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
KONFIGUROVATEĽNÁ PARALELNÁ EXEKÚCIA SYSTÉMOVÝCH TESTOV V RÁMCI PROJEKTU STRIMZI

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR   Bc. MAROŠ ORSÁK
AUTOR PRÁCE

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

BRNO 2022

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

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

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

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

Rozšířený abstrakt

Do tohoto odstavce bude zapsán rozšířený výtah (abstrakt) práce v českém (slovenském) jazyce.

Configurable parallel execution of system tests within the Strimzi project

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana X... Další informace mi poskytli... Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

<div align="right">

. . . .

Maroš Orsák

November 29, 2021

</div>

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant apod.).

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 at the beginning knew relatively nothing to do. 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, 19] is a container management system, and recently 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 the deployment of Apache Kafka [14, 23, 13, 2] on Kubernetes. Complexity, horizontal scalability and distribution system; are all attributes of a system called *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* product 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 great user interface. (https://grafana.com/)

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

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

Related work focuses on improving the overall verification time of a Strimzi product. Several releases of Strimzi gives 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
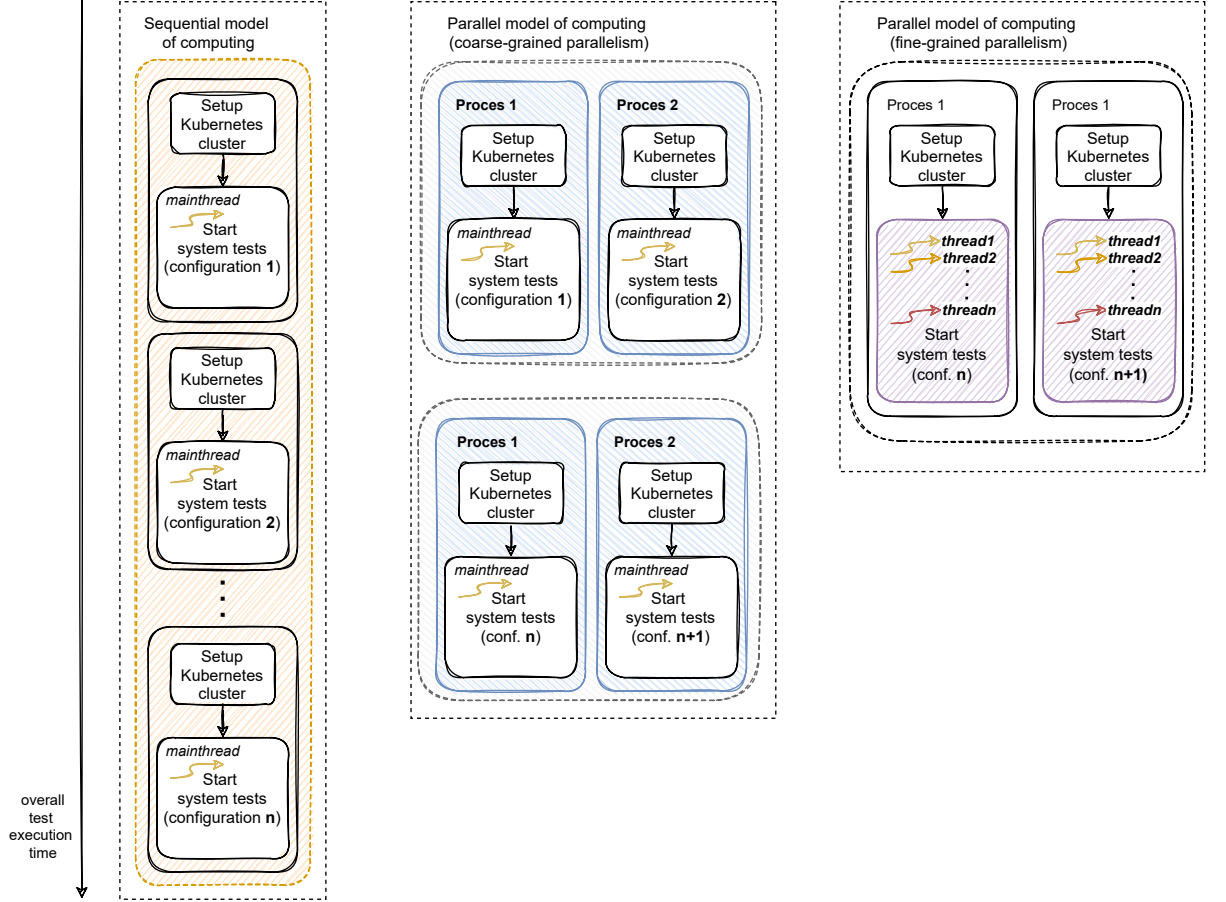
Figure 1.1: Evolution of our test framework execution

of this effort for coarse-grained parallelism in performing 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.

Acquisition of this work This work deals with the parallelisation of Kubernetes Strimzi system tests. The author contributed the given code to the open-sourced project

Strimzi available on Github[5], which also makes it possible to inspire other *kube-native*[6] products to implement such solutions. The comprehensive benefit of this work is the significant 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 of 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 deep analysis on the thesis implementation. Finally, we conclude the entire diploma thesis with the knowledge that has been acquired and at the same time with the possible future work in Chapter 7 and Chapter 8.

---

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

[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., Kubernets 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 [15] 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* [19]. 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* [17] and *How system tests work* [16].

2.1 Kubernetes

In 2014, Google came 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 leader and follow architecture. The leader node controls Kubernetes resources and the follow 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 originally 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, management of hardware, 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. **Virtualization** — 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, you can run many of these virtual machines on a single physical server, and your 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 a full operating system. At the same time, the management of these entities is not easy if we imagine production with hundred virtual machines. Another limitation is, that sometimes applications need to share information with each other and the strong isolation of VMs makes this difficult.

3. **Containerization** — In the last phase, containerization is considered as a lightweight alternative to virtualization. The difference between these two phases is that virtualization 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 virtualization, they have their filesystem, memory, and space. Containerization 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, containerization still has several shortcomings, such as managing more running containers simultaneously, making debugging challenging, etc.

4. **Container orchestration** — The phase of the present. Let's 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 analyze 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 virtualization era, which started in the early 70s. IBM Cambridge Scientific Center began the development of CP-40, the first operating system that implemented complete virtualization. However, what is very important to note is that the first known virtualization software was VMware, created in 1997. Afterwards, the lightweight era comes with an idea whose functionality was based on containerization [12]. Finally,

---

[5]Hypervisor - It is a software that manages virtual machines, for instance, VMware or VirtualBox.
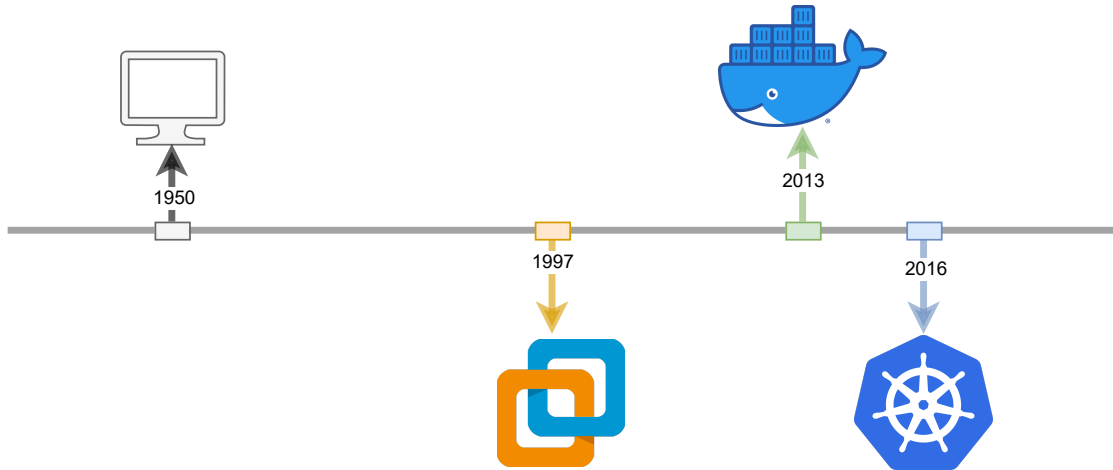
Figure 2.1: Evolution of virtual technologies

we have a manager who takes care of the overall management of the individual containers and guarantees their reliability, scales it 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

---

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

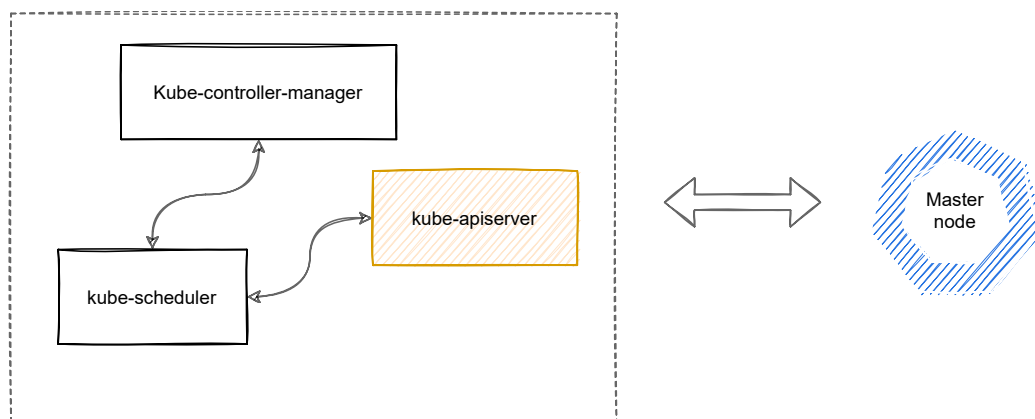biggest cloud providers such as Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes Service (EKS), and Google Kubernetes Service (GKS) have five master nodes, which are replicated in case of any failure. The master node 2.2 contains several components such as *kube-controller-manager*, *kube-scheduler* 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 2.1.2.

