a classic lock.
- **Read-write lock** – Multiple readers can read from a shared resource. However, once a thread is locked in ReaderLock, it is not possible to get a thread that wants to modify the value of the shared resource. This is only possible if the thread that reads the value released *ReaderLock* for the shared resource. At this point, the thread can be locked using *WriterLock*, and no other thread can access it. This type of lock is intended mainly for situations where we have more threads that will read from a given shared resource and fewer threads that will write (i.e., databases).

3.5 Synchronisation

The main problems posed by mutexes are, for example, *busy-waiting*, deadlock, livelock or starvation.

Definition 7 ***Busy waiting*** *– waiting until thread, which is in the critical section, release lock or flag. The mutual exclusion problem requires waiting, and there is no way to avoid it.*

Elimination of busy-waiting is possible using another synchronisation primitive such as *Semaphore* or *Condition variable*. The *Condition variable* represents a queue of threads waiting for a specific event and associated with a Mutex. Using these two parts, they implement a higher abstraction called the *Monitor*. The Monitor is a high-level synchronisation primitive that ensures mutual exclusion while giving threads the ability to wait until an event occurs. Noteworthy is the fact that the *Condition variable* involves three operations:

- **Wait** – If a thread locks the Mutex and then verifies the *Condition variable* and finds that the condition is not satisfactory, it immediately switches to the Wait state, unlocks the Mutex, and queues in the wait queue. until the *notify()* signal, which automatically locks the Mutex again and tests the *Condition variable*.
- **Signal** – If a thread has finished executing, it signals with *notify()* and thus wakes one thread from the *Waiting* state.
- **Broadcast** – A variant of the signal operation that wakes up all threads in the queue.

Another synchronisation mechanism is a Semaphore. Sometimes also referred to as a superset of a mutex. If we imagine the simplest Semaphore, we get a mutex. The main difference between a mutex and a semaphore is that the Semaphore allows access to a critical section to more than one thread simultaneously. The amount added to such a section is conditioned by the number one initialises in the Semaphore. The basic principle is that if a thread wants to access a critical section, it must increment this number. If the number reaches zero at that moment, no other thread can access the critical section. If the thread wants to exit the critical section, it decrements the counter. Another difference between a mutex and a semaphore is that a mutex can lock and unlock the same thread, whereas a semaphore can lock and unlock a different thread.

3.6 Asynchronous tasks

Another crucial aspect of parallelisation is knowing what an asynchronous task is. It is an object that is in charge of a predetermined task. This task is performed parallel to the main thread. Imagine a situation where we have to perform several tasks. For example, create several different objects that take a certain amount of time to create. If we used the classical strategy of creating one object after another, the whole process would take a very long time. Hence, we have another alternative; for each of these objects, we submit an asynchronous task. However, it is essential to remember that if we have only four CPUs available and want to create more tasks, for example, twelve, this will result in a situation where the other eight will have to wait until these first threads are done. Therefore, it is better to use *ThreadPool* to create a new thread for each task.

ThreadPool is an object that creates and manages several threads, also called worker threads. If one thread completes its task, *ThreadPool* immediately assigns a new job to the free thread. This eliminates the creation process and thus relieving the load on resources. However, if we want to submit one asynchronous task, then in the main thread, we want the future result. Thus, we use the *Future* mechanism.

Future is another object that creates one asynchronous task. According to intuition, we could deduce that the name was given to this mechanism because we do not know the value initially, but it will be available soon. It also provides access to asynchronous operations, so most implementations have the *get()* method. This operation is blocking and will usually be called if one is at a point where one needs a given result from the asynchronous task. The result will be available as soon as the task is completed.

We could go on to more complex parallelisation concepts, such as partitioning, mapping, agglomeration, concurrent objects and consensus algorithms. However, these topics are not necessary to understand the following chapters. Nevertheless, if the reader has these interesting ones, we recommend reading these facts from the books *An Introduction to Parallel Programming* [16] or *The Art of Multiprocessor Programming* [9].

Chapter 4

Proposal of parallel approach

In this chapter, the author describes the overall design for parallelism in the computation of the Strimzi system tests. At first, Section 4.1 explains the prevailing problems in the Strimzi system tests. Then, Section 4.2 describes alternatives to solve these problems. Finally, the best possible option is proposed that meets all the necessary needs. Next, in Section 4.3 we propose changes that have to be made, especially in the *ResourceManager*, where the current algorithms for resource management are implemented and which currently do not support a thread-safe implementation. Finally, in Section 4.4 and Section 4.5 a proposal for *method-wide* and *class-wide* parallelisation is specified, which is described in detail with the steps that need to be done for its construction (conflicts it contains and solutions proposed using learned knowledge from previous chapters).

4.1 Bottlenecks of current approach

As discussed in Section 2.4, the time required for a given test set is exceptionally time-consuming. It is easier to maintain the correctness of a program using the sequential computing model, but the benefit that parallelism offers are hard to ignore. Nevertheless, one has to ask oneself whether it is possible and worth the investment. To answer such a question, we can use Amdahl's law, which we learned about in Section 3.1. For simplicity, assume that the unit of work will be a test case. It will therefore be necessary to map how many tests can be parallelised. We can find out that by analysing whether a test case contains any shared variable with other tests (i.e., shared *Kafka, KafkaMirrorMaker, KafkaConnect, KafkaUser, KafkaTopic resource*). Once it does not contain any variable, we can declare the test as parallelisable. If a given test contains such a shared variable, it implies that it will have to run in an isolated environment. The manual analysis found that 250 tests could be run in parallel, and 115 must be isolated. So if we apply Equation (1), (which we learned in Section 3.1), the total number of tests is 365. The parallelisable part is *p = 250/365*. The sequence part will be equal to *seq = 1 - p = 115/365*. For only four-core CPUs, we get the following acceleration (3).

$$S = \frac{1}{1 - \frac{250}{365} + \frac{\frac{250}{365}}{\frac{4}{1}}} = \sim 2.1x\ acceleration \tag{3}$$

If we increase the number of CPUs to 8, the total acceleration will be 2.5 times, and if we scale it to 16 CPUs, the acceleration will be almost three times. Consequently, if we imagine that our system tests have a total executive time of 40 hours, all tests will last

approximately 13 hours with parallelisation. Thus, we just showed that it pays to parallelise with this first step.

Another disadvantage of the current approach is that it does not use multiple Namespaces. In our case, for each test suite, we always have one *Namespace* in which we operate. Parallelism allows us to manage multiple namespaces simultaneously while ensuring that the test cases do not overlap. Subsequently, we create in each Namespace a Cluster Operator, again and again; this process usually takes one minute. The ideal approach should be that the Cluster Operator should see all Namespaces and share them for all test suites. Using this approach eliminates much lost time. However, we must be aware of a particular test suite or the test case that will require a different Cluster Operator configuration. At that moment, we must guarantee that some label will annotate that single test case for the entire test suite to run in isolation.

The disadvantages of the current approach mentioned above may be a clear argument for why such a change is necessary. What is also necessary to mention is the structure of the Resources classes in the Strimzi system tests. These are classes that encapsulate both pre-prepared templates and, at the same time, the whole mechanism of creation. If we want

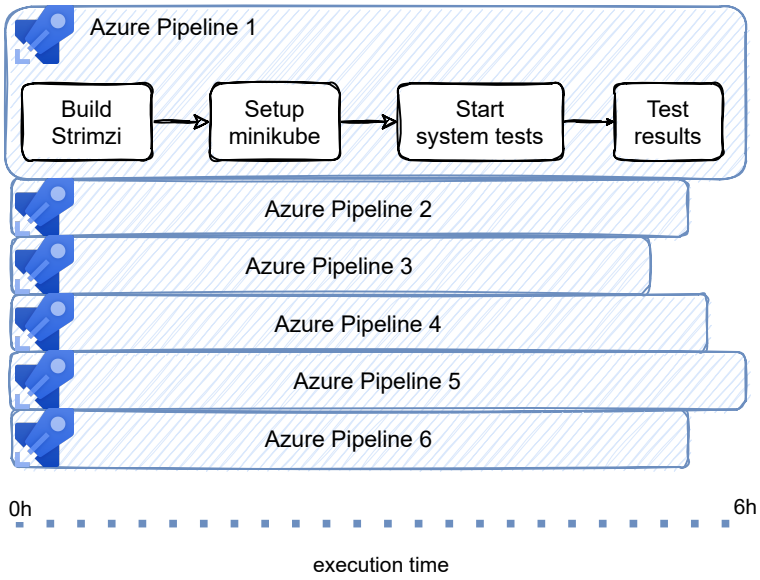


Figure 4.1: Azure pipelines in form parallelism used to execute our system tests

to create a resource, we do it using *KafkaResource.kafkaEphemeral(...).done()* method calls and similarly with other resources. The correct API should propagate everything for the client writing the tests via the *ResourceManager* class where a simple *create()* method would be called. Nevertheless, this fact is more a matter of architecture and not a form of the execution model.

Finally, we can discuss the last limitation for which it is necessary to change. In the 2.4 section, we did not mention such a fact, but there is an attempt at parallelism when using the Microsoft Azure Pipelines. On this infrastructure, we decompose our system tests into several distinctive subsets and run them as Azure separate pipelines[1]. In Figure 4.1 one can see such decomposition. The attentive reader might ask why we cannot run 40 or 100

[1] **Azure pipeline** – one can imagine a pipeline as an Object which encapsulates multiple commands executed in order. Moreover, it is also executed as a separate process.

Azure pipelines and thus reduce the total execution time of the tests. Unfortunately, we are limited only to running six Azure Pipelines simultaneously. By this limitation, the complete set of tests takes approximately 6 hours, which is still a significant amount of time. Similarly, we try to reduce the time at the Jenkins pipeline when using OpenStack[2] and Amazon Web Services infrastructure[3]. However, this Strimzi product must be verified for multiple configurations when running a separate Kubernetes cluster for the entire test suite. Once we launch several such Kubernetes clusters, we are also limited by infrastructure quotas. Overall execution time reduced can be seen in the following Figure 4.2.

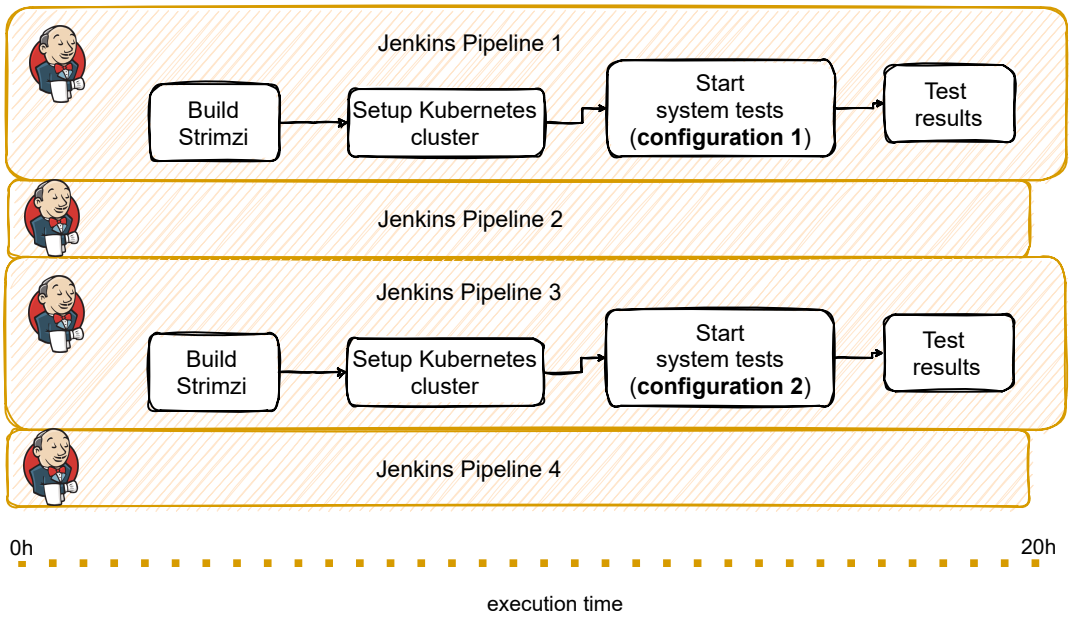


Figure 4.2: Jenkins pipelines in a form parallelism used to execute our system tests

It is also important to mention that we are limited by the number of processes (i.e., pipelines) that use the separate Kubernetes cluster. On Amazon Web Services and Openstack infrastructures, we have not limited the computing resources. This is the fact that we must use and thus think about how parallelisation will lead the way. Undoubtedly, this will not be at the levels of processes, but parallelisation is possible directly in the test set (i.e., using threads) thanks to the available computational resources. However, this decision evokes the approaches described in the next section.