Algorithm 1 Generic algorithm for each Kubernetes controller

1: $desired\_state \leftarrow controller.obtain\_desired\_state()$
2: **while** $True$ **do**
3:     $desired\_state \leftarrow controller.obtain\_desired\_state()$
4:     $current\_state \leftarrow controller.observe\_current\_state()$
5:     **if** $current\_state \not\models desired\_state$ **then**
6:         $controller.reconcile()$
7:     $desired\_state \leftarrow current\_state$

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 suas *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.
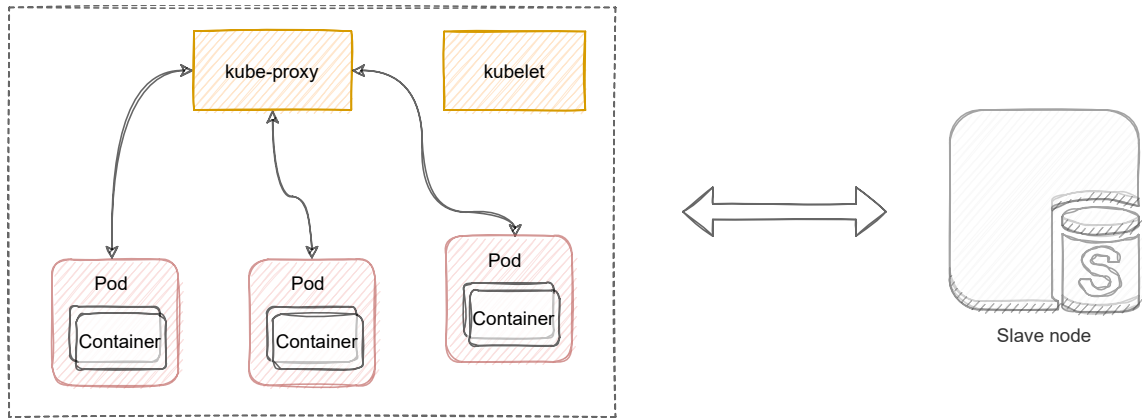
Figure 2.3: Representation of the Slave node

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 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
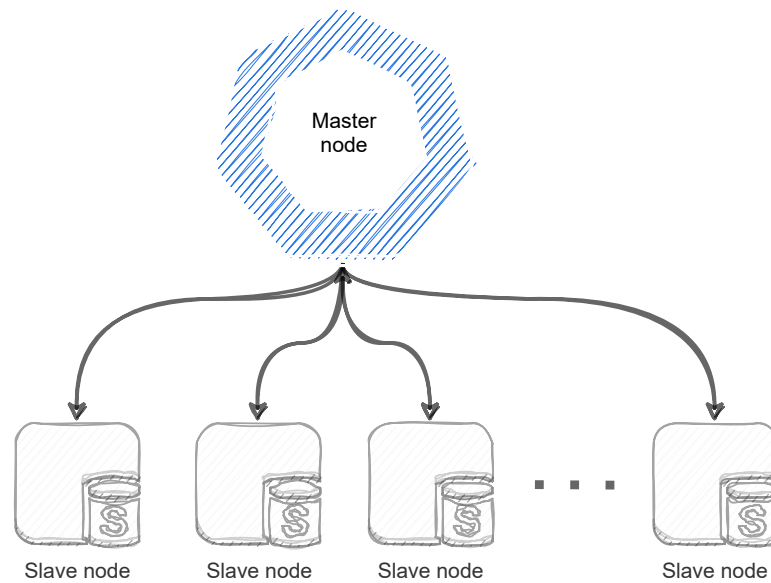
Figure 2.4: Relation between master and slave nodes

containers can run in it. These containers share storage, network, and specification 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 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, you want to expose your 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 you can find a real IP address. For example, if your demands are high and you want 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 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 belongs 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 your 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 the event of a heavy load. The same applies even if the given number of Pods is already high (then 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.

   Based on our knowledge, the skillful reader will realize 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 takes care of 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 are equivalent to the current state. Additionally, Deployment supplies the remaining properties, which are Rolling Update and Rollback. Since Deployment is a fully-fledged object in the Kubernetes API similar to Service, Pod, or Volume, that gives

---

[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
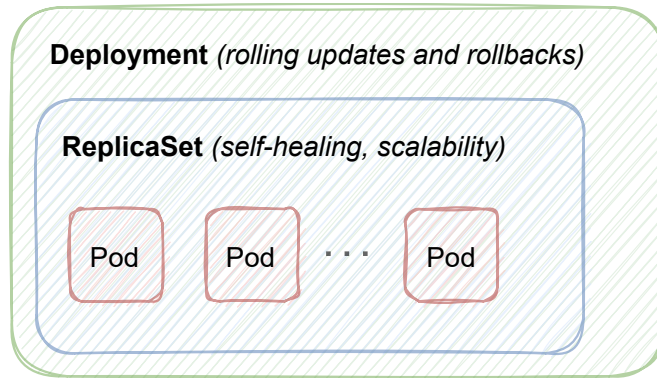
Figure 2.5: Hierarchy of Deployment, ReplicaSet and Pod inspired by The Kubernetes Book [19]

us the ability to define such an object in YAML files, such an object can then be edited and updated, 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 reconciliation loop described in 2.1.2, scaling operations, and 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 own 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 plays 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 the event of an unexpected failure.
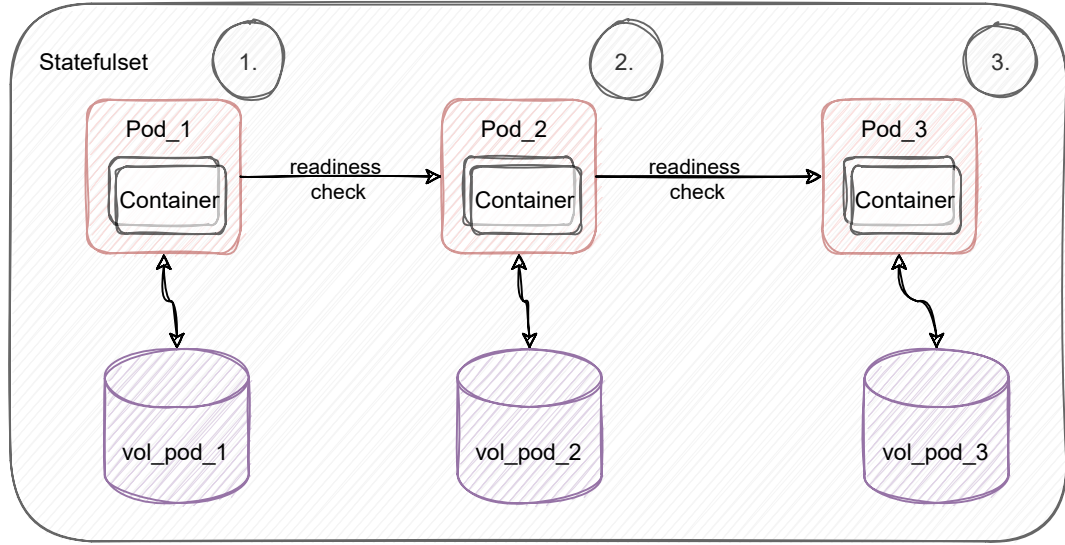
Figure 2.6: StatefulSet ordered creation of Pods

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* [23] and *Real-Time Data and StreamProcessing at Scale* [14]. Moreover, the Kafka streams subsection is based on *Mastering Kafka Streams and ksqlDB Building real-time data systems* [13]. We also used Kafka's documentation [2] as the most up-to-date reference. In these books and documentation, there can be found a more detailed explanation of Kafka itself.

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 that are 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 [15]

In the past, companies had applications or systems that share large amounts of data. Usually, these applications would provide valuable information to another application. So, there was one source system and one target system. But 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 onne of main reasons Kafka was invented. Lets 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. In that case, each source system

---

[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
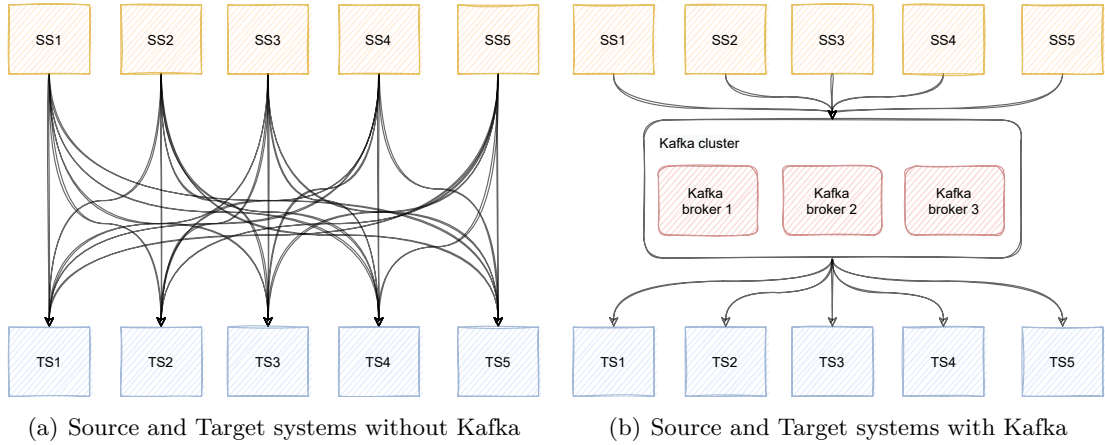
Figure 2.7: How to make system more efficient with Kafka

only has to bind to the Kafka broker, and all data are delivered by a single link. You can see the updated system in Figure 2.7b.

2.2.2 Fundamental concepts

In this subsection, we describe fundamental concepts of Apache Kafka such as Producer, Consumer, Kafka broker, Kafka cluster, and so on. The description is based on the *Real-Time Data and StreamProcessing at Scale* [14] 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 names such as Kafka server or Kafka node. 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 the 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 to 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 abstractions, which means that it is no longer necessary to create the producers themselves, but those entities are encapsulated in the client.

14

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. In the consumer configuration, it is necessary to specify the topic from which the consumer will read. 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 an single logical unit. Kafka does not support reading from one specific partition more than one consumer simultaneously. The reason why this concept was created is based on a straightforward question - *How are we able to consumes 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 describing why Kafka is so flexible in comparison with the traditional messaging based on AMQP protocol like ActiveMQ or RabbitMQ.
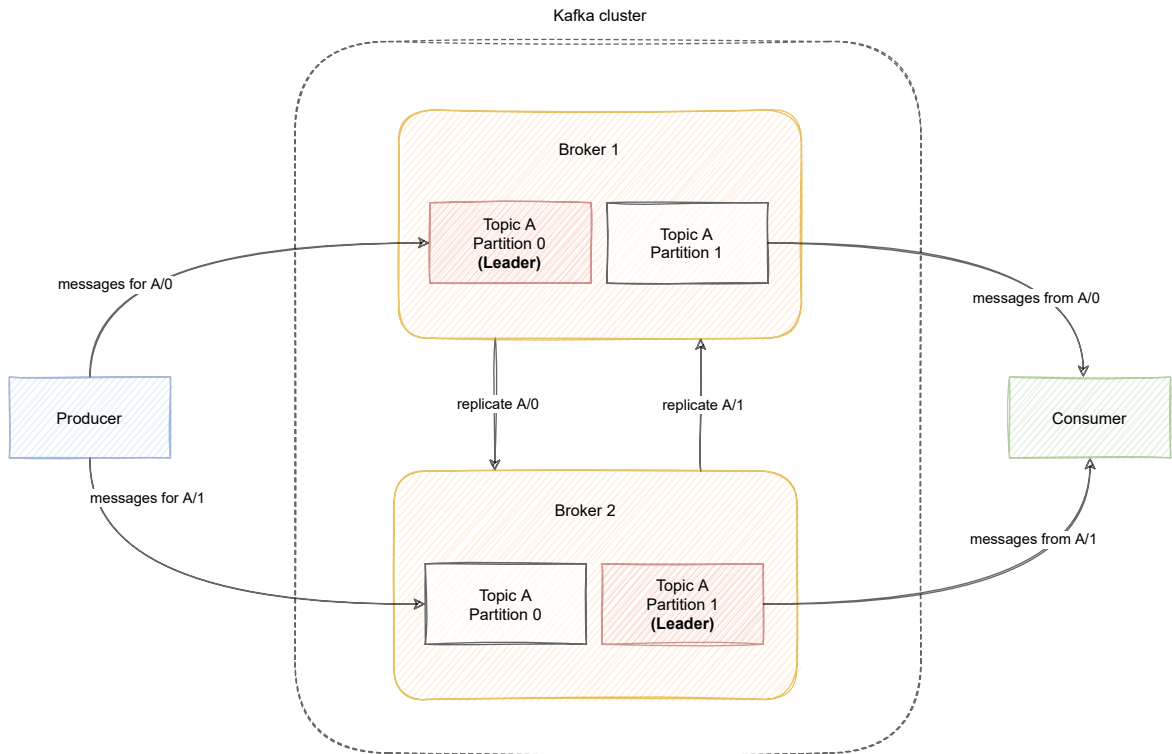
Figure 2.8: Kafka topic partition replication scenario in Kafka cluster inspired by *Real-Time Data and StreamProcessing at Scale* [14]

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. The replication factor is a number, which defines how many replicas will be available on the other brokers from the

---

[10]**Pooling** – periodic querying to the server in that case, to the Kafka broker
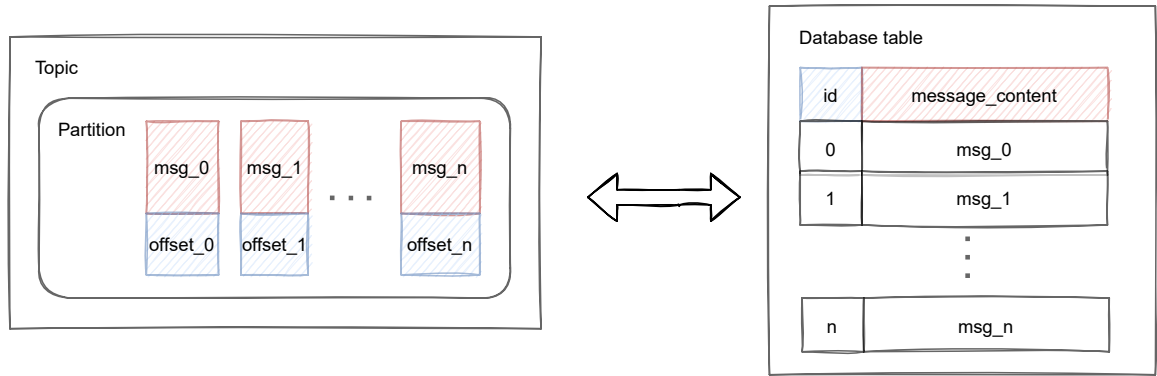
Figure 2.9: Equivalence of Kafka topic and database table

Kafka cluster. Imagine the following scenario – we have a Kafka cluster with three Kafka brokers. We create a new topic with a unique name by 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 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 your Topic 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 next 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 move on nineteen and so on. The Figure 2.10 illustrate this scenario. In general, we can understand offset as the message index.
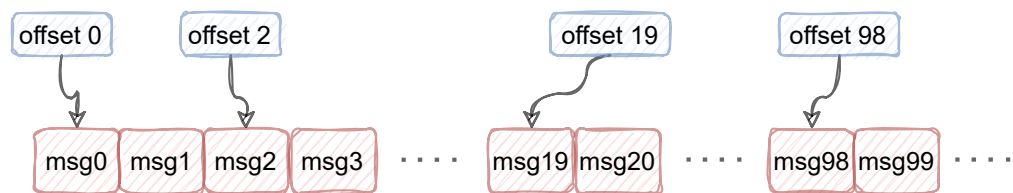
Figure 2.10: Partition offset

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,

---

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

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 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 Topic into several partitions. Practically speaking, you use the Kafka Streams API to deploy multiple instances of an application, each of which will handle a subset of the work. For illustration, one Topic 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.
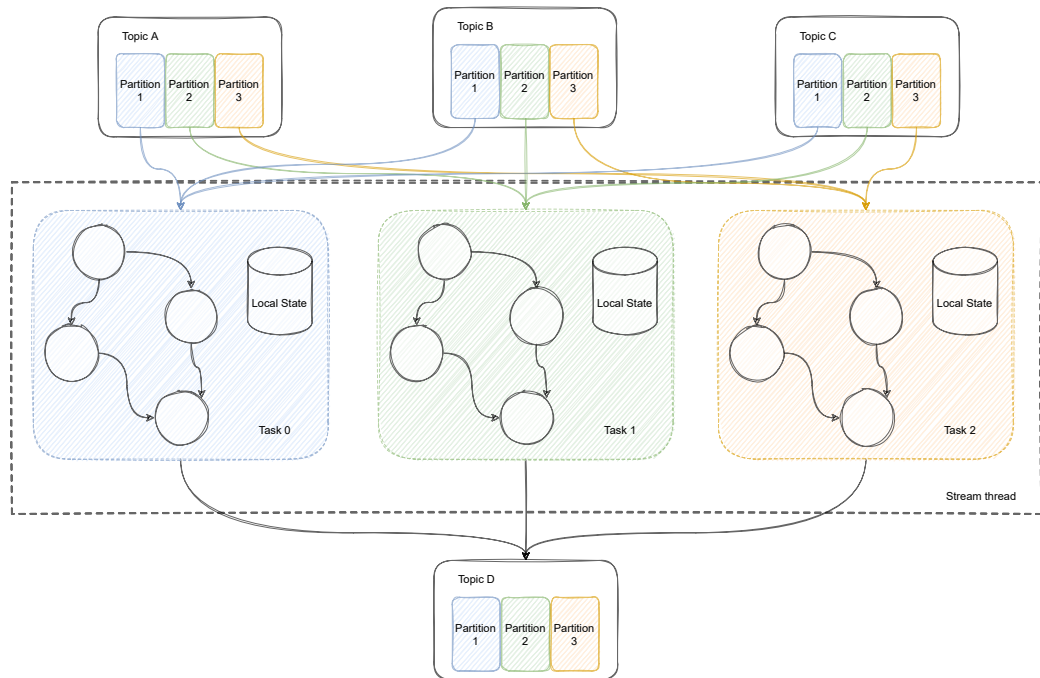
Figure 2.11: Kafka Streams with local state stores inspired by Kafka Documentation [2]

2. **Reliability** – If an error occurs on any node, Kafka automatically distributes the load to other nodes. However, we must realize 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

---

[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.
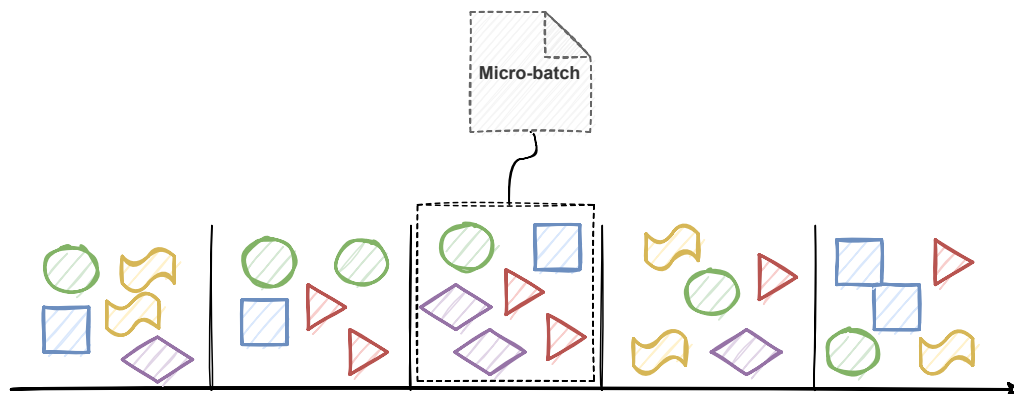
Figure 2.12: Micro-batching processing (typical for different systems) inspired by [13]

micro-batching, which occurs very much in machine learning where work is divided into several batches. These groups are then loaded into memory then 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 low latency and is considered true data streaming. Figure 2.13 illustrates the event-at-a-time processing strategy.
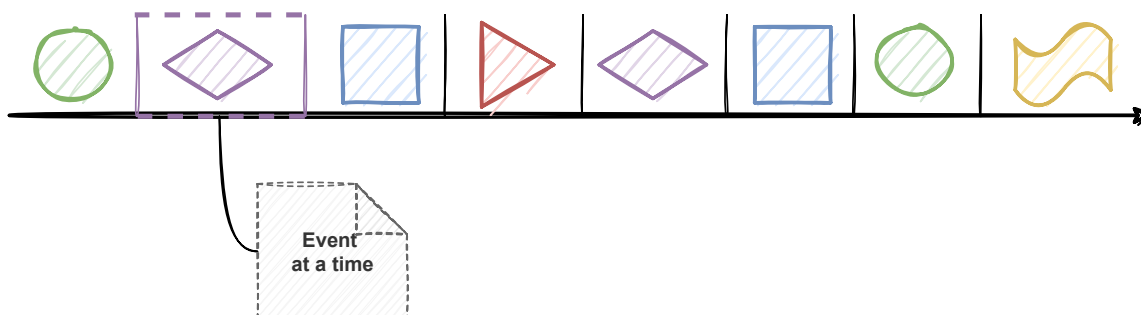
Figure 2.13: Kafka Streams uses event-at-a-time processing inspired by *Mastering Kafka Streams and ksqlDB Building real-time data systems* [13]