4.2 Possible approaches

From the previous section, we could notice that any attempt to parallelise at the process level (i.e., spawn more pipelines) was impossible, especially in terms of individual infras-

[2]**OpenStack** – is a cloud computing infrastructure that manages physical machines, virtual servers or containers. At the same time, this product is one of the three most active open-sourced products globally. (https://www.openstack.org/)

[3]**Amazon Web Services** – also like OpenStack, is a cloud computing infrastructure that offers a myriad of services (i.e., Amazon Elastic Compute Cloud, Amazon Simple Storage Service). A very admirable attribute of this service is the availability level according to SLA (service level agreement) up to 99.9%. (https://aws.amazon.com/)

tructures' constraints. As a result, we have no choice but to go one level lower and try to parallelise at the test level and thus use the threads.

4.2.1 Writing own testing framework

The first and the most challenging alternative is to write a new testing framework. One would say that this may be an old-fashion approach, but it also has its advantages. One of the leading benefits is flexibility. Imagine that we want to configure how many test cases and test suites we want to run simultaneously. The natural way to do this is using Futures. Each parallel suit is associated with its *Future*, and one uses a composite future to await the completion of all of them. We could do that by using *JDK ExecutorService*[4] and *CompletableFuture*[5]. However, the problem is that writing a new tool would mean writing new tests and partially rewriting them all. Since our tests are currently designed on top of the JUnit5 platform, it is not very acceptable for us to do such a thing.

4.2.2 Writing our own Junit5 Engine

Another alternative to reduce the overall load of rewriting all tests would be to write a new JUnit5 Engine. In this case, we would have to write the overall logic of the lifecycle test. It would help if one remembered how we described the dependencies of the current Strimzi system tests in Section 2.4.1. This dependency eliminates the worry of *TestDiscovery* and *TestExecution*. Therefore, if we want to create our *TestEngine*, we have to implement our own *TestDiscovery* and thus create our implementation similar to Algorithm 2.4.1. Furthermore, we need to create our *TestExecution* mechanism. The testing mechanism could be very similar to the previous subsection, thus using the *CompletableFutures* and *ExecutorService* classes that Java offers. One may invoke the idea that this is the best approach that eliminates the discovery of all tests and the overall work of designing a new tool. Unfortunately, it also has its disadvantages. One of them is that if one decides to write their *TestEngine*, they must realise that this eliminates all the annotation support offered by Junit5 *TestEngine* (i.e., @Test, @TestFactory, @ParametrizedTest, @Isolated and @TestTemplate). It is clear that if we write a new *TestEngine*, we have to write our own annotated tests and write our annotations. With this knowledge, even this approach does not meet our needs.

4.2.3 JUnit5 parallelisation

The last alternative is the use of Junit5 *TestEngine* parallelisation. These almost three-year-old features of the Junit5 platform (released 3rd September in 2018) have a lot to offer. For example, parallelisation support for running multiple test cases at one time is possible using the *Java Fork / Join* framework. This framework also includes the implementation of the *ThreadPool* object, which we described in Chapter 3. The overall logic utilises reusable Threads, where, for example. Thread A completes the execution of Test 1; it will be assigned another test immediately and thus, we eliminate the redundant creation of threads. The main advantage of such an approach is that it is not necessary to rewrite a complete performance of the tests. Moreover, it is unnecessary to implement TestDiscovery

[4] **ExecutorService** – is a Java object, which provides a way to execute tasks on threads asynchronously.

[5] **CompletableFuture** – is a superset to Future, which we learned about at the end of Chapter 3. Moreover, it provides exception handling, allows us to combine CompletableFuture, and has many auxiliary methods

and TestExecution because JUnit5 TestEngine already offers them. Related to this is keeping all the annotations mentioned in the previous subsection. Another advantage is the possibility of configuration where we can enable parallelisation using the following commands:

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = concurrent
```

With this setting, it is possible to run the suite test in parallel using Junit5 parallelisation. If we change *junit.jupiter.execution.parallel.mode.default = concurrent* then we let concurrent execution of test cases and test suite run simultaneously. Another good aspect of this feature is the ability to choose the best variant of the parallelisation strategy:

- **Fixed** – *ThreadPool* has a predefined number of threads to work with and can be changed in the configuration using *parallel.config.fixed.parallelism*.
- **Dynamic** – *ThreadPool* has a predefined number of threads based on the calculated available processors multiplied by the number specified by *parallel.config.dynamic.factor*.
- **Custom** – possible custom implementation of the strategy.

However, this configuration does not apply to scenarios where we want to run a particular set of tests in parallel and the other sequentially. Therefore, Junit5 also provides a possible dynamic rewrite of the configuration at build time using the @Execution annotation, which can contain two values for sequential execution (@Execution (SAME_THREAD)) of a test suite or test case or @Execution (CONCURRENT) for concurrent execution of class or test case. Thanks to the mentioned annotations, we can achieve decompositions of tests that will run in parallel and sequentially. It may be apparent to the reader that our needs will be met by using this feature of the Junit5 Engine offers.

However, another common problem with the approach we have described is that the current ResourceManagement is not ready for parallelisation. This problem forces us to rewrite our test architecture, and with that comes the rewriting of the ResourceManager class and its Resource classes.

4.3 Architecture changes

In this section, we will describe all the necessary changes in our system test architecture. We start with designing thread-safe algorithms responsible for managing the resources with which the individual test cases operate. Finally, we describe the design of individual resource classes that will use the Interface pattern[6]

4.3.1 Resource classes

If we think about the whole architecture of the system tests from Section 2.4, one will notice that the Resource classes contain two large pieces. The first is management methods (i.e., create(), delete()), and the second part is predefined templates, which are then used in test cases. Therefore, we suggest that the given parts of the code must be divided into classes, where the methods used for management would be left in these classes. However,

[6]**Interface pattern** – one of the most popular design patterns, which defines a set of operations and creates a contract for a class that must implement these operations.

predefined templates moved to the so-called *Templates* classes. With further improvements and better design, we propose to create an interface that will contain methods for resource management, and each type of Resource class will need to implement such an interface. The given interface should consist of the following abstract methods:

- **getKind()** – an abstract method that will serve as a type identifier of the given resource instance,
- **get()** – the abstract method that will serve as a single resource,
- **create()** – the abstract method responsible for creating the resource,
- **delete()** – the abstract method responsible for deleting a given resource,
- **waitForReadiness()** – the abstract method, for waiting for a given resource until it is ready.

Thanks to this change, we will create a generic method at the heart of the ResourceManager class.

4.3.2 ResourceManager

The most critical part of the system test module is ResourceManager. In Section 2.4, we described how this class works and what exactly it contains. To maintain the context of all resources with which the three types of stacks are currently used. If we are in the *@BeforeAll* context, then it is clear that we switch the pointer stack to the class stack. On the other hand, we switch to the method stack before each test case. However, the cautious reader will realise that such a mechanism will not work in parallel executions.

As part of the change, we propose eliminating all three stacks used to maintain the context and creating a HashMap that will have the name of the test case as an identifier (key). We create a contract for a person who creates the tests to do not equal themselves. As a value in the given map, we will store a Stack that will store all types of resources, i.e. there will always be one stack for each test case. Related to this section is a change in resource creation management. We propose the following thread-safe algorithm 4, which eliminates the invocation of methods from individual Resource classes, but all this will be done within the ResourceManager class. In the given algorithm, there are 3 phases:

- **Find** – finding the resource type and invoking it within the Kubernetes API,
- **Store and future deletion** – save the resource to the stack and automatically delete it throughout the lifecycle,
- **Readiness check** – waiting if a given resource is deployed in a Kubernetes cluster (optional).

Algorithm 4 Thread-safe algorithm for creation resources inside *Resource manager*

Input: ExtensionContext context, GenericType resources

```
1:  for each resource ∈ resources do
2:      type ← findResourceType(resource)
3:      type.create(resource)
4:      // here starts critical section
5:      all_resources.computeIfAbsent((test_name), k− > newStack <> ())
6:      all_resources.get((test_name).push(deleteResource(resource)
7:      // here ends critical section
8:      if wait for resource readiness then
9:          for each resource ∈ resources do
10:             type ← findResourceType(resource)
11:             wait for resource readiness
12:         end for each
13: end for each
```

An essential aspect of this proposed algorithm is also the ExtensionContext, which will identify the current place of execution. There is an ExtensionContext for each test case, and it contains metadata about the test.

Another part is in case the user wants to create ten resource instances independently of each other asynchronously and then create a Barrier[7] because the following verification steps require all resources. Another thread-safe algorithm 5 does a very similar process, waiting for all resources to be created asynchronously. The identification of which resource to wait for is within the given ExtensionContext.

Algorithm 5 Thread-safe algorithm for sychronising resources inside the *Resource manager*

Input: ExtensionContext context

```
1:  Stack<Resource> resources = resourceStack.get(context.getTestName());
2:
3:  // sync all resources
4:  for each resource ∈ resources do
5:      if resource == null then
6:          continue;
7:
8:      type ← findResourceType(resource)
9:      Φ ← getResourceWaitCondition(type)
10:     wait(resource, Φ)
11: end for each
```

Finally, we have the last part, which is deleting resources from the stacks. We propose a thread-safe algorithm 6, which will be used for the overall cleaning of the test environment. Its functionality is configurable. In the beginning, it finds out the condition of the emptiness of the map that contains all the resources. Subsequently, if it does not contain anything,

[7] **Barrier** – is a mechanism in concurrency, which is used to synchronise multiple threadsprocesses. Therefore, any threadprocess has to wait for all the threadsprocesses in that place. Subsequently, if all threadsprocesses arrive at the given place, the threadsprocesses are awakened and can continue their execution

the whole execution ends. However, if the map is not empty, deletion begins. Once this phase is completed, everything will be deleted from the map.

Algorithm 6 Thread-safe algorithm for deletion of resources the inside *Resource manager*

Input: ExtensionContext context