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 make 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 a large number of features that are transparent to the users. These include configuration, parallelization, 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, number of tasks executed in parallel, and the connector URL. The first such 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. In case 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, which is 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:

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 in any way, 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, 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 components such as Kafka Broker, Topic, Consumer, and Producer. The lack of libraries and writing 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
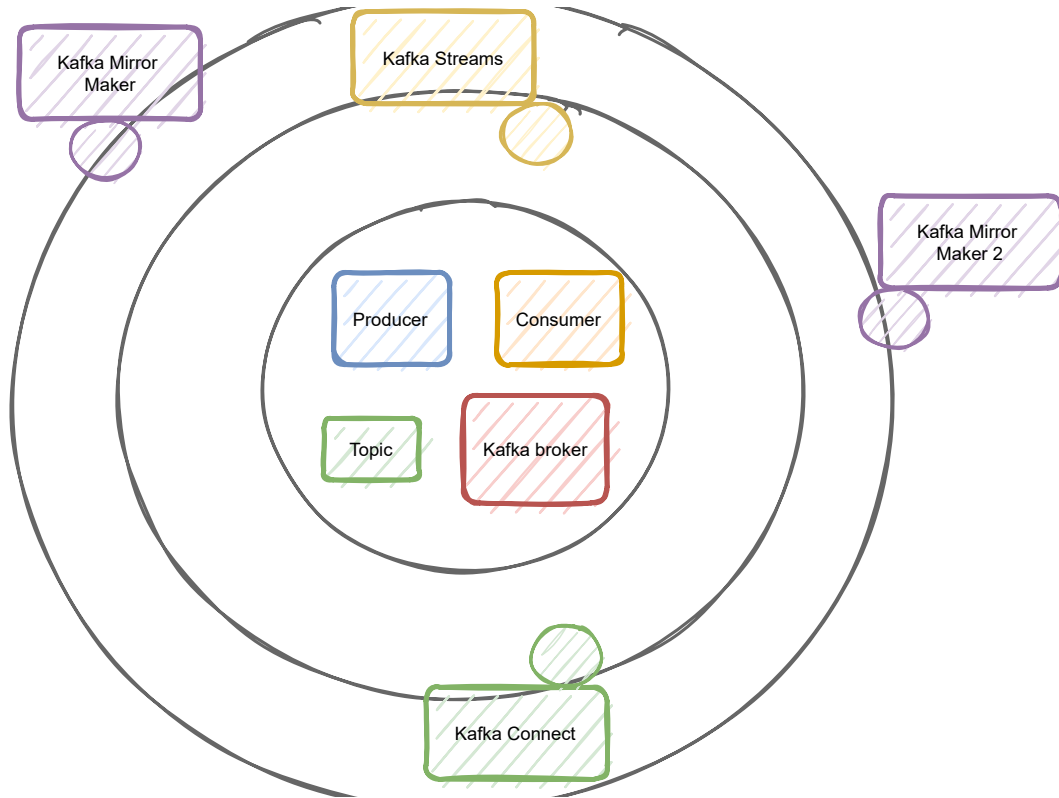
Figure 2.14: The entire Apache Kafka ecosystem.

starting with the Kafka Broker, Producer, Consumer, and Topic. There are 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, I 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. It is therefore 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 2.1.2 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 your 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 and others.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: kafkas.kafka.strimzi.io
  labels:
    app: strimzi
spec:
  group: kafka.strimzi.io
  names:
    kind: Kafka
   ...
  scope: Namespaced
   ...
  schema:
    openAPIV3Schema:
      type: object
      properties:
        spec:
          type: object
          properties:
            kafka:
              type: object
              properties:
                version:
                  type: string
                replicas:
                  type: integer
                  minimum: 1
                image:
                  type: string
                listeners:
                  type: array
                  minItems: 1
                   ...
```

```
apiVersion: kafka.strimzi.io/v1beta2
kind: Kafka
metadata:
  name: my-cluster
spec:
  kafka:
    version: 2.8.0
    replicas: 3
    listeners:
      - name: plain
        port: 9092
        type: internal
        tls: false
      - name: tls
        port: 9093
        type: internal
        tls: true
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      log.message.format.version: "2.8"
      inter.broker.protocol.version: "2.8"
    storage:
      type: ephemeral
  zookeeper:
    replicas: 3
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

(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 that 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** [22] – 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 main units, where the first unit is the Kafka architecture and 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 of which performs 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 communicates by default via the REST API, and the user can create, delete,
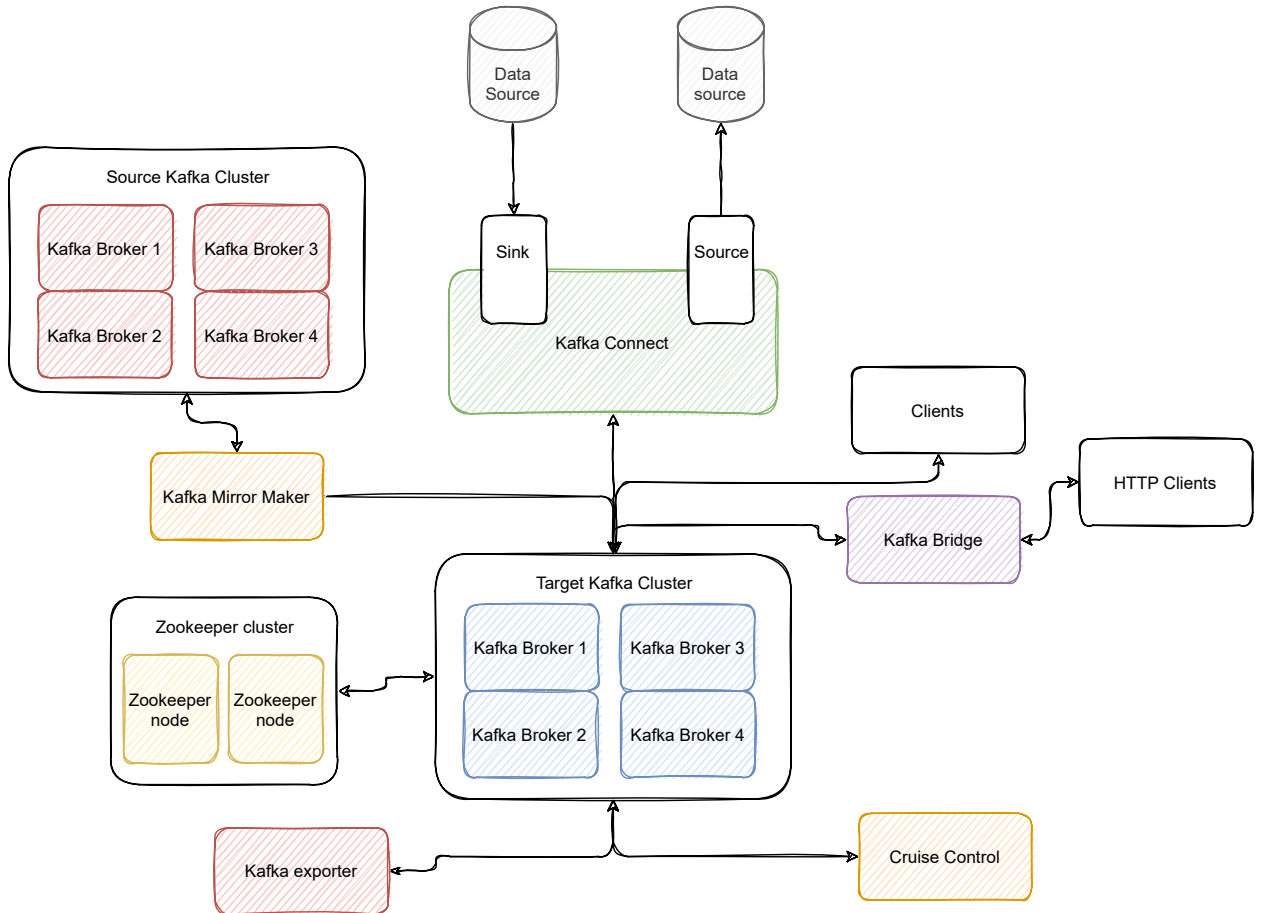

Figure 2.16: Strimzi Kafka architecture

---

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

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 and is used to extract additional metrics and supply them too Prometheus[15]. Then we have Kafka Connect and Kafka Mirror Maker, where we described the meaning of these components in the 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 other metrics. Cruise Control creates an workload model and analyzes when necessary to perform balancing and rearrange the load across the Kafka cluster. Everything we have described is show in Figure 2.16.

The second part in 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.
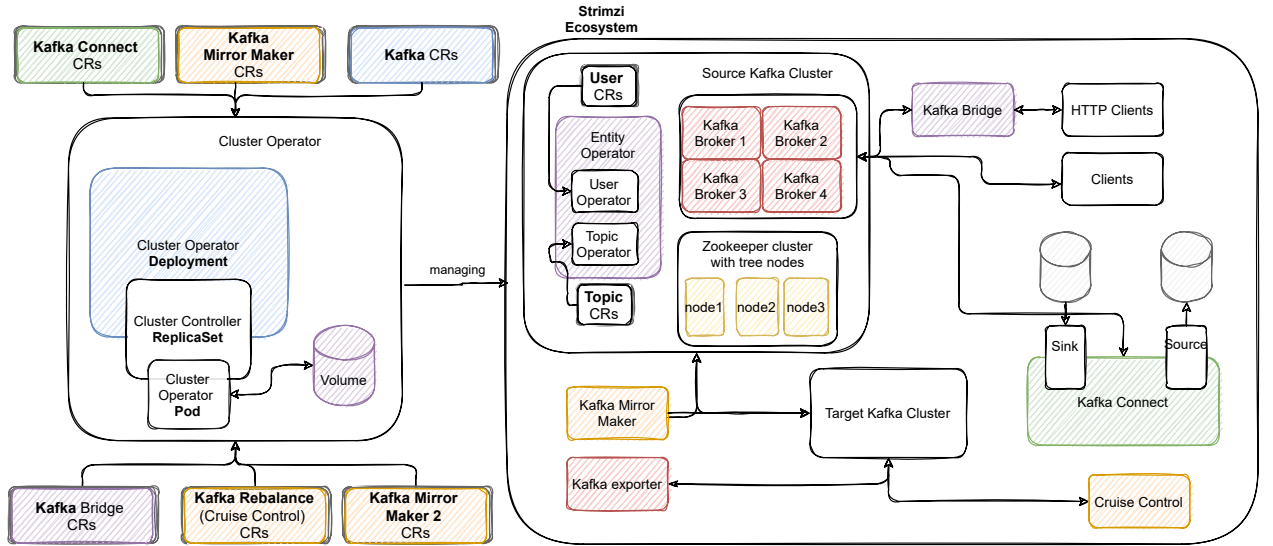
Figure 2.17: Strimzi Operators architecture with Strimzi ecosystem

The Topic Operator takes care of creating, deleting and updating individual Topics. It is also necessary to mention that the Topic Operator ensures synchronization 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 change topic property at the same time. The first action is considered as 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*

---

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

where the intersection is not empty. The second one is treated as incompatible change. It must deterministically select by some winner policy implemented inside Topic Operator.

The User Operator is responsible for the Kafka User Resource, which specifies authentication and authorization 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 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. In authorization, several components can also be used, such as ACLs (access control lists). For more complex rules, there is support for the Keycloak or Hydra authorization server. 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, 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 are executed. For instance, integration tests will not run if unit tests fail, similarly for integration and system tests. Furthermore, system tests run on multiple infrastructures such as *Openstack*, *Microsoft Azure* or *Amazon Web Services.* On each of these infrastructures, there are certain limitations for the set of tests. Since these are Kubernetes system tests, it is essential to realize 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 serves 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.

---

[16]**@BeforeAll** – is JUnit5 annotation, where one specify 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 waiting for a specific event. For instance, waiting for Rolling Update, if one change Kafka configuration
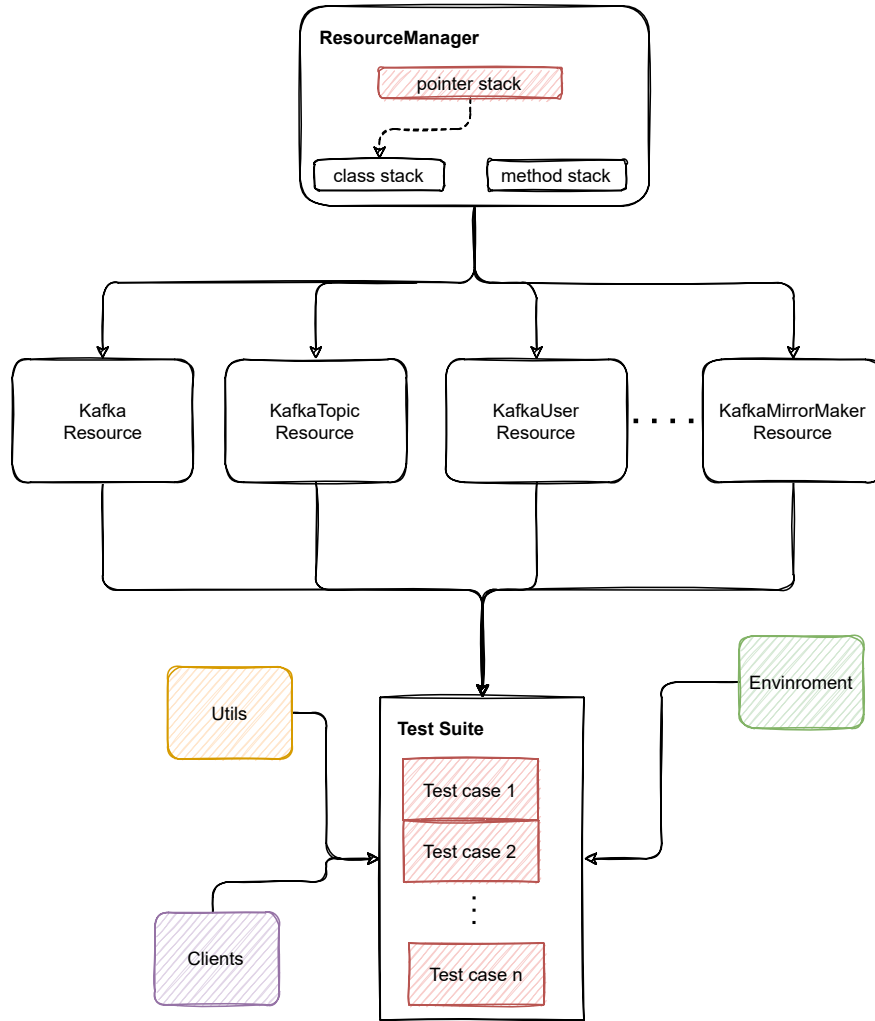
Figure 2.18: Strimzi system tests top-level component architecture

2.4.1   JUnit5 relation and execution of test cases

The entire implementation and management of the test lifecycle are handled by Junit5 Engine. 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 a scan of 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.

   (a) Iterate over all registered resolvers in the order they were registered in and find the first one that returns 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, it is necessary to add several mechanisms (i.e., creation of Kubernetes cluster, communication with Kubernetes cluster, management of Kubernetes resources, wait for conditions). 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 consisting of three master nodes and three worker nodes. Each of them 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 Java client with many number of methods that communicate directly via the Kubernetes REST

API. Most of the methods are designed to create, update, delete and retrieve a given re-
source. 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();*.

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;

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 and where the main/pointer stack points to 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 explicitly delete the resource, it is
possible via the following command *ResourceType.delete(name)*. Algorithm 2.4.2 defines
clean-up phase.

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 by 2.19.

```
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()
            .editEntityOperator()
                .editUserOperator()
                    .withNewInlineLogging()
                        .addToLoggers("rootLogger.level", "DEBUG")
                        .endInlineLogging()
                .endUserOperator()
                .editTopicOperator()
                    .withNewInlineLogging()
                        .addToLoggers("rootLogger.level", "DEBUG")
                    .endInlineLogging()
                .endTopicOperator()
            .endEntityOperator()
        .endSpec();
}
```

Figure 2.19: Default Kafka Custom Resource in KafkaResource.class

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

So if we summarize 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, which consists 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 2.20 we can execute n Test cases where
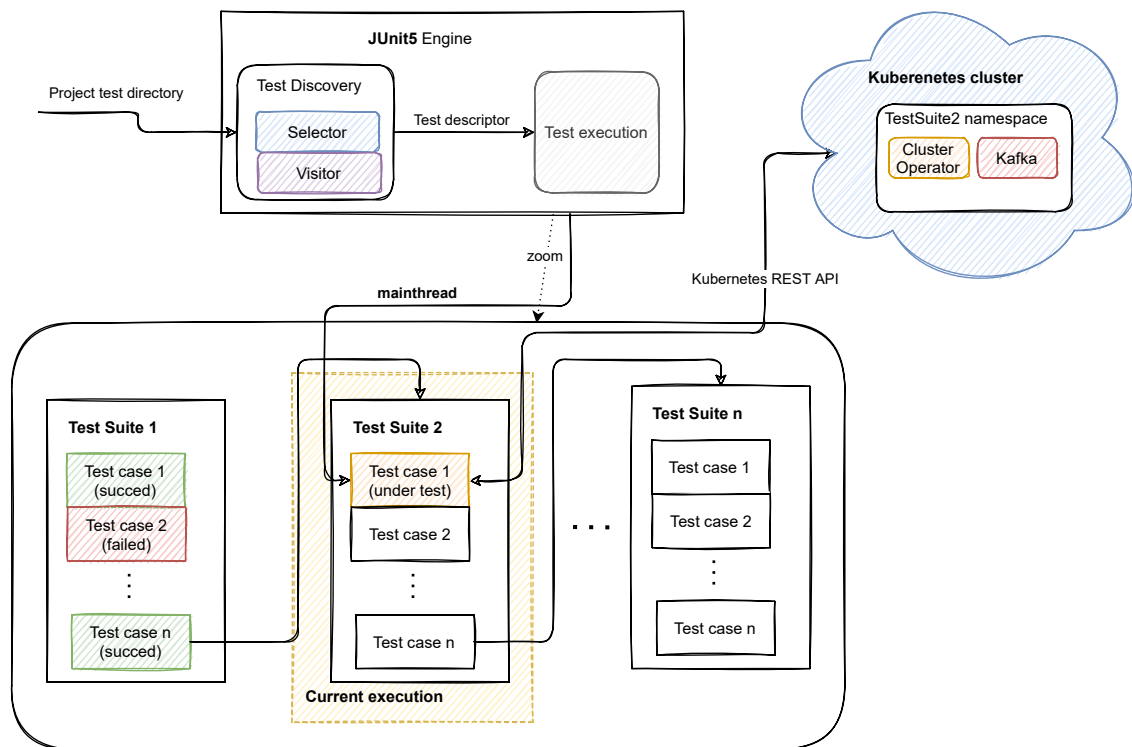
29

Figure 2.20: Strimzi system tests execution flow

Test Suite 2 is currently executed and specifically Test Case 1. The attentive reader will realize that the overall 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* [18] and *The Art of Multiprocessor Programming* [10].