```
1: Ψ ← mapResourceEmptinessCondition(context)
2: if Ψ then
3:     break; // everything is deleted
4: while !Ψ do
5:     // checking if some exception in scope of extension context arised
6:     resources.get(context.getDisplayName()).pop().getThrowableRunner().run();
7: // remove stack from map
8: resources.remove(context.getDisplayName());
```

4.4 Method wide parallelisation

In this section, we will describe our proposal for a possible method-wide parallelisation. Method-wide parallelisation is where each test suite will be isolated, and each test case will run in parallel, if possible. We have already approached the condition for running the tests in parallel in Section 4.1. So this is a test that does not use any shared resources. The proposal is decomposed into several steps: which are described in the next paragraphs.

The first step is to create a unique name mechanism for all the resources that are used in the test cases. Since these are Kubernetes system tests, the created resources do not have a random naming generated. By randomisation, we eliminate possible conflicts in parallel execution in a given test suite. Furthermore, random naming does not require additional synchronisation of conflicting resources because each newly created resource will have a different name.

The second step is to create Kubernetes methods that will support namespace operations. This is possible thanks to the Kubernetes client, which we already have in the system tests. However, it contains too complicated invocations of methods, and so for our purposes, it is better to encapsulate this complexity into factory methods. These are mainly methods for communication with the Kubernetes environment (i.e., Pod, ReplicaSet, Deployment, Services, Custom Resource, Custom Resource Definition).

The third step provides a mechanism that determines which methods can be performed parallel and which need to be isolated. For parallel tests, we propose use the *@Parallel-Test* annotation. This annotation will encapsulate the *@Test* annotation, so the JUnit5 framework recognises it as a test. It will also be necessary to add information so that the test can run in parallel. Thanks to the @Execution annotation, which will be set to the value *CONCURRENT*, the test will always run in parallel. On the other hand, tests that will require isolation will use @IsolatedTest annotation. This annotation will be a bit more complex because it will contain not only the @Test annotation but also the read-write lock. As a reminder from Chapter 3, the read-write lock consists of two types of locks. Reader-Lock allows multiple readers to read from a shared source. However, no thread can write to the source if even one reader reads. If no reader reads anymore and one thread wants to write, the file will be locked using WriterLock. Here, however, another thread cannot access until the same thread releases it. So for the @IsolatedTest annotation, we propose using this type of lock to guarantee the system's safety property (mutual exclusion).

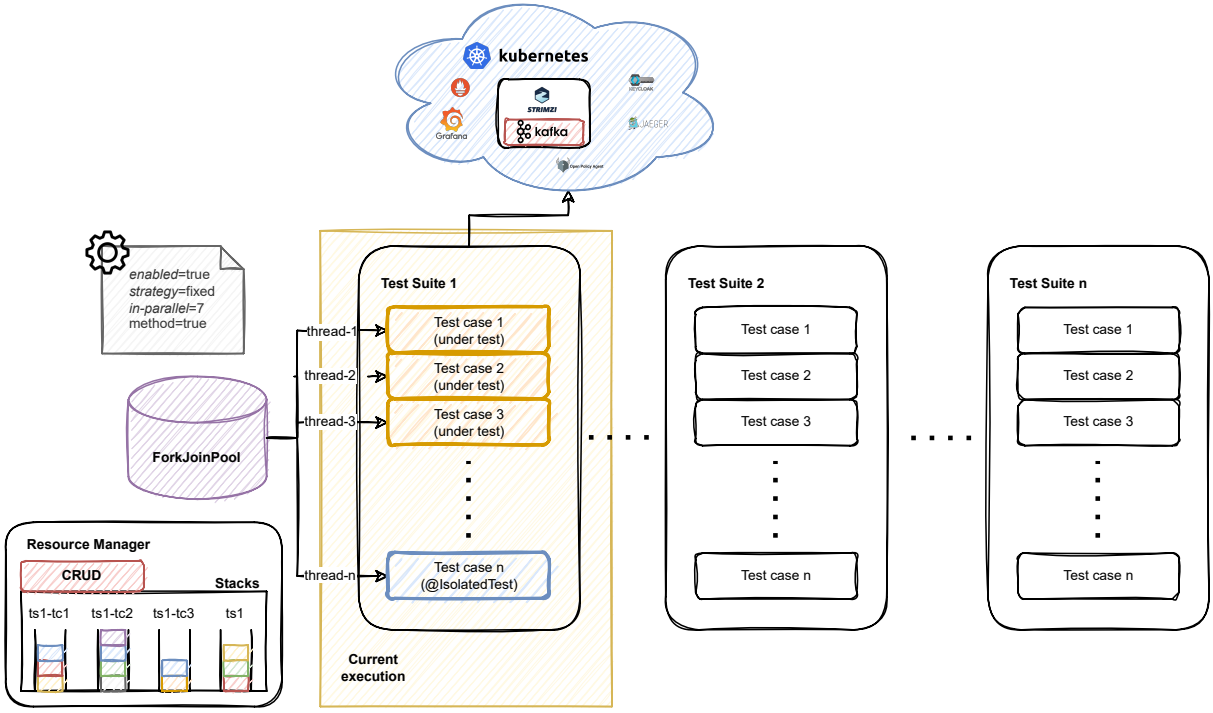


Figure 4.3: The best scenario in *method-wide* parallelism. n number of threads are executed, and there is no one *@IsolatedTest* in the test suite, which means that all test runs simultaneously. Note that *Tc* means Test case in short.

In Figure 4.3 it is possible to see the best scenario that can happen in *method-wide* parallelisation. Moreover, we must realise that if the test suite theoretically contains all *@IsolatedTest*, it would be a sequential execution. Of course, if the computer on which the tests would run contained no more than two CPUs, then it is not possible to run multiple parallel threads with each other (it is possible, but the processor would then have to make many **context switches**, which would lead to a significant decrease in performance). Thus, the more CPUs a given computer/cluster will have, the quicker the results are.

4.5 Class wide parallelisation

In this section, we will describe and suggest what steps are needed to support *class-wide* parallelisation. At first, in Section 4.5.1, we describe all the necessary changes that need to be made. Furthermore, it is restructuring and creating a new class for managing all possible Cluster Operator configurations. Next, we describe the rollback mechanism needed to solve the problem with two test suites that need different configurations. We follow up on this in Section 4.5.2, where we solve the given problem completely. Finally, in Section 4.5.3 we propose a mechanism that determines when to execute test suites in parallel.

4.5.1 Shared Cluster Operator

This change requires multiple interventions in the test suite. Since a new Cluster Operator is currently being created in each test suite, we must always have this Cluster Operator

available in a shared context. This is accompanied by how it will be possible to obtain such a context. In Section 4.3, especially in the description of the ResourceManager component, we partially described the ExtensionContext object, which serves as a test identifier thanks to a *hashcode*[8]. However, we must be aware that any ExtensionContext in either the *@BeforeEach* or *@BeforeAll* scopes of the code cannot be used. If we used such an ExtensionContext, the Shared Cluster Operator would be deleted after the test suite in *@AfterAll* has perished. One elegant approach to solving this problem is to use the *extensioncontext.getRoot()* context, which ensures that the Cluster Operator is not deleted prematurely. Another problem is the lack of an annotation/extension that creates a shared Cluster Operator only once if multiple test suites are run. We propose to create such an annotation *@BeforeAllOnce*. Thanks to JUnit5 and its flexibility, it will be possible to implement such a mechanism by overriding *@BeforeAllCallback*.

Another significant change that needs to be made is the unification of the Cluster Operator installation. This requires a design that encapsulates multiple configurations of the Cluster Operator and would be easy to use for the client. The answer to this is the *Builder design pattern*, which will allow the client to specify the necessary configuration it will require. On the other hand, a person implementing this mechanism will disable parts that he does not want to make available to the user using operators' visibility (i.e., private, protected, package-protected). This eliminates the number of factory methods currently in the project and increases the overall readability of the code. An example of the resulting implementation and invocation for a given client might look exactly like the code shown in 4.4.

```
// cluster operator deployment configuration
clusterOperatorDeployment = new SetupClusterOperatorBuilder()
    (1) .withClusterOperatorName("my-cluster-operator")
    (2) .withExtensionContext(sharedExtensionContext)
    (3) .withNamespace("infrastructure-namespace")
    (4) .withWatchingNamespaces("*")
    (5) .withOperationTimeout(...)
    (6) .withReconciliationInterval(...)
    (7) .withExtraEnvVars(...)
    (8) .createInstallation()
        .runInstallation();
```

Figure 4.4: One of the possible invocation of Cluster Operator deployment using the Builder desing pattern.

This may not be clear from the Figure 4.4, but the *runInstallation()* method should encapsulate all installations such as RBAC, HELM, and BUNDLE. Each of these installations has its preparation of the environment, and therefore it is necessary to distinguish them. For clarity, we will also describe the individual parameters that we indicated in Figure 4.4.

1. **withClusterOperatorName** – will be used to specify the exact name of the Cluster Operator Deployment.

[8] **Hashcode** – hashcode in Java is usually an integer value that has the same number for the identical objects. However, if the objects differ in one of the instance attributes, the hashcode must have a different value. This is a known contract between a Class and its implemented *int hashCode()* method.

2. **withExtensionContext** – possible ExtensionContext specification for resource management. In this case, a shared ExtensionContext object that will ensure that the instance is not deleted prematurely.
3. **withNamespace** – specification of the Namespace name to be created for the Cluster Operator. In this case, the infrastructure Namespace is used.
4. **withWatchingNamespaces** – specification of the Namespaces that the Cluster Operator must observe. In most cases, this will be a configuration where the Cluster Operator is set to *, which semantically means that it observes all Namespaces available in the Kubernetes cluster.
5. **withOperationTimeout** – timeout specification for Cluster Operator internal operations (ie, Kafka cluster, Kafka Mirror Maker creation).
6. **withReconciliationInterval** – specification of the control loop loop interval.
7. **withExtraEnvVars** – additional possible configurations using environment variables (i.e., Strimzi operator namespace labels or Strimzi network policy generation).
8. **createInstallation** – instance construction with pre-supplied attributes.

The last change within the shared Cluster Operator is to create a rollback mechanism that will solve the problem if we have two test suites with different Cluster Operator configurations. Note that it is not possible to have multiple Cluster Operator deployments, as this would overlap and at the same time disrupt the operators. Therefore, we propose to create a rollback mechanism that will solve this problem. The 7 algorithm shows the principle of operation. Specifically, we suggest that the algorithm be divided into two phases where the first is to delete all currently deployed resources. The second phase is the deployment of a new Cluster Operator with a default configuration.

Algorithm 7 Cluster Operator rollback algorithm

```
1:  // 1st phase
2:  // trigger that we will again create namespace
3:  if Environment.isHelmInstall() then
4:      helmResource.delete();
5:  if Environment.isOlmInstall() then
6:      olmResource.delete();
7:  if Environment.isBundleInstall() then
8:      // clear all resources related to the extension context
9:      ResourceManager.getInstance().deleteResources(sharedExtensionContext));
10:     KubeClusterResource.getInstance().deleteNamespace(infrastructure-namespace);
11: // 2nd phase
12: defaultInstance ← buildDefaultInstallation();
13: deployedInstallation ← defaultInstance.runInstallation();
14:
15: return deployedInstallation;
```

However, there is another problem that even this mechanism will not solve, and that is the guarantee that test suites with different Cluster Operator configurations will run in isolation. This issue will be resolved in the following Section 4.5.2.

4.5.2 @IsolatedSuite

One way to solve the problem is when we have different configurations of Cluster Operator, it is necessary to supply some form of synchronisation. Recall *@IsolatedTest* from *method-wide* parallelisation. In this case, we suggest making a different approach because in the *@IsolatedTest*, we used *@ResourceLock*. *@ResourceLock* locks the overall computation, and no other threads can proceed with its execution. In the scope of method-wide parallelisation, this approach is applicable. However, if we use this approach in class-wide parallelisation and thus annotate such test class with *@ResourceLock* using *read_write* lock, it will always execute only one test case. The reason why is that that *@ResourceLock* will be propagated to each test method and thus resulting in sequence mode. Because of this problem, we propose to create *@IsolatedSuite* as a labelling unit and implement an additional synchronisation mechanism, which will take care of multiple *@IsolatedSuite*. The easiest way how to tackle such a problem would be using *AtomicBoolean* as a flag. When *@IsolatedSuite* starts its execution, it will set such a flag, and after everything is complete, it will release it.

4.5.3 @ParallelSuite

Additionally, we will have to design a mechanism for running multiple test suites in parallel. One way how to tackle this problem is to create an annotation that overrides configuration same as *@ParallelTest* that will contain an *@Execution* annotation with the value *CONCURRENT* and thus guaranteeing parallel execution. Nevertheless, we would not be able to configure method-wide parallelisation with such an approach. So the final solution is to override configuration using system property[9], when we need it. For instance, we left the default value system property for method-wide parallelisation (i.e., *same_thread*). By contrast, we set it to *concurrent* if we need to execute test suites in parallel. At the same time, we supply metadata in the form of @ParallelSuite annotation to these classes, which can be run in parallel with other classes.

[9] junit.jupiter.execution.parallel.mode.classes.default

Chapter 5

Implementation

This chapter is devoted to the implementation of additional functionality (i.e., parallelism) into the test framework within the Strimzi project. Implementation Listings (e.g, 5.1, 5.2...) are presented in Java programming language. Moreover, in Section 5.1 we describe an implementation of the first possible level of parallelism for more minor instances of the Kubernetes cluster. Finally, for more comprehensive instances (i.e., multi-node Kubernetes clusters), we explain the implementation of even higher-level parallelism in Section 5.2.

The author contributed the given code to the open-sourced project Strimzi, available on Github[1]. Specifically, these changes can be found in the *systemtest* module[2]. Installation and configuration of individual parallelisation levels are described in Appendix A. Eventually, we move more complex and extensive code snippets into Appendix B.

5.1 Stage 1 – method-wide parallelisation

In this section, we describe the solutions of the individual steps proposed in Section 4.4, which were necessary to perform an adaptation to method-wide parallelisations. We start with an explanation of how to resolve the uniqueness of test resources in Section 5.1.1. Furthermore, we describe the core implementation and the necessary reworking of test resources, as well as *ResourceManager* in Section 5.1.2. Next, in Section 5.1.3 the author present a mechanism that determines whether a given test case has to be executed in parallel or in isolation. Finally, in Section 5.1.4 we explain how such parallelisation can be configured, and in Section 5.1.5 we describe its usability within our infrastructure.

5.1.1 Unique Naming for each resource[3]

Several sources (e.g, *Kafka cluster*, *KafkaConnect*, *KafkaMirrorMaker*), which are used in test cases, are necessary to work with unique names to avoid conflict (e.g, replace existing or already created resources). That is why we created the class *TestStorage*[4], which will include unique generated name of the necessary resources (e.g, name of the *Namespace*, *Kafka cluster*, *KafkaTopic*, *Producer*, *Consumer*). All is possible thanks to *ExtensionContext* object, where each test case has a different *ExtensionContext*, and therefore it can be used as a unique identifier between test cases. Eventually each test case has to instantiate

[1]Strimzi Github repository – https://github.com/strimzi/strimzi-kafka-operator
[2]systemtest module – https://github.com/strimzi/strimzi-kafka-operator/tree/main/systemtest
[3]Upstream pull request – https://github.com/strimzi/strimzi-kafka-operator/pull/4092
[4]TestStorage – https://github.com/strimzi/strimzi-kafka-operator/pull/5446/

TestStorage class and then access resources (e.g, *Namespace*, *Kafka cluster*, *KafkaTopic*, *Producer*, *Consumer*).

5.1.2 Resource Manager re-work[5]

As described in Section 4.3, we created Interface *ResourceType <T extends HasMetadata>*. Where *T* is a generic type and can take subtypes (e.g, Kafka, KafkaBridge, KafkaMirrorMaker). In other words, everything that contains the object *HasMetadata*[6]. Listing 5.1 shows the individual method signatures in the given interface.