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 useful 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 due to 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 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 [9], 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). Finally, we have to put the ratio between the sequential (single-processor) time and the parallel time, and we get 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*, etc., 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 end up in a deadlock[1]. The second is the liveness property, which tells us: „*An excellent thing will happen eventually*". For instance, the program always terminates. Thus, if we combine these two properties, then 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. In this type of memory, all processors access the same memory in the global address space.

---

[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. In the Uniform Memory Access architecture, each processor has its cache memory, 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). In general, 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. Update to shared memory (while the other processor reads a value that has not yet been updated and will therefore 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 its time on the processor and swap it with the different Process. When this happens, the Operating system stores the state of Process or thread and then load 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*. *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 aquire a lock whenever one wants to modify a thread or read from a shared resource. Then one modifies the source and finally release the lock. Acquiring a lock is an atomic operation performed as a single action and cannot be interrupted by other threads.

We know several implementations of lock, but not all of them guarantee us the liveness property. As a reminder, the liveness property tells us that: „*A particular 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 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 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 in 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 how 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. This is because 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 that 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 sometime in the near future. 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* [18] or *The Art of Multiprocessor Programming* [10].

Chapter 4

Proposal of parallel approach

In this chapter, the author describes the overall design for parallelism 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 extremely 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 whether it 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 we performed 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 parallelizable 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 3 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, with this first step, we just showed that it pays to parallelise.

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 shared 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, a pre-prepared templates and, at the same time, the whole mechanism of creation. If we
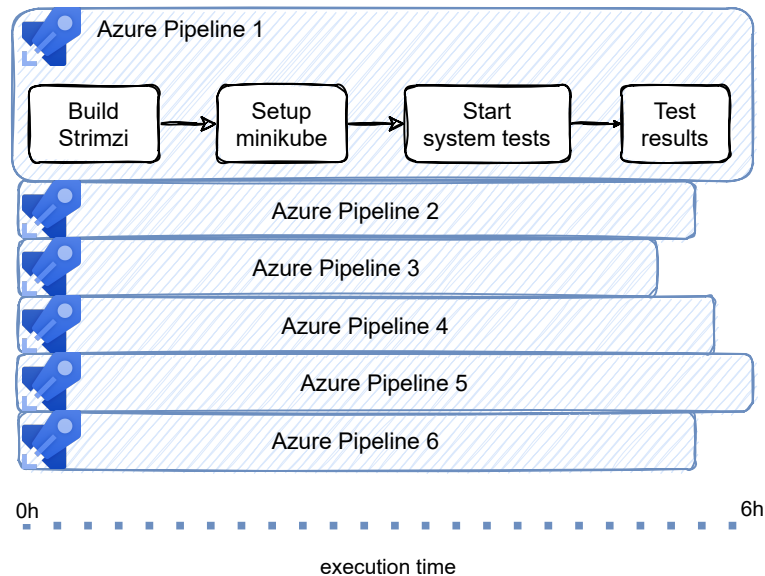
Figure 4.1: Azure pipelines in form parallelism used to execute our system tests

want 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 make a change. In the 2.4 section, we did not mention such a fact, but there is an attempt of 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 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 run six Azure Pipelines simultaneously. By this limitation, the total set of tests takes approximately 6 hours, which is still a significant ammount 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.
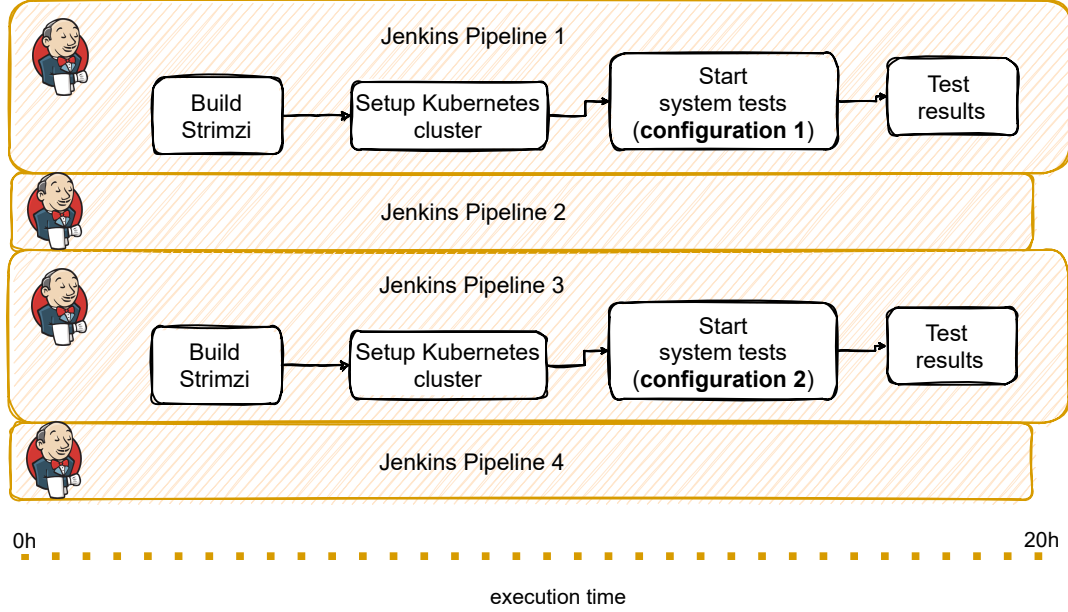
Figure 4.2: Jenkins pipelines in a form parallelism used to execute our system tests

It is also important to mention that we are limited by the number of processes (i.e., pipelines) that always use the separate Kubernetes cluster. On Amazon Web Services and Openstack infrastructures, we are not limited the computing resources we use. 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 difficult 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 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 own *TestEngine*, we have to implement our own *TestDiscovery* and thus create our own implementation similar to Algorithm 2.4.1. Furthermore, we need to create our own *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 on 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 own annotations. With this knowledge, even this approach does not meet our needs.

4.2.3 JUnit5 paralelization

The last alternative is the use of Junit5 *TestEngine* parallelisation. This, almost 3-year-old, feature of the Junit5 platform (released 3rd September in 2018) has 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 redundant creation of threads. The main advantage of such an approach is that it is not necessary to rewrite a complete perform of the tests. Moreover, it is unnecessary to implement TestDiscovery and

---

[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

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 parallelization 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** – The ThredPool object will have a predefined number of threads to work with and can be changed in the configuration using *…parallel.config.fixed.parallelism.*
- **Dynamic** – ThredPool object will have 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 the section 2.4, one will notice that the Resource classes contains two large pieces. The first are management methods (i.e., create(), delete()), and the second part are 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 are used for management would be left in these

---

[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.

classes. However, 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 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, before each test case, we switch to the method stack. 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 4.3.2, 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** $resource \in resources$ **do**
2:     $type \leftarrow findResourceType(resource)$
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** $resource \in resources$ **do**
10:          $type \leftarrow findResourceType(resource)$
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 asynchronously create ten resource instances independently of each other and then create a Barrier[7] because the following verification steps require all resources. Another thread-safe algorithm 4.3.2 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** $resource \in resources$ **do**
5:     **if** $resource == null$ **then**
6:         $continue;$
7:
8:     $type \leftarrow findResourceType(resource)$
9:     $\Phi \leftarrow getResourceWaitCondition(type)$
10:     $wait(resource, \Phi)$
11: **end for each**

Finally, we have the last part, that is deleting resources from the stacks. We propose a thread-safe algorithm 4.3.2, 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

---

[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

anything, 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: $\Psi \leftarrow mapResourceEmptinessCondition(context)$
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 paralelization