```
public interface ResourceType<T extends HasMetadata> {
    String getKind();
    T get(String namespace, String name);
    void create(T resource);
    void delete(T resource);
    boolean waitForReadiness(T resource);
}
```

Listing 5.1: Interface used across all resources

Each resource then signs a contract with the ResourceType interface in our test framework. For instance, the *Kafka* resource implementation (Listing 5.2).

```
public class KafkaResource implements ResourceType<Kafka> {
    @Override
    public String getKind() { return Kafka.RESOURCE_KIND;}
    @Override
    public Kafka get(String namespace, String name) {...}
    @Override
    public void create(Kafka resource) {...}
    @Override
    public void delete(Kafka resource) {...}
    @Override
    public boolean waitForReadiness(Kafka resource) {...}
    // implementation of each methods omitted for clarity
}
```

Listing 5.2: Kafka resource sings contract with ResourceType interface

Nevertheless, the most critical part of the entire Strimzi test framework is *ResourceManager*. As described in the design 4.3.2, instead of three stacks (i.e., pointer, class, and method), we had to adapt a solution with hash maps, which for each test case will keep each stack in which will contain the test resources. At the same time, thanks to the proposed algorithms (4, 6), the algorithm for creating resources according to the generic type *T* finds out which method to invoke. Moreover, a parallel algorithm for deleting individual resources from a given stack. Finally, the 5 algorithm for synchronisation of parallel generating resources is most useful in the parallel preparation of individual resources for a given test case. An example of such a preparation phase (Listing 5.3).

[5] https://github.com/strimzi/strimzi-kafka-operator/pull/4137

[6] HasMetadata – is an interface of Kubernetes resources that contain metadata object

```
// create resources in parallel (simultaneously)
resourceManager.createResource(extensionContext, false,
    KafkaTemplates.kafka().build()
    KafkaTemplates.kafkaWithMetrics().build(),
    KafkaMirrorMakerTemplates.kafkaMirrorMaker().build(),
    KafkaConnectTemplates.kafkaConnect().build(),
    KafkaClientsTemplates.kafkaClients().build()
);
// synchronize point (barrier)
resourceManager.synchronizeResources(extensionContext);
```

Listing 5.3: Example of parallel preparation of resources

The overall implementation of individual algorithms (4, 5 and 6) can be seen in the Appendix B.

5.1.3 Injection of the runtime annotations

Another crucial part is creating a mechanism that will provide information, which test case may be executed in parallel mode or run in complete isolation. In Section 4.4, we propose such annotations offered by the Java language. We implemented three types of annotations for method-wide parallelisation. The most concise annotation is *@ParallelTest*, which overrides the parallelism configuration at runtime. It is possible to see the given implementation of such an annotation on Listing 5.4. An essential part is *@Execution(ExecutionMode.CONCURRENT)*, where the semantics of this line means that the given annotation will overwrite the given configuration from a sequential mode to parallel mode and thanks to *@Retention(RUNTIME)* it will do so at runtime.

```
@Target(ElementType.METHOD)
@Retention(RUNTIME)
@Execution(ExecutionMode.CONCURRENT)
@ResourceLock(mode = ResourceAccessMode.READ, value = "global")
@Test
public @interface ParallelTest { }
```

Listing 5.4: Implementation of the @ParallelTest annotation

Another annotation (Listing 5.5) we have implemented to be responsible for the complete isolation of is *@IsolatedTest*. At an initial glance, it is remarkably similar to the previous annotation. However, there is one major difference when using *@ResourceLock*. When *@ParallelTest* uses a read lock, *@IsolatedTest* uses a *read_write* lock. The idea is that *read_write* lock will completely isolate us from other tests. Multiple *@ParallelTest* will be performed at the same time, and *@IsolatedTest* will wait until this lock is released (because these two annotations share the same @ResourceLock named *global*).

```
@Target(ElementType.METHOD)
@Retention(RUNTIME)
@Inherited
@ResourceLock(mode = ResourceAccessMode.READ_WRITE, value = "global")
@Test
public @interface IsolatedTest {
```

```
    String value() default ""; // reason why it needs isolation
}
```

Listing 5.5: Implementation of the @IsolatedTest annotation

Finally, we implemented the last annotation due to product requirements *@ParallelNamespaceTest* (i.e., each Kafka cluster has to be in its own namespace). This annotation is equivalent to *@ParallelTest*, but there is a slight distinction. We create an additional namespace for each test case. Scenarios where we mainly use it are when multiple Kafka clusters are deployed for a given test or when we use *KafkaMirrorMaker* (by default, we need two Kafka clusters).

5.1.4 Configuration

The method-wide parallelisation configuration can be set up in several ways (a) using system properties (Listing 5.16), (b) using the junit-platform.properties configuration file (Listing 5.17).

```
-Djunit.jupiter.execution.parallel.enabled = true
-Djunit.jupiter.execution.parallel.config.fixed.parallelism = <n>
// parallel.mode.default has default value same_thread
// parallel.mode.classes.default has default value same_thread
```

Listing 5.6: (a) Configuration via system properties

In both cases, n threads will be released, where each thread will perform one test case at a time, and if it finishes its work, it will move on to the next test case. This is repeated until there is no more test to execute in the given test class.

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = same_thread
junit.jupiter.execution.parallel.config.strategy = fixed
junit.jupiter.execution.parallel.config.fixed.parallelism = <n>
```

Listing 5.7: (b) Configuration via file

5.1.5 Application

Method-wide parallelisation for our testing framework is most efficient for more diminutive infrastructures, typically with parameters e.g, 24GB RAM and eight cores. We use such infrastructure as part of nightly testing. In the circumstances, we have less power available; it is necessary to count on it that in more demanding test cases (i.e., a test case using *KafkaMirrorMaker* or several Kafka clusters), the cluster will be unstable, which will lead to poor test results and overall test timeouts. On the other hand, in the case of more powerful infrastructure (i.e., multi-node Kubernetes cluster), it is possible to use the following form of parallelism.

5.2 Stage 2 – class-wide parallelisation

In this section, we explain the solutions of the individual steps proposed in Section 4.5, which were necessary to perform for adaptation to class-wide parallelisations. At first we describe

complete re-work of Cluster Operator installation in Section 5.2.1. Next, in Section 5.2.2 and Section 5.2.3 we exemplify mechanism for isolation of test suites and describe common problems in relation of isolation. Moreover, we present another component required for class-wide parallelisation and management of all *Namespaces* in Section 5.2.4. Finally, in Section 5.2.5 we explain how such parallelisation can be configured, and in Section 5.2.6 we describe its usability within our infrastructure.

5.2.1 Deployment of shared Cluster Operator across all suites

As we proposed in Section 4.5.1 for class-wide parallelisation, we want to create a shared Cluster Operator alongside all test classes. Such an approach is possible thanks to root *ExtensionContext*, which guarantees that the Cluster Operator instance will be deleted only after the overall execution. We will describe the two primary phases of our proposed JUnit5 extension (i.e., *BeforeAllOnce* setup phase – Listing 5.8). The method must use synchronised to prevent multiple threads in a race condition. This situation would occur if two or more *@ParallelSuite* threads passed the *!BeforeAllOnce.systemReady* condition and then started to create an instance of CO twice. At the same time, we may notice that it is necessary to change the configuration of the given Cluster Operator to different configurations.

```
synchronized private static void systemSetup(
    ExtensionContext extensionContext) {
    if (!BeforeAllOnce.systemReady) {
        sharedExtensionContext = extensionContext.getRoot();
        if (StUtils.isParallelSuite(extensionContext)) {
            BeforeAllOnce.systemReady = true;
            if (Environment.isNamespaceRbacScope() &&
                !Environment.isHelmInstall()) {
            clusterOperator = new SetupClusterOperator
                .SetupClusterOperatorBuilder()
                .withExtensionContext(sharedExtensionContext)
                .createInstallation()
                .runInstallation();
            } else {
            // setup cluster operator before all suites only once
                clusterOperator = new SetupClusterOperator
                .SetupClusterOperatorBuilder()
                .withExtensionContext(sharedExtensionContext)
                .withNamespace(Constants.INFRA_NAMESPACE)
                .withWatchingNamespaces(Constants.WATCH_ALL_NAMESPACES)
                .createInstallation()
                .runInstallation();
            }
        }
        sharedExtensionContext.getStore(ExtensionContext.Namespace.GLOBAL)
            .put(SYSTEM_RESOURCES, new BeforeAllOnce());
    }
}
```

Listing 5.8: Setup phase of shared Cluster Operator

The last essential aspect is the last line in Listing 5.8, where we create an instance of the given extension and thus implicitly call the *close()* method. The class implements the *Autocloseable* Interface, which ensures that such a method is called at the end of an instance's life. Inside the *close()* method, we have to reset the flag counter and also call the *Singleton*[7] instance of the shared Cluster Operator to uninstall all components (Listing 5.9).

```
public synchronized void close() throws Exception {
    BeforeAllOnce.systemReady = false;
    // complete un-install all components
    SetupClusterOperator.getInstanceHolder().unInstall();
}
```

Listing 5.9: Teardown phase of shared Cluster Operator

Furthermore, recall from Section 4.5.1, when we proposed the unification of the Cluster Operator configuration option via the Builder design pattern. Due to the numerous factory methods already becoming difficult to manage, it was necessary. The overall implementation without auxiliary methods (omitted for brevity) can be found in Listing B.5.

The last part we proposed in Section 4.5.1 was the *Rollback* algorithm (7). This is necessary if we have a situation of two classes that require a distinct configuration of the Cluster Operator e.g, Thread A terminates the execution of test class X and thread B will start the execution of test class Y, which needs a default configuration[8] that differs from the current. In such a situation, we trigger the *Rollback* algorithm.

5.2.2 Isolation of test Suites

The problem of isolating several classes that need different Cluster Operator configurations has been described and explained in Section 4.5.2. We implemented an annotation, which primarily serves as a label for that class.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE })
public @interface IsolatedSuite { }
```

Listing 5.10: Implementation of the @IsolatedSuite annotation

Moreover, we have implemented an additional mechanism that will ensure synchronisation between such test suites (i.e., *@IsolatedSuite*. We create *SuiteThreadController* class, and one of the synchronisations that are implemented inside this class is for *@IsolatedSuite*. If *@IsolatedSuite* starts its execution, it sets the given boolean value to true, which prevents the following thread from going through the given while loop. If *@IsolatedSuite* completes its execution, it sets the given boolean value to false, and the following thread will be able to start its execution. Furthermore, we notice the keyword *synchronised* in the method

[7]Singleton pattern – it is one of the creational design patterns, restricting instantiation of the class to only one instance. Thus invocation of *getInstanceHolder* method results always in the same instance.

[8]Default configuration – Such configuration is used alongside with *@ParallelSuite*. Thus if @IsolatedSuite ends its execution and starts @ParallelSuite, it always comes with triggering *Rollback* algorithm.

definition. The main reason why this keyword is necessary is that for multiple threads (i.e., *@IsolatedSuite*), they will have to wait until the thread that is currently in the loop is dropped and unlocks the lock, which implicitly adds *synchronised* (Listing 5.11).

```
public synchronized void waitUntilEntryIsOpen(
    ExtensionContext extensionContext) {
    // only one thread at a time
    while (this.isOpen.get()) {
        // Suite Y is waiting to lock to be released.
        Thread.currentThread().sleep(...);
    }
    // Suite X has locked the @IsolatedSuite and other
    // @IsolatedSuites must wait until lock is released.
    this.isOpen.set(true);
}
```

Listing 5.11: Implementation of the @IsolatedSuite synchronisation mechanism

5.2.3 SuiteThreadController

As mentioned in Section 5.2.2, the *SuiteThreadController* class provides multiple synchronizations. We have already described the first type in the previous Section for *@IsolatedSuite* classes. However, there are several other scenarios between classes (i.e., @ParallelSuite and @IsolatedSuite) that can occur:

1. case – only *@ParallelSuite* will be executed, (no need synchronisation)
2. case – only *@IsolatedSuite* will be executed, (Section 5.2.2)
3. case – several *@ParallelSuite* will be executed followed by a few *@IsolatedSuite*,
4. case – several *@IsolatedSuite* will be executed followed by a couple of *@ParallelSuite*,
5. case – *@ParallelSuite* starts, then *@IsolatedSuite* and finally *@ParallelSuite*,
6. case – *@IsolatedSuite* starts, then *@ParallelSuite* and finally *@ParallelSuite*.
7. case – ForkJoinPool spawning additional threads, which exceeding our configured parallelism limit.

In the first case, it is clear that we will not need any synchronisation between classes because they all use the same Cluster Operator configuration. Nevertheless, in the third case, it is necessary to provide some form of synchronisation. The scenario that could occur is that @IsolatedSuite would be the last to be executed, and at the same time, our testing framework runs a few @ParallelSuite. However, this means nothing for @IsolatedSuite, because no lock is attached to it, and it would start modifying the shared Cluster Operator and thus disrupt the execution of @ParallelSuite classes. Therefore, we have implemented an atomic counter into the *SuiteThreadController* class, which will increase if the thread starts executing @ParallelSuite and decreases as soon as it completes. Subsequently, the thread (i.e., @IsolatedSuite) that wants to start executing will not be able to start until the @ParallelSuite counter is equal to zero. In this case, the previous possible pessimistic scenario is eliminated (Listing 5.12).

```
public void waitUntilZeroParallelSuites(
    ExtensionContext extensionContext) {
```

```
    // until more that 0 parallel suites running in parallel 'active
       waiting'
    boolean preCondition = true;
    while (preCondition) {
        Thread.sleep(...);
        // runningTestSuitesInParallelCount variable is
        // changed by other threads (i.e., @ParalleSuites)
        preCondition = runningTestSuitesInParallelCount.get() > 0;
    }
}
```

Listing 5.12: @ParallelSuite and @IsolatedSuite synchronisation mechanism

For the fourth case, when we start executing a few @IsolatedSuite and then @ParallelSuite, we can reduce this to the problem when we only run the @IsolatedSuite (2nd case) because it is necessary to synchronise between @IsolatedSuite and none for @ParallelSuite.

In the fifth case, when we begin to execute @ParallelSuite, then several @IsolatedSuite, we eventually start running several @ParallelSuite. Thus, this includes the combination of synchronisation from cases 2 and 3. The analogy for the sixth case is a combination of these two cases.

In the last case, synchronisation is required when ForkJoinPool spawns multiple threads that exceed our configured parallelism limit. It does this because *ForkJoinPool* uses a *worker-steal* algorithm. Unfortunately, this technique spawns additional threads when synchronisation primitive blocks thread (e.g, this can lead to a situation where the user sets a fixed value of parallelism to two when he expects to run at most two test classes with two test cases. However, in some borderline situations, it could be that ForkJoinPool will spawn five threads instead of two. Because of this, we have implemented an additional mechanism that will make such threads sleep if they exceed the value of parallelism (Listing 5.13). Noteworthy is that if @ParallelSuite completes its execution, then in @AfterAll (i.e., at the end of the test class), it notifies and sets the value of *isParallelSuiteReleased* to true. Thus allow one of the waiting @ParallelSuite to start its execution.

```
public void waitUntilAllowedNumberTestSuitesInParallel(
    ExtensionContext extensionContext) {
    final String testSuiteToWait =
        extensionContext.getRequiredTestClass().getSimpleName();
    waitingTestSuites.add(testSuiteToWait);

    // wait zone for threads, which exceed maximum
    // of allowed test suites in parallel
    while (!isRunningAllowedNumberTestSuitesInParallel()) {
        // waiting to proceed with execution but current thread
        // exceed maximum of allowed test suites in parallel
        Thread.currentThread().sleep(...);

        // release and lock again
        if (isParallelSuiteReleased.get()) {
            // lock
            isParallelSuiteReleased.set(false);
            // remove selected test suite to continue its execution
```

```
            waitingTestSuites.remove(testSuiteToWait);
            break;
        }
    }
    // selected test suite is released
}
```

Listing 5.13: Additional synchronization for multiple @ParallelSuite that exceed our configured parallelism limit and are spawned by ForkJoinPool

5.2.4 TestNamespaceManager

Another component that is required for class-wide parallelisation is *TestNamespaceManager*. Recall the @ParallelNamespaceTest annotation, creating its namespace for such a test case. Also, assume the situation of multiple test classes (i.e., @ParallelSuites) running in parallel. If we run more than one such class, it is necessary to ensure that each test class operates with its namespace. Of course, we can think of several approaches to solve this problem by using static information to define separate Namespaces for each test suite that would need it. However, it would require a manual approach when adding another such test class. Another approach, using dynamic information, would be to find out which classes will need such a namespace at runtime. We obtain this information using the recursive method we implemented, which obtains all @ParallelSuite (Listing 5.14).

```
private void retrieveAllSystemTestsNames(File stFiles) {
    if (stFiles.getName().endsWith(Constants.ST + ".java") &&
        !stFiles.getName().contains(Constants.ISOLATED)) {
        this.stParallelSuitesNames.add(stFiles.getName());
    }
    File[] children = stFiles.listFiles();
    if (children == null) {
        return;
    }
    for (File child : children) {
        retrieveAllSystemTestsNames(child);
    }
}
```

Listing 5.14: Dynamically list all @ParallelSuites

Subsequently, we will make a namespace for each of these classes and store it in a hash map where the key will be the class name (e.g, *TracingST.getClass().getName()*), and if the given test class wants to get this namespace it will do so simply as illustrated in Listing 5.15.

```
@ParallelSuite
class TracingST {
    private final String namespace =
        testSuiteNamespaceManager.getMapOfAdditionalNamespaces()
            .get(TracingST.class.getSimpleName())
            .stream().findFirst().get();
```

```
    // other attributes ommited for brevity...
}
```

Listing 5.15: @ParallelSuite query generated (dynamically) namespace

5.2.5 Configuration

Recall from Section 5.1.4, where we present configuration of the method-wide parallelisation. To enable class-wide parallelisation, we can use two ways same as describe in Section 5.1.4 but with different values (a) using system properties (Listing 5.16), (b) using the junit-platform.properties configuration file (Listing 5.17). Note that we have to override system property parallel.mode.classes.default to concurrent. With that we can run multiple test classes simultaneously. Moreover, we may notice that the system property parallel.mode.default is not concurrent. That is because we override this value using the annotations @ParallelTest and @ParallelNamespaceTest.

```
-Djunit.jupiter.execution.parallel.enabled = true
-Djunit.jupiter.execution.parallel.mode.classes.default= concurrent
-Djunit.jupiter.execution.parallel.config.fixed.parallelism = <n>
```

Listing 5.16: (a) Configuration via system properties