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 that could arise 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. Thanks to the Kubernetes client, which we already have in the system tests, this is possible. 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 in parallel and which need to be isolated. For parallel tests, we propose use the *@ParallelTest* 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, read-write lock consists of two types of locks. ReaderLock allows multiple readers to read from a shared source. However, if even one reader reads, no thread can write to the source. 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 rereleases it. So for the @IsolatedTest annotation, we propose using this type of lock to guarantee the system's safety property (mutual exclusion).
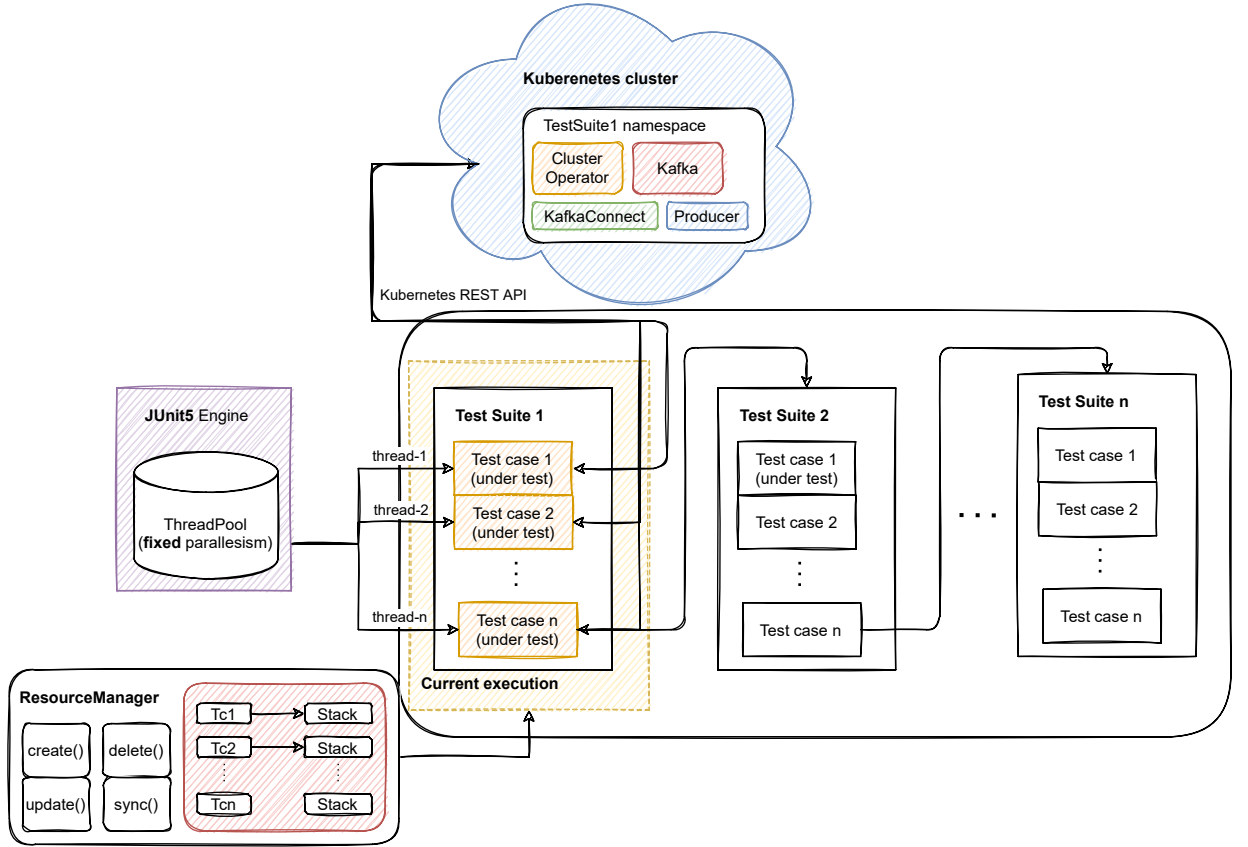
Figure 4.3: The best scenario in *method-wide* parallelism. *n* number of threads are executed, and there is no one *@IsolatedTest* in the test suite, which means that all test runs simultaneously. Note that *Tc* means Test case in short.

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 available, the quickier the results.

4.5   Class wide paralelization

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, 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. Finally, we follow up on this in Section 4.5.2, where we solve the given problem completely.

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 the question of 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

---

[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.

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.

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 4.5.1 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: // 1*st 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: // 2*nd 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* parallelization. In this case, it will be no different. We suggest using a read-write lock for the new *IsolatedSuite* annotation, which will encapsulate the lock. All the rules we described in the *method-wide* parallelisation will be the same as for *@IsolatedSuite*, so they will be the same semantically. The only difference will be the fact that @IsolatedTest will be an annotation applied only to method-scope, where *@IsolatedSuite* the annotation will be applied only to class-scope.

Additionally, we will have to design a mechanism for running multiple test suites in parallel simultaneously. An attentive reader, one will know the solution, and so we will solve it by creating an annotation that will contain an @Execution annotation with the value *CONCURRENT*, thus guaranteeing parallel execution. We suggest calling such an annotation *@ParallelSuite*.

Chapter 5

Implementation

5.1 Resource Manager paralell approach

5.2 Parallel annotations

5.3 Method wide paralelization

5.4 Class wide paralelization

5.5 Complications during implementation

Chapter 6

Experimental evaluation

6.1 Experimental setup

6.2 Results

6.3 Evaluation of the obtained results

Chapter 7

Future work

Chapter 8

Conclusion

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] ČERNOCKÝ, J. *English SOS* [online]. 2016 [cit. 2019-10-02]. Available at: https://merlin.fit.vutbr.cz/wiki/index.php/English_SOS.

[9] 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.

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

[11] HEROUT, A. *Herout.net – Poznámky učitele, kouče, čtenáře.* [online]. 2018 [cit. 2019-10-02]. Available at: http://www.herout.net/.

[12] 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.

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

[14] 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.

[15] 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/.

[16] 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/.

[17] 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/.

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

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

[20] Pyšný, R. *BiBTeX styl pro ČSN ISO 690 a ČSN ISO 690-2.* Brno, CZ, 2009. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Available at: https://www.fit.vut.cz/study/thesis/7848/.

[21] Rábová, Z., Hanáček, P., Peringer, P., Přikryl, P. and Křena, B. *Užitečné rady pro psaní práce* [online]. FIT VUT v Brně, november 2008 [cit. 2019-10-02]. Available at: https://www.fit.vut.cz/study/theses/theses-advice/.

[22] 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/.

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

Appendix A

How to use this template

This chapter describes individual parts of the template, followed by a brief instructions on how to use it. If you have any questions, comments etc, feel free to email them to `sablona@fit.vutbr.cz`.

Template parts description

Once you extract the template, you will find the following files and directories:

bib-styles Literature styles (see below).

obrazky-figures Directory for your images. Currently contains `placeholder.pdf` (a.k.a TODO image – see below) and image keep-calm.png to demonstrate inserting raster images (you don't submit these images with your thesis). It is advised to use shorter directory name, so that it is only in your chosen language.

template-fig Template images (BUT logo).

fitthesis.cls Template (design definition).

Makefile Makefile used to compile the project, count standard pages etc. (see below).

projekt-01-kapitoly-chapters-en.tex File for Your text (replace it's contents).

projekt-20-literatura-bibliography.bib Reference list (see below).

projekt-30-prilohy-appendices-en.tex File for your appendices (replace it's contents).

projekt.tex Main project file – definitions of formal parts.

The style of literature in the template is from Ing. Radek Pyšný [20], whose work was improved by prof. Adam Herout, dr. Jaroslav Dytrych and Mr. Karel Hanák to comply with the norm and support all frequently used types of citations. Its documentation can be found in the appendix

Aside from compilation to PDF, the Makefile also offers additional functions:

- rename files (see below),

- count standard pages,

- run a wave that adds unbreakable spaces,

- compress (zip) the result, ready to be sent to your supervisor and checked (make sure that all the files you've added are included, if not, add them manually).

Keep in mind that the wave is not perfect. You always need to check whether or not there is something inappropriate at the end of a line manually – see Online language handbook[1].

Similar rules apply also in English - see eg. article Run Ragged[2], according to which there should be no prepositions, dash or short words (2–3 letters) at the end of the lines, the two lines following each other should not end with a comma and line break should not be also in the phrases from 2-3 words.

Pay attention to page numbering! If the table of contents is 2 pages long and the second page contains only "Enclosures" and "List of enclosures" (but there is no enclosure), the page numbering is changed by 1 (table of contents and contents "mismatch"). The same thing happens if the second or third page contains only "References" and there's a chance that this can occur in other situations too. There are multiple solutions to this (from editing the table of contents, setting the page counter all the way to more sophisticated methods). **Check the page numbering before you submit your thesis!**

Recommendations for working with the template

1. **Make sure you have the latest version of template.** If you have a template from last year, there should be a newer version (updated information, fixed errors etc.) available at the faculty or study advisor web pages.

2. **Choose a language**, that you want to use for your technical report (czech, slovak or english) and consult your supervisor about your choice (unless it was agreed upon in advance). If your language of choice is not czech, set the respective template parameter in file projekt.tex (e.g.: `documentclass[english]{fitthesis}` and translate the declaration and acknowledgement to english or slovak).

3. **Rename the files.** When you extract the files, there should be a file named projekt.tex. If you compile it, it will create a PDF with technical report named projekt.pdf. If multiple students send their supervisor projekt.pdf to have it checked, they have to rename them. For that reason, it is advised to rename the file so that it contains your login and (if needed, abbreviated) work topic. Avoid using spaces, diacritic and special symbols. An appropriate name for your file can look like this: "xlogin00-Cleaning-and-extraction-of-text.tex". You can use the included Makefile to rename it:

   ```
   make rename NAME=xlogin00-Cleaning-and-extraction-of-text
   ```

4. Fill in the required information in file, that was originally named projekt.text, that means type, year (of submission), thesis title, author's name, department (according to specification), supervisor's titles and name, abstract, keywords and other formal requirements.

---

[1]Internetová jazyková příručka http://prirucka.ujc.cas.cz/?id=880
[2]Run Ragged https://24ways.org/2013/run-ragged/

5. Replace the contents of thesis chapters, references and enclosures files with the contents of your technical report. Individual enclosures or thesis chapters can be saved to separate files – if you choose this approach, it is advised to comply with the file naming convention, and the number will be followed by the chapter title.

6. If you don't need enclosures, comment the respective part in projekt.tex and erase everything from the corresponding file or delete it. Don't try to come up with an aimless enclosures just to have something in that file. An appropriate enclosure can be the contents of included memory medium.

7. Delete the chapter and attachment files for a language you haven't used (with or without `-en`).

8. Assignment that you download in PDF from FIT IS (link "Thesis assignment") save to file `zadani.pdf` and enable its insertion into work by appropriate template parameter (`documentclass[zadani]{fitthesis}`) in `projekt.tex`.

9. If you don't want to print references in color (i cannot recommend this without consulting your supervisor), you'll need to create a second PDF for printing and set the template printing parameter:
   (`documentclass[english,zadani,print]{fitthesis}`). Colored logo must not be printed in black and white.

10. The binder templace where the thesis will be typeset can be generated in faculty IS at specification. Can be enabled for dissertation using the `cover` parameter in template.

11. Don't forget that source files and (both versions) PDF has to be on a CD or other medium included in the technical report.

Instructions for double-sided printing

- **It is advised to consult your supervisor about double-sided printing.**

- If you used double-sided printing for your thesis and it's thickness is smaller than the thickness of the binder, it doesn't look too good.

- Enabled using the following template parameter:
  `\documentclass[twoside]{fitthesis}`

- After printing a double-sided sheet, make sure that the canon of page construction is in the same position on both pages. Inferior printers with duplex printing unit usually cause a shift by 1–3 mm. This can be solved with some printers. Print the odd pages first, put them back into the same tray and print the even pages.