```
junit.jupiter.execution.parallel.enabled = true
junit.jupiter.execution.parallel.mode.default = same_thread
junit.jupiter.execution.parallel.mode.classes.default = concurrent
junit.jupiter.execution.parallel.config.strategy = fixed
junit.jupiter.execution.parallel.config.fixed.parallelism = <n>
```

Listing 5.17: (b) Configuration via file

5.2.6 Application

The class-wide parallelism for our testing framework is used mainly in the more extensive infrastructures we have at our disposal (i.e., multi-node Kubernetes or Openshift cluster, where each node has 8 CPU cores and 16GB RAM). Therefore, the implemented solution was tested on AWS infrastructures and, at the same time Openstack. The question is, what is the optimal configuration for each infrastructure (Listing 5.1). The next chapter, which is devoted to experimentation, can answer such a question.

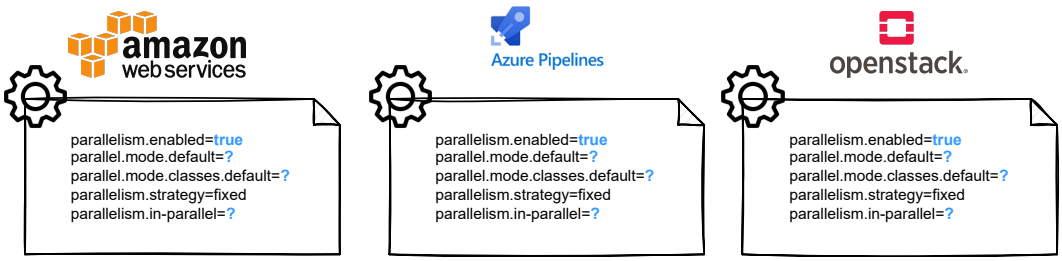


Figure 5.1: Find optimal configuration for each infrastructure

Chapter 6

Experimental evaluation

This chapter is devoted to testing and experimentally evaluating of proposed parallel execution designed and implemented in Chapters 4, 5. In addition, we designed experiments to prove the parallelism we created scales (i.e., method or class-wide).

6.1 Experiments design

The overall design of the experiments is divided into three main categories; (a) preliminary experiments to prove that the parallelisation we propose is capable of vertical scaling. These experiments will be performed for small Kubernetes instances (i.e., Minikube) and multi-node Kubernetes clusters. The expected results should be positive because parallelisation will have the best possible implementation environment (e.g, for method-wide parallelisation, it will be a test class containing only tests that are capable of parallel computation, similarly to class-wide parallelisation.); (b) the next part will be the acceptance of production-based experiments, which will primarily provide information on whether it is beneficial to use parallelisation in a small subset of tests, where mostly half of the tests are capable of parallel execution. Acceptance experiments will include a subset of our system of tests, where of course, there will also be tests and test classes, which are not capable of parallel execution, and thus synchronisation will occur. Possibly the parallelisation will not be suitable for acceptance experiments because most test suites consist of one or two test cases, and the overall preparation phase of the test suite is long.; (c) the last type of experiment, socalled regression, will already include the entire test suite currently offered by the Strimzi project. It will tell us whether the given parallelisation is eligible for the Strimzi. Moreover, a significant acceleration is expected because test classes often contain ten or more tests. On the other hand, we also have many tests that need total isolation, which potentially can slow down the whole performance.

We mainly use the Openstack and Amazon Web Services infrastructures to perform all the experiments, which will provide us with the necessary hardware resources. Furthermore, for preliminary experiments, we use four types of instances:

- **Kubernetes cluster** – multi-node, where this instance will provide 24 virtual cores and 48 GB RAM (without taking into account master nodes)
- **Minor instance of minikube** – single-node, where this instance will provide two virtual cores and 8GB RAM

- **Typical instance of minikube** – single-node, where this instance will provide four virtual cores and 16GB RAM
- **Comprehensive instance of minikube** – single-node, where this instance will provide eight virtual cores and 32GB of RAM

6.2 Preliminary experiments

Recall 3.1 Amdahl's formula from Chapter 3. We will not count the unit of work as the number of tests capable of parallel execution, but we will use a more accurate way (i.e., execution time). We also introduce a new formula (4), which also calculates the theoretical time after acceleration and then, thanks to this the result, we calculate the total possible acceleration using the formula (5). All markings are the same as described in Chapter 3 under Amdahl's law; we have T_{new} and T_{old}. T_{old} describes the time necessarily performed (i.e., sequentially) by a given task. On the other hand, T_{new} describes the time after acceleration Equation (4)

$$T_{new} = (1 - p) * T_{old} + \frac{p}{s} * T_{old} \tag{4}$$

$$S = \frac{T_{old}}{T_{new}} \tag{5}$$

In the case of our experiment, we have the test class **SecurityST**, which includes twenty-one test cases. All these tests can be performed in parallel and are a perfect candidate to obtain information that parallelisation is capable of vertical scaling. What should be noted is the fact that the shared Cluster Operator is deployed before the execution tests, where usually this deployment lasts from one to six minutes (we choose a mean value of three minutes). So in our case, the part that can be parallelised will be equal to $p = \frac{171}{174}$. The first instance we use is a multi-node Kubernetes cluster with 24 virtual cores and 48 GB of RAM. Empirically, we obtained data on how long it takes to complete a given test class sequentially, using such information in Amdahl's law.

$$T_{new} = (1 - \frac{171}{174}) * 174 + \frac{\frac{171}{174}}{24} * 174 = \ 10 \ minutes \tag{6}$$

In Equation (6), one can see the theoretical time we should approach in first experiments executing **SecurityST** test suite. Furthermore, the entire acceleration could be up to 17 times (i.e, Equation (7)). Of course, we know from practice that we will not get exactly such an acceleration; we can solely get nearer to it.

$$S = \frac{174}{10} = 17.4x \tag{7}$$

Additionally, we use the following notation in the tables:

- ✗ – disabled parallelism (e.g, method or class-wide), or test execution containing errors (e.g, cluster crashed, because of out of memory problem)
- ✓ – enabled parallelism (e.g, method or class-wide), or test execution without any issues
- ⚠ – test execution with flaky tests because of resource capacity

In the following Table 6.1, we can see the individual preliminary experiments performed over our implementation. For clarity, a sequential variant is also included. We slowly increased the threads used to determine if a given parallelisation scales there (i.e., we started from two to sixteen). As part of our experimentation, we found that up to twelve threads would be the best candidate for **SecurityST**. As shown in Table 6.1, when using sixteen threads, the given Kubernetes cluster was destroyed. The reason was mainly the capacity resources (i.e., we deploy Kafka cluster and many other resources for each test case). At the same time, we can notice that we did not reach the theoretical acceleration that we calculated in Equation (6). However, this is due to several factors (e.g, tests do not take the same time or slower deployment volumes within Kafka clusters). Nevertheless, one needs to realise that if we had hypothetically unlimited resources (i.e., cores, RAM), we would not be able to overcome the acceleration we calculated (i.e., Equation (8)).

Number of Threads	Method-wide	Class-wide	Resource issues	Execution time
1	✗	✗	✓	02:54 h
2	✓	✗	✓	01:40 h
3	✓	✗	✓	01:04 h
4	✓	✗	✓	50:58 min
5	✓	✗	✓	43:53 min
6	✓	✗	✓	39:42 min
7	✓	✗	✓	33:22 min
9	✓	✗	✓	28:06 min
10	✓	✗	✓	27:56 min
12	✓	✗	✓	**24:32 min**
13	✓	✗	⚠ (7 flake test cases)	01:54 h
16	✓	✗	✗ (Cluster crashed)	...

Table 6.1: The **SecurityST** contains twenty-one test cases, and all of them could be executed in parallel (i.e., contains @ParallelTest or @ParallelNamespaceTest annotation). Moreover, each test case deploys a Kafka cluster, which perfectly verifies if the Kubernetes cluster or Minikube (i.e., single-node) can handle such a load.

$$\lim_{s \to \infty} S_{max} = \frac{1}{1-p} = \frac{1}{1-\frac{171}{174}} = 58x \tag{8}$$

Our acquired acceleration in a perfect environment is less than $S_{\max} = 58x$ and at the same time $S_{teo} = 17.4x$. However, this is confirmed by the fact that we will never be better than $S_{\max}$ and also, we will never achieve a possible theoretical acceleration (i.e., S_{teo}) because such results are entirely typical for this kind of experiment. Overall, our acceleration is $S_{practical} = \frac{174}{24.5} = 7.1x$, which proves following relation $S_{practical} < S_{teo} < S_{\max}$.

Other preliminary experiments we performed were on more minor instances where it was a matter of course that the results accelerations compared to a multi-node cluster

will be significantly lower and slower. Therefore, Amdalh's law will also contain a much lower theoretical acceleration. For a machine containing four virtual cores, the estimated theoretical time is $T_{new\_teo\_medium}$, which is equal to $T_{new\_teo\_medium} = (1 - \frac{182}{185}) * 185 + \frac{\frac{182}{185}}{4} * 185 =$ approximately 49 minutes. So the theoretical acceleration of the instance could be $S_{new\_teo\_medium} = \frac{185}{49} = 3.8x$. Nevertheless, as we can see in Table 6.2, we did not accomplish such a same acceleration. However, we have come close enough, and the practical acceleration is $S_{new\_practical\_medium} = \frac{185}{79} = 2.34x$. We could use a maximum of three cores because, in the case of four cores, the virtual machine crashes due to a lack of memory. CPU utilisation was approximately 80% during the use of the four cores.

Number of Threads	Method-wide	Class-wide	Resource issues	Execution time
	Flavour:	8GB RAM	2 vCPUs	
1	✗	✗	✓	**03:19 h**
2	✓	✗	✗ (Cluster crashed)	...
	Flavour:	16GB RAM	4 vCPUs	
1	✗	✗	✓	03:05 h
2	✓	✗	✓	01:45 h
3	✓	✗	✓	**01:19 h**
4	✓	✗	✗ (Cluster crashed)	...
	Flavour:	32GB RAM	8 vCPUs	
1	✗	✗	✓	03:04 h
2	✓	✗	✓	01:46 h
3	✓	✗	✓	01:16 h
4	✓	✗	✓	59:52 min
5	✓	✗	✓	49:16 min
6	✓	✗	✓	**48:16 min**
7	✓	✗	✗ (Cluster crashed)	....

Table 6.2: Multiple experiments for various flavours of single-node Kubernetes instances for the **SecurityST** suite. Both of these flavours (i.e., orange and red one prove that parallelisation is vertically scaling on more minor instances), the yellow one (i.e., using two virtual cores and eight GB RAM) is not able to run either two test cases in parallel resulting in OOM problem (i.e., Out of memory).

We also have done other experiments to prove that our implemented class-wide parallelisation is capable of vertical scaling. Therefore, we selected a set of test classes that do not need any form of synchronisation or isolation (that is, they do not contain @IsolatedSuite annotation). Specifically, these will be classes containing the @ParallelSuite annotation, and they are HttpBridgeScramShaST, HttpBridgeTlsST, ThrottlingQuotaST, TopicST, UserST, ReconciliationST and CruiseControlConfigurationST. Together they con-

tain thirty test cases where twenty-nine do not need any form of synchronisation, and only one test case needs isolation from other tests. More precisely, we have ten @ParallelNamespaceTest, for repetition; these are tests that deploy the Kafka cluster and thus rank among the more resource-intensive. Next, we have 19 @ParallelTest can also be said to be lightweight variants on the need for total resources, and finally, one @IsolatedTest guaranteeing isolation from other parallel tests. In case we would like to calculate a possible theoretical acceleration, it is necessary to know the sequence time and, at the same time, the time of one @IsolatedTest. The total time of our selected tests is 107 minutes, of which @IsolatedTest lasts two and a half minutes. If we add the preparation time of the shared Cluster Operator to this, we get to five and a half minutes and therefore, the possible parallel time will be equal to $p = \frac{101.5}{107}$. The speedup factor is equal to the number of virtual CPUs we have available (i.e., $S = 24$), and thanks to that, all values can be set to formula as defined above (i.e., Equation (4)).

$$T_{new} = (1 - \frac{101.5}{107}) * 107 + \frac{\frac{101.5}{107}}{24} * 107 = \ 10 \ minutes \tag{6}$$

After the calculation, it turns out that the theoretical acceleration using twenty-four cores will approach ten minutes, and by this outcome, we can compute theoretical speed up which is $S_{teo} = \frac{T_{old}}{T_{new}} = \frac{107}{10} = \ 10.7x$ and $\lim_{s\to\infty} S_{\max} = \frac{1}{1-p} = \frac{1}{1-\frac{101.5}{107}} = \ 19x$.

What should be noted is that for class-wide parallelisation, the best possible scenario is to have a consistent test distribution. Ideally, such distribution where most test cases support parallel execution and in each test class are enough tests. (i.e., have test classes containing more minor parallel test cases than configured parallelism). This gives us the most out of the given type of parallelisation. For instance, suppose that we have five test classes, and each of them will have two tests (these tests will be capable of parallel execution). The best possible scenario would be to run such a set of tests with ten threads, guaranteeing that all threads will be busy. However, the test classes generally do not provide such an even distribution, which is almost impossible in practice (i.e., have the same number of tests for each test class).

Number of Threads	Method-wide	Class-wide	Resource issues	Execution time
1	✗	✗	✓	01:47 h
5	✓	✓	✓	43:53 min
6	✓	✓	✓	34:06 min
10	✓	✗	✓	**01:06 h**
10	✓	✓	✓	**29:49 min**
15	✓	✓	⚠ 1 flake	39:21 min

Table 6.3: Experiments aimed at class-wide parallelisation and one execution for method-wide by which we compare these two approaches and found non-correlation. Overall thirty test cases were executed (i.e., nineteen @ParallelTest, ten @ParallelNamespaceTest and one@IsolatedTest).

The experiments we performed can be seen in Table 6.3 similar to method-wide we first added a total sequential run to the parallelisation; then, we increased the number of

threads. Furthermore, we also compared the implementation of method-wide (i.e., orange row colour), where the overall implementation took significantly more than in the case of class-wide parallelisation (i.e., green row colour). The main reason why the use of ten threads of class-wide parallelisation was more than half an hour better was because not more than ten tests were in each test class, and therefore unnecessarily, many threads were used in method-wide parallelisation, which were not actually used. On the other hand, class-wide parallelisation has made full use of ten threads, as it can perform several classes simultaneously, thus significantly increasing the total time.

6.3 Production experiments

In this section, we will include two different experiments. First, in Section 6.3.1 they will be described as shown experiments on a small production set of system tests. Moreover, these experiments provide information on whether parallelisation is applicable even for a small production-based set of tests. In the next section (i.e., Section 6.3.2), we get a different view of the extensive set of system tests that are currently available in the Strimzi.

6.3.1 Subset of our Strimzi system test

This type of experiment will enclose our genuine subset of tests. These tests also give us whether it is beneficial to perform them parallel. Since we know that the profile also contains enough test cases for which isolation is necessary (i.e., @IsolatedTest) and test classes (i.e., @IsolatedSuite). Therefore, a significantly weaker acceleration is expected than the preliminary experiments, which have the best possible parallelisation environment. Furthermore, since we found out that flavour *2CPUs and 8GB RAM* is not able to perform even two parallel tests, it is thus unnecessary for this type of experiment.

So if we look at a more detailed way in our production-based tests, we find out precisely that it contains 13 test classes and 35 tests. Of these, 6 test classes require complete isolation (i.e., @IsolatedSuite) and the same for 5 test cases (i.e., @IsolatedTest). However, what is interesting is to be careful, and parallel tests could sometimes be in border situations taken as @IsolatedTest. One of the cases is where we have three isolated classes containing only one test that can perform the parallel implementation. This is because test cases in @IsolatedSuite will only be executed after the calculation is completed by @ParallelSuite or another @IsolatedSuite. We have eight tests that require complete isolation and 27 tests capable of parallel execution. However, this fact still does not guarantee that the problem will scale vertically. One reason is that the *MetricsIsolatedST* test class contains 18 parallel tests, where almost all of them do not last more than a few seconds. Thus, there will be little success in this class for parallelisation. At the same time, the overall test set is not an ideal sample for method-wide parallelisation because these are test classes that do not contain several tests. However, where it can be a potential success, the use of class-wide parallelisation is not great. Some paralleled classes (i.e., @ParallelSuite) contain pre-preparation of their test environment (i.e., *HttpBridgeTlsST*, *RollingUpdateST*). Furthermore, in the case of class-wide parallelisation, they create it independently of the second test class, meaning that where it will be possible to save time will be mainly in these parts. However, we do not think we will see an intense acceleration.

The experiments we performed on a given test sample do not differ much from the previous ones. We started with typical sequential execution and gradually increased the number of threads (see Table 6.4). We found out that during the use of method-wide parallelisa-

tion, there was no acceleration at all. Thus, a scenario where this form of parallelisation is not very suitable due to the small number of tests in the given test classes. However, where we could potentially succeed was by using class-wide parallelisation. Within the more powerless machine (i.e., *16GB RAM and four virtual CPUs*), the acceleration was almost non-existent, even in the class-wide case. On the other hand, using a more robust machine

Number of Threads	Method-wide	Class-wide	Resource issues	Execution time
	Flavour:	16GB RAM	4 vCPUs	
1	✗	✗	✓	01:36 h
2	✓	✗	✓	01:31 h
2	✓	✓	✓	01:31 h
3	✓	✓	✓	01:25 h
4	✓	✓	✓	**01:24 h**
5	✓	✓	✗ (Cluster crashed)	...
	Flavour:	32GB RAM	8 vCPUs	
1	✗	✗	✓	01:31 h
2	✓	✗	✓	01:30 h
3	✓	✓	✓	01:21 h
4	✓	✓	✓	01:18 h
5	✓	✓	✓	01:15 h
6	✓	✓	✓	01:06 h
7	✓	✓	✓	01:06 h
8	✓	✓	✓	**01:03 h**
9	✓	✓	✓	01:06 h

Table 6.4: Combination of experiments (i.e., using method and class-wide parallelisation) primarily aiming at class-wide parallelisation. Moreover, experiments were performed for more-minor instances of Kubernetes.

(i.e., *32GB RAM and eight virtual CPUs*), we could get to eight threads with only 1.4x acceleration, which is not very advantageous in using the number of resources needed for parallelisation. Hence, it is clear that more minor instances of Kubernetes are not suitable for this type of sample (i.e., acceptance production-based) due to the above facts.

At the same time, we wanted to try a similar scenario in the case of using a more significant instance of Kubernetes (i.e., multi-node). The results we obtained were the same as for the more minor instances of Kubernetes, also due to the size of the test set (i.e., Table 6.5).

Number of Threads	Method-wide	Class-wide	Resource issues	Execution time
1	✗	✗	✓	01:31 h
5	✓	✓	✓	01:04 h
7	✓	✓	✓	**01:03 h**
10	✓	✓	✓	01:12 h

Table 6.5: Experiments performed by using class-wide parallelisation for a more robust Kubernetes cluster.

6.3.2 Entire system tests of the Strimzi

The last type of experiment we tried was a regression (i.e., productionbased). In other words, a very robust set of test cases contains everything (i.e., @IsolatedTest, @IsolatedSuite, @ParallelTest, @ParallelSuite). Currently, this test sample contains approximately 65 test classes, and more than half of them are classes requiring synchronisation (i.e., @IsolatedSuite). Specifically, these are 37 isolated classes, and with the use of the add-on, we find that there are classes that can perform them at the same time precisely 28. This quantification can give us an approximate possible result of the experiments. Given that sequential execution takes almost twenty-one hours, the ideal scenario would be to get below half (i.e., ten hours) of execution time. A total of four types of experiments will be performed; (a) VM with 16GB RAM and four virtual cores, (b) VM with 32GB RAM and eight virtual cores, (c) Kubernetes cluster with six nodes (three masters and three workers) and (d) Kubernetes cluster with nine nodes (three masters and six workers). We do not expect much acceleration for (a) because this is a test suite when OOM may have a problem. This is mainly due to test cases containing the component *KafkaMirrorMaker* or *KafkaMirrorMaker2*, which needs a lot of RAM. On the other hand, for alternative (b), we already assume an acceleration approaching half. At the same time, however, it will perform method-wide parallelisation for both types of experiments. In another Kubernetes instance (c) type, we assume the possible implementation of at least five threads in parallel and, thus, some acceleration. Finally, for (d), it is a matter of course that the most significant possible acceleration is expected and, at the same time, the use of class-wide parallelisation for a vast number of threads.

In the first run of experiments for small instances of Kubernetes (i.e., using one VM), we obtained the following information. For VMs with 16GB RAM and four virtual cores, no form of parallelisation is possible because already with method-wide parallelisations with the use of two threads, the given VM falls on the lack of memory (Table 6.6). This is the case in test cases using the *KafkaMirrorMaker* or *KafkaMirrorMaker2* components. At the same time, the combination of *KafkaConnect* with *KafkaMirrorMaker/2*. Let us remember that one component of *KafkaMirrorMaker/2* requires two Kafka clusters, and hence is very memory intensive.

The second type of experiment using a more powerful VM was slightly more favourable in terms of results. We could even use four threads where the total time spent performing was 12h from the flood 21h. However, using five threads, we got into the same problem as in previous experiments (i.e., OOM problem), as shown in Table 6.6. Overall, it can be

Number of Threads	Method-wide	Class-wide	Resource issues	Execution time
	Flavour:	16GB RAM	4 vCPUs	
1	✗	✗	✓	20:33 h
2	✓	✗	✗ (Cluster crashed)	...
	Flavour:	32GB RAM	8 vCPUs	
1	✗	✗	✓	20:32 h
2	✓	✗	✓	14:24 h
3	✓	✗	✓	12:21 h
4	✓	✗	✓	**11:56 h**
5	✓	✗	✗ (Cluster crashed)	...

Table 6.6: Combination of experiments (i.e., using method parallelisation) on production-based test sample with different VMs.

assessed (for small instances of Kubernetes) that ideal candidates for this test sample will use VM with 32GB RAM and eight virtual cores.

Another form of experimentation (i.e., (c) and (d)) they were even more massive on resources. Therefore, we expected better results. For the Kubernetes cluster using three

Number of Threads	Method-wide	Class-wide	Resource issues	Execution time
	Flavour:	3 master nodes	**3 workers**	
1	✗	✗	✓	20:24 h
5	✓	✗	✓	**11:09 h**
10	✓	✗	✗ (32 test errors)	11:51 h
	Flavour:	3 master nodes	**6 workers**	
15	✓	✓	✓	09:53 h
20	✓	✓	✓	08:54 h
25	✓	✓	✓	**08:14 h**
30	✓	✓	✗ (Cluster crashed)	...

Table 6.7: Experiments performed by using method-wide and class-wide parallelisation for a more robust Kubernetes clusters (variation with three and six worker nodes).

master and three worker nodes, we got the best results using five threads and were we can get from 21h to 11h, as can be seen in Table 6.7. When we used more threads, the result was worse, or the test cases fell due to a lack of memory (e.g, using ten threads, we got 32 error tests).

The results were much better for the Kubernetes cluster using three master and six worker nodes. We could use up to twenty-five threads, which resulted in a decrease in computing time. Interestingly, almost half of the test classes that support class-wide parallelization (i.e., @ParallelSuites) improved its total exercise time from about eight hours to one hour, which results in up to 8-fold acceleration. On the other hand, we could not fully use the strength within the remaining thirty-seven test classes that need an isolation environment (i.e., @IsolatedSuite). It is evident that tests are performed in parallel in a given class, but occasionally there are five or fewer tests in a given class, which prolongs the entire execution time. From the sequential execution time (i.e., twenty-one hours), we got to eight hours (Table 6.7). Moreover, one hypothesis would undoubtedly improve the overall performance, but it would also worsen the overall readability of the test sample. For instance, if we were to re-structure the test suites, where we would be test classes (mainly @IsolatedSuite). Then we add test cases that require the same *Cluster Operator* configuration, and these test suites would contain a maximum of twenty-five test cases. It would significantly improve overall time because test classes will not contain less than five test cases, resulting in a situation where most threads are working. Therefore, we eliminate scenarios where we configure twenty-five threads running in parallel, and some test classes have only three or fewer test cases, meaning that the seventeenth threads are sleeping. Re-structuring our test classes can be very friendly at first glance. Unfortunately, we would sacrifice readability (e.g., tests that should belong to a separate test class we combine into some most similar), thus reducing scenarios where we can have @IsolatedSuite with less than five test cases. Nevertheless, we could end up in a scenario where these test cases in one @IsolatedSuite will not have any standard features. From a performance point of view, this can also be summarized by the following quote said by Donald Ervin Knuth.

> *Premature optimization is the root of all evil (or at least most of it) in programming.*

In the case of using 30 threads, we got to the problem of OOM and Kubernetes cluster crashing. We can evaluate that Kubernetes for a given test sample is the optimal candidate cluster using three master and six workers nodes for larger instances.

Chapter 7

Conclusion

The thesis began with a description of the basic principles of *Kubernetes* and *Kafka*. Furthermore, we described a project that encapsulates Kafka and uses it on top of the Kubernetes (i.e., Strimzi). We further explained the architecture and principles in the system tests of the Strimzi project. In addition, we gained knowledge about parallel execution, which we successfully used in this thesis. Moreover, we identified the challenges of the current system test architecture. Therefore, we designed and implemented a solution to solve these problems (i.e., using multiple pipelines and creating sub-sets of tests is not horizontally scalable due to our cloud services that provide resources). Thus, this information motivated us to design and implement a mechanism of fine-grained parallelism in our test framework (i.e., using memory and central processing units) that the cloud services offer us. Finally, our experiments showed that parallelisation could scale vertically for different test samples.

Based on the performed experiments, we found that within the environment that fully supports parallelisation, there were results the acceleration within factor 8. We obtained the identical factor in the case of production experiments (i.e., from eight to one hour) when performing only classes that support class-wide parallelisation (i.e., @ParallelSuite). By contrast, overall production experiments (i.e., together with @IsolatedSuite) showed partially more inadequate results (i.e., a factor of 2.5). We reach such a factor mainly due to the structure of the test classes and their content (i.e., the number of tests within the test class; meaning a @IsolatedSuite with fewer than five test cases when twenty-five threads in parallel are configured and thus twenty threads sleep). Furthermore, more than half of the classes require to perform in complete isolation (i.e., @IsolatedSuite).

We contributed the given code to the open-sourced project Strimzi, available on Github[1], which also makes it viable to inspire other *kube-native* products to enforce such solutions. Making parallel execution possible started from 0.23.0 (released in May 2021) to the 0.29.0 Strimzi version. Whether a method or class-wide parallelisation, both steps have been completed and merged into the main branch of the Strimzi project, where this implementation will be available from version 0.29.0. Our parallelism model of system tests is used in continuous integration systems (i.e., Jenkins, Azure Pipelines), where the overall computational time is much faster than in sequential computational computation (proved by experiments).

[1]Strimzi Github repository – https://github.com/strimzi/strimzi-kafka-operator

Bibliography

[1] AUTHORS, K. *Kubernetes* [online]. 2019 [cit. 2021-08-11]. Available at: https://mapr.com/products/kubernetes/assets/k8s-logo.png.

[2] AUTHORS, T. K. *Apache Kafka documentation* [online]. 2021 [cit. 2021-08-14]. Available at: https://kafka.apache.org/documentation/.

[3] AUTHORS, T. K. *History* [online]. 2019 [cit. 2021-08-11]. Available at: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/#going-back-in-time.

[4] AUTHORS, T. K. *Namespaces* [online]. 2019 [cit. 2021-08-11]. Available at: https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/.

[5] AUTHORS, T. K. *Service* [online]. 2019 [cit. 2021-08-11]. Available at: https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types.

[6] AUTHORS, T. S. *Strimzi blog posts* [online]. 2021 [cit. 2021-09-26]. Available at: https://strimzi.io/blog.

[7] AUTHORS, T. S. *Strimzi Kafka Operator* [online]. 2021 [cit. 2021-09-26]. Available at: https://strimzi.io/docs/operators/latest.

[8] GUSTAFSON, J. L. Amdahl's Law. In: PADUA, D., ed. *Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, p. 53–60. DOI: 10.1007/978-0-387-09766-4_77. ISBN 978-0-387-09766-4. Available at: https://doi.org/10.1007/978-0-387-09766-4_77.

[9] HERLIHY, M., SHAVIT, N., LUCHANGCO, V. and SPEAR, M. *The Art of Multiprocessor Programming*. 2nd ed. Morgan Kaufmann, 2020. ISBN 0124159508.

[10] INC., D. *Docker* [online]. 2016 [cit. 2021-08-11]. Available at: https://s3-us-west-2.amazonaws.com/com-netuitive-app-usw2-public/wp-content/uploads/2016/06/small_v-trans.png.

[11] MITCH, S. *Mastering Kafka Streams and ksqlDB Building real-time data systems*. 1st ed. O'Reilly Media, Inc., 2021. ISBN 1492062499.

[12] NEHA NARKHEDE, T. P. *Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale*. 1st ed. O'Reilly Media, 2017. ISBN 979-8703756065.

[13] ORSÁK, M. *Real Time Data Processing with Strimzi Project*. Brno, CZ, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/22425/.

[14] Orsák, M. *How system tests work* [online]. 2021 [cit. 2021-08-11]. Available at: https://strimzi.io/blog/2020/12/03/how-the-system-tests-works/.

[15] Orsák, M. *Introduction to system tests* [online]. 2021 [cit. 2021-08-11]. Available at: https://strimzi.io/blog/2020/09/21/introduction-to-system-tests/.

[16] Pacheco, P. *An Introduction to Parallel Programming*. 1st ed. Morgan Kaufmann, 2011. ISBN 0123742609.

[17] Poulton, N. *The Kubernetes Book*. 1st ed. Independently published, 2021. ISBN 979-8703756065.

[18] Scholz, J. *Deploying Kafka on Kubernetes with Local storage using Strimzi* [online]. 2018 [cit. 2021-09-26]. Available at: https://strimzi.io/blog/2018/06/11/deploying-kafka-on-kubernetes-with-local-storage-using-strimzi/.

[19] Stopford, B. *Designing Event-Driven Systems*. 1st ed. O'Reilly Media, Inc., 2018. ISBN 9781491990650.

Appendix A

Manual

The author assume that one has already prepare Kubernetes cluster and it is connected to such instance. Here is the a few steps how to run (a) method-wide or (b) class-wide parallelisation:

1. clone the Strimzi repository – git clone https://github.com/strimzi/strimzi-kafka-operator
2. enter the cloned repository – cd strimzi-kafka-operator
3. download needed utilities in directory development-docs/DEV_GUIDE.md
4. build Strimzi project – mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
5. (a) run system tests using five threads by method-wide parallelisation
   - mvn verify -pl systemtest -Pall -Djunit.jupiter.execution.parallel.enabled=true -Djunit.jupiter.execution.parallel.config.fixed.parallelism=10
6. (b) run system tests using five threads by class-wide parallelisation
   - mvn verify -pl systemtest -Pall -Djunit.jupiter.execution.parallel.enabled=true -Djunit.jupiter.execution.parallel.config.fixed.parallelism=10 -Djunit.jupiter.execution.parallel.mode.classes.default=concurrent
7. (optional) one can also run such parallelisation in the InteliJ IDE by specifying these properties inside configuration.

Appendix B

Implementation details

```
@SafeVarargs
public final <T extends HasMetadata> void createResource(
    ExtensionContext testContext,
    boolean waitReady, T... resources) {
    for (T resource : resources) {
        ResourceType<T> type = findResourceType(resource);
        LOGGER.info("Create/Update {} {} in namespace {}",
            resource.getKind(), resource.getMetadata().getName(),
            resource.getMetadata().getNamespace() == null ? "(not set)"
                : resource.getMetadata().getNamespace());

        // ignore test context of shared Cluster Operator
        if (testContext != BeforeAllOnce.getSharedExtensionContext()) {
            // if it is parallel namespace test we are gonna replace
            // resource a namespace
            if (StUtils.isParallelNamespaceTest(testContext)) {
                if (!Environment.isNamespaceRbacScope()) {
                    final String namespace = testContext
                        .getStore(ExtensionContext.Namespace.GLOBAL)
                        .get(Constants.NAMESPACE_KEY).toString();
                    LOGGER.info("Using Namespace: {}", namespace);
                    resource.getMetadata().setNamespace(namespace);
                }
            }
        }

        type.create(resource);

        synchronized (this) {
            STORED_RESOURCES.computeIfAbsent(testContext.getDisplayName(),
                k -> new Stack<>());
            STORED_RESOURCES.get(testContext.getDisplayName()).push(
                new ResourceItem<T>(
                    () -> deleteResource(resource),
```

```
                    resource
                ));
            }
        }

        if (waitReady) {
            for (T resource : resources) {
                ResourceType<T> type = findResourceType(resource);
                assertTrue(waitResourceCondition(resource,
                    ResourceCondition.readiness(type)),
                    String.format("Timed out waiting for %s %s in namespace
                    %s to be ready",
                    resource.getKind(),
                    resource.getMetadata().getName(),
                    resource.getMetadata().getNamespace()));
            }
        }
    }
```

Listing B.1: Complete thead-safe method for parallel creation resources

```
public void deleteResources(ExtensionContext testContext) throws Exception
    {
    LOGGER.info(String.join("", Collections.nCopies(76, "#")));
    if (!STORED_RESOURCES.containsKey(testContext.getDisplayName()) ||
        STORED_RESOURCES.get(testContext.getDisplayName()).isEmpty()) {
        LOGGER.info("In context {} is everything deleted.",
            testContext.getDisplayName());
    } else {
        LOGGER.info("Delete all resources for {}",
            testContext.getDisplayName());
    }

    // if stack is created for specific test suite or test case
    AtomicInteger numberOfResources =
        STORED_RESOURCES.get(testContext.getDisplayName()) != null ?
        new AtomicInteger(STORED_RESOURCES.get(
        testContext.getDisplayName()).size()) :
        // stack has no elements
        new AtomicInteger(0);
    while (STORED_RESOURCES.containsKey(testContext.getDisplayName()) &&
        numberOfResources.get() > 0) {
        STORED_RESOURCES.get(testContext.getDisplayName())
            .parallelStream().parallel().forEach(
            resourceItem -> {
                try {
                    resourceItem.getThrowableRunner().run();
                } catch (Exception e) {
                    e.printStackTrace();
```

```
                }
                numberOfResources.decrementAndGet();
            }
        );
    }
    STORED_RESOURCES.remove(testContext.getDisplayName());
    LOGGER.info(String.join("", Collections.nCopies(76, "#")));
}
```

Listing B.2: Complete thead-safe method for parallel deletion resources

```
public final <T extends HasMetadata> void synchronizeResources(
    ExtensionContext testContext) {
    Stack<ResourceItem> resources = STORED_RESOURCES.get(
        testContext.getDisplayName());

    // sync all resources
    for (ResourceItem resource : resources) {
        if (resource.getResource() == null) {
            continue;
        }
        ResourceType<T> type = findResourceType((T) resource.getResource());

        waitResourceCondition((T) resource.getResource(),
            ResourceCondition.readiness(type));
    }
}
```

Listing B.3: Complete thead-safe method for synchronize resources

```
private final ResourceType<?>[] resourceTypes = new ResourceType[]{
    new KafkaBridgeResource(),
    new KafkaClientsResource(),
    new KafkaConnectorResource(),
    new KafkaConnectResource(),
    new KafkaMirrorMaker2Resource(),
    new KafkaMirrorMakerResource(),
    new KafkaRebalanceResource(),
    new KafkaResource(),
    new KafkaTopicResource(),
    new KafkaUserResource(),
    new BundleResource(),
    new ClusterRoleBindingResource(),
    new DeploymentResource(),
    new JobResource(),
    new NetworkPolicyResource(),
    new RoleBindingResource(),
    new ServiceResource(),
    new ConfigMapResource(),
```

```
    new ServiceAccountResource(),
    new RoleResource(),
    new ClusterRoleResource(),
    new ClusterOperatorCustomResourceDefinition(),
    new SecretResource(),
    new ValidatingWebhookConfigurationResource()
};
```

Listing B.4: List of supported resources inside ResourceManager

```
public class SetupClusterOperator {

    private ExtensionContext extensionContext;
    private String clusterOperatorName;
    private String namespaceInstallTo;
    private String namespaceToWatch;
    private List<String> bindingsNamespaces;
    private long operationTimeout;
    private long reconciliationInterval;
    private List<EnvVar> extraEnvVars;
    private Map<String, String> extraLabels;
    private ClusterOperatorRBACType clusterOperatorRBACType;

    public SetupClusterOperator(SetupClusterOperatorBuilder builder) {
        this.extensionContext = builder.extensionContext;
        this.clusterOperatorName = builder.clusterOperatorName;
        this.namespaceInstallTo = builder.namespaceInstallTo;
        this.namespaceToWatch = builder.namespaceToWatch;
        this.bindingsNamespaces = builder.bindingsNamespaces;
        this.operationTimeout = builder.operationTimeout;
        this.reconciliationInterval = builder.reconciliationInterval;
        this.extraEnvVars = builder.extraEnvVars;
        this.extraLabels = builder.extraLabels;
        this.clusterOperatorRBACType = builder.clusterOperatorRBACType;

        // assign defaults is something is not specified
        if (this.clusterOperatorName == null || this.clusterOperatorName.
            isEmpty()) {
            this.clusterOperatorName = Constants.STRIMZI_DEPLOYMENT_NAME;
        }
        // if namespace is not set we install operator to 'infra-namespace'
        if (this.namespaceInstallTo == null || this.namespaceInstallTo.
            isEmpty()) {
            this.namespaceInstallTo = Constants.INFRA_NAMESPACE;
        }
        if (this.namespaceToWatch == null) {
            this.namespaceToWatch = this.namespaceInstallTo;
        }
        if (this.bindingsNamespaces == null) {
```

```
            this.bindingsNamespaces = new ArrayList<>();
            this.bindingsNamespaces.add(this.namespaceInstallTo);
        }
        if (this.operationTimeout == 0) {
            this.operationTimeout = Constants.CO_OPERATION_TIMEOUT_DEFAULT;
        }
        if (this.reconciliationInterval == 0) {
            this.reconciliationInterval = Constants.RECONCILIATION_INTERVAL;
        }
        if (this.extraEnvVars == null) {
            this.extraEnvVars = new ArrayList<>();
        }
        if (this.extraLabels == null) {
            this.extraLabels = new HashMap<>();
        }
        if (this.clusterOperatorRBACType == null) {
            this.clusterOperatorRBACType = ClusterOperatorRBACType.CLUSTER;
        }
        instanceHolder = this;
    }

    public static class SetupClusterOperatorBuilder {

        private ExtensionContext extensionContext;
        private String clusterOperatorName;
        private String namespaceInstallTo;
        private String namespaceToWatch;
        private List<String> bindingsNamespaces;
        private long operationTimeout;
        private long reconciliationInterval;
        private List<EnvVar> extraEnvVars;
        private Map<String, String> extraLabels;
        private ClusterOperatorRBACType clusterOperatorRBACType;

        public SetupClusterOperatorBuilder withExtensionContext(
            ExtensionContext extensionContext) {
            this.extensionContext = extensionContext;
            return self();
        }
        public SetupClusterOperatorBuilder withClusterOperatorName(
            String clusterOperatorName) {
            this.clusterOperatorName = clusterOperatorName;
            return self();
        }
        public SetupClusterOperatorBuilder withNamespace(
            String namespaceInstallTo) {
            this.namespaceInstallTo = namespaceInstallTo;
            return self();
```

```
}
public SetupClusterOperatorBuilder withWatchingNamespaces(
    String namespaceToWatch) {
    this.namespaceToWatch = namespaceToWatch;
    return self();
}

public SetupClusterOperatorBuilder withBindingsNamespaces(
    List<String> bindingsNamespaces) {
    this.bindingsNamespaces = bindingsNamespaces;
    return self();
}

public SetupClusterOperatorBuilder withOperationTimeout(
    long operationTimeout) {
    this.operationTimeout = operationTimeout;
    return self();
}
public SetupClusterOperatorBuilder withReconciliationInterval(
    long reconciliationInterval) {
    this.reconciliationInterval = reconciliationInterval;
    return self();
}

// currently supported only for Bundle installation
public SetupClusterOperatorBuilder withExtraEnvVars(
    List<EnvVar> envVars) {
    this.extraEnvVars = envVars;
    return self();
}

// currently supported only for Bundle installation
public SetupClusterOperatorBuilder withExtraLabels(
    Map<String, String> extraLabels) {
    this.extraLabels = extraLabels;
    return self();
}

// currently supported only for Bundle installation
public SetupClusterOperatorBuilder withClusterOperatorRBACType(
    ClusterOperatorRBACType clusterOperatorRBACType) {
    this.clusterOperatorRBACType = clusterOperatorRBACType;
    return self();
}

private SetupClusterOperatorBuilder self() {
    return this;
}
```

```
        public SetupClusterOperator createInstallation() {
            return new SetupClusterOperator(this);
        }
    }
}
```

Listing B.5: Cluster Operator builder pattern