- Leave a blank page after title page, table of contents, references, list of tables, list of appendices and other lists to make sure that the following part starts on an odd page (`\cleardoublepage`).

- Check the final result thoroughly.

Paragraph style

Paragraphs have justified alignment and there are multiple methods for formatting them. In Czech paper literature, a paragraph indentation method is common, where each paragraph of the text have the first line of a paragraph indented by about one to two quads, that is, about two widths of the capital letter M of the base text (always about the same preselected value). In this case, the last line of the previous paragraph and the first line of the following paragraph are not separated by a vertical space. The interleaving between these lines is the same as the interleaving inside the paragraph [21].

Another method is indenting paragraphs, which is common for electronic typesetting and for English texts. In this method, the first line of a paragraph is not indented and a vertical space of approximately half of a line is inserted between the paragraphs. Both methods can be used in the thesis, however, the latter method is often more suitable. Methods should not be combined.

One of the above methods is set as the default in the template, the other can be selected by the template parameter "`odsaz`".

Useful tools

The following list is not a list of all useful tools. If you have experience with a certain tool, feel free to use it. However, if you don't know which tool to choose, consider the ones listed below:

MikTeX LaTeX for Windows – a distribution with simple installation and great automated package downloading. MikTeX even has it's own editor, but I highly recommend TeXstudio.

TeXstudio Portable opensource GUI for LaTeX. Ctrl+click switches between source text and PDF. Integrated spell checker[3], syntax highlighter etc. To use this tool, you need to first install MikTeX or another LaTeX distribution.

WinEdt A good combination for Windows is WinEdt + MiKTeX. WinEdt is a GUI for Windows, and if you want to use it, you need to first install MikTeX or TeX Live.

Kile Editor for KDE (Linux) desktop environment. Real-time preview. To use this tool, you need to have TeX Live and Okular installed.

JabRef Neat and simple Java program for bibliography (references) file management. No need to learn anything – provides a simple window and a form for entry editing.

InkScape Portable opensource vector graphic (SVG and PDF) editor. Excellent tool to use to create images for technical text. Difficult to master, but the results are worth it.

GIT Great tool for teamwork when it comes to projects, but can be incredibly useful even to a single author. Simple version control system, backup options and transfer between multiple computers.

---

[3]Spell checker for czech version can be installed from https://extensions.openoffice.org/de/project/czech-dictionary-pack-ceske-slovniky-cs-cz

Overleaf Online LaTeX tool. A real-time compilation of source text that allows for simple collaboration (supervisor can continuously keep an eye on the progress made), move to a place in source file just by clicking in the PDF preview, spell checker etc. There are some limitations to what you can do if you want to use it for free (some people are comfortable with it for dissertation, others can run into it while they write a bachelor's thesis) and it is rather slow for long texts. FIT BUT has for students and employees of a license, which can be activated on `https://www.overleaf.com/edu/but`.

Note: Overleaf does not use template Makefile – to get compilation to work, you need to go to the menu and select `projekt.tex` as s Main document.

Appendix B

Writing english texts

This chapter is taken from web pages of Jan Černocký [8].

A lot of people write their technical reports in english (which is good!), but they make a lot of unnecesary mistakes (which is bad!). I'm not an english export myself, but I've been using this language for a while now to write, read and even communicate – this chapter contains a handful of important things. If you want to be certain that your thesis or article is 100 % correct, your best bet is to hire a native speaker (preferably someone who is technically capable and understands what you write about . . .).

In general

- Before you jump into it head first, I suggest you read a handful of technical articles written in english and try to remember or preferably understand how you should approach writing one yourself.

- Always use a spell checking tools – built in tools in Word, or in OpenOffice. If you work on Linux, I suggest you use ISPELL. Some spell checking (I think it's the one in PSPad) are not very good and ignore a lot of mistakes.

- Use grammer checking tools. I'm not entirely sure if there is one available for Linux, but the one in Word is fairly decent and if it underlines anything with green color, it's probably wrong. You can even copy and paste Latex source code here, fix any and all grammar errors and save it as a clean text again. If you use vim, there's a built in grammar checking tool too, and it's capable of detecting typos and errors in basic grammar. Write this in the first line of your thesis tex file:

  ```
  % vim:spelllang=en_us:spell
  ```

  (alternatively `en_gb` for OED english) *Editor's note:* There is a very good online tool Grammarly[1], with free basic version.

- Online dictionaries are good, but don't rely on them in every situation. Usually you get multiple choices and not all of them are correct for the given context.

---

[1] https://www.grammarly.com/

- You can probably figure out what the correct option is by looking each option up and seeing the context in which they're used, example given: "advantage/privilege/facility of approach". Online dictionaries give you a handful of results. Look them up one by one using google search:

```
"advantage of this approach" 1100000 hits
"privilege of this approach" 6 hits
"facility of this approach"  16 hits
```

I'm not saying it's 100 % correct, but at least you have something to go on. This can be used to find the correct connectives (e.g. "among two cases" or "between two cases"?)

SVOMPT and concord

The structure of an english sentence is SVOPMT: SUBJECT VERB OBJECT MANNER PLACE TIME and there's no other way around it. It is not a flexible structure. There are possibly exceptions in things like a theater play, where something needs to be emphasized. Subject must be present in every single single sentence, people tend to forget as some languages have a sentence structure where the subject can be implicit and not mentioned. SVOMPT applies to dependent clauses too!

```
BAD: We have shown that is faster than the other function.
GOOD: We have shown that it is faster than the other function.
```

Concord or grammatical agreement between two words in a sentence – it sounds silly, but people make countless mistakes here.

```
he has
the users have
people were
```

Articles

Articles in english are a nightmare and almost all of us fail to use them correctly. The basic rule is, that if there's a particular noun, it's preceeded by "the". Definite articles must be in following phrases:

```
the first, the second, ...
the last
the most (superlatives and adverbs) ...
the whole
the following
the figure, the table.
the left, the right - on the left pannel, from the left to the right ...
```

On the contrary, there can't be an article when you're referring to a specific figure, chapter, etc.

```
in Figure 3.2
in Chapter 7
in Table 6.4
```

The use of "a" and "an" is based on the pronounciation, rather than how the word is written:

```
an HMM
an XML
a universal model
a user
```

Verbs

Passive voice can be tricky – regular verbs are usually not a problem, irregular verbs however are a common source of errors, typically

```
packet was sent (rather than send)
approach was chosen (rather than choosed)
```

. . . most of the time, the spell checker will correct it, but it's not guaranteed.

Tenses are a mess at times. If something just is in general, use present tense. If you did something, use past tense. If you got results that already exist and you just discuss them, use present tense. Try to avoid complicated tenses such as present perfect or worse past perfect if you're not 100 % sure.

```
JFA is a technique that works for everyone in speaker recognition.
We implemented it according to Kenny's recipe in \cite{Kenny}.
12000 segments from NIST SRE 2006 were processed. When compared
with a GMM baseline, the results are completely bad.
```

Sentence length and structure

- Try to write shorter sentences. If you sentence is 5 lines long, it's probably a pain to read, if it can even be done.

- Comma is a powerful tool and you should use it for your sentence structure. Use a comma to seperate the initial dependent clause from the main independent clause. Sometimes it is appropriate to put a comma just before "and" (unlike other languages)!

```
In this chapter, we will investigate into ...
The first technique did not work, the second did not work as well,
and the third one also did not work.
```

The specifics of a technical text

When writing a technical text, don't use common phrases such as

```
he's
gonna
Petr's working on ...
```

and others. The only tolerated thing is "doesn't", but you can never go wrong with "does not".

Technical texts utilize passive voice a lot more than active voice:

```
BAD: In this chapter, I describe used programming languages.
GOOD: In this chapter, used programming languages are described.
```

If you want to use active voice, it's more common to use "we", even though you work alone. "I", "my", etc. are only used when you need to emphasize that you are the person of utmost importance, for example in the conclusion or when discussing "original claims" in disertation.

Common erros in words

- Pay attention to his/hers, it's not "it's" but "its"

- Image is not picture, it's figure.

- The connective is "than", not "then" – bigger than this, smaller than this ... very common error! "Then" is used in the context of time.

Appendix C

Checklist

This checklist was taken from a template for academic work, that is available on Adam Herout's blog [11], based on the ideas of Igor Szöke[1], with their permission.

A big part of the safety of air transport are checklists. They have checklists for basically anything and everything, even the most cut-and-dry procedures. If a pilot can get over the tedious process of marking off every single checkbox of a procedure, you can as well. Make a checklist of your own before you submit your thesis. **Yes, really:** print it, grab a pencil and check every single item on the list. It will make your life easier — avoid unnecessary errors that can be fixed within a couple minutes — as well as others', at very least your supervisor and reviewer of your thesis.

Structure

☐ You can tell that the assignment was completed just by looking at the chapter titles as well as their structures.

☐ There is no chapter with less than four pages (except for introduction and conclusion). And if so, I discussed this with my supervisor and they gave me a green light.

Figures and charts

☐ Every single image and table was checked and their position is close to the text that references them. In other words, they're easy to find.

☐ Every single image and table has a good enough caption, to ensure that the figure makes sense on it's own, without the necessity to read the text. (There's no harm in a long caption.)

☐ If an image is taken from somewhere, it is mentioned in the caption: "Taken from [X]."

☐ Texts in all images have a font size similar to the surrounding text (neither significantly larger, nor significantly smaller).

☐ Charts and schemes are vector graphics (eg. in PDF).

☐ Screenshots don't use lossy compression (they're in PNG).

---

[1] http://blog.igor.szoke.cz/2017/04/predstartovni-priprava-letu-neni.html

☐ All images are referenced in the text.

☐ Axes in charts have their captions (name of the axis, units of measurement, values) and a grind if need be.

Equations

☐ Identifiers and their indexes in equations are single letters (except for rather uncommon cases like t_{max}).

☐ Equations are numbered.

☐ All the variables and functions that haven't been explained yet are explained below (or rarely above) the equation.

Citations

☐ **All used sources are cited.**

☐ URL adresses referencing services, projects, sources, github, etc. are referenced using `\footnote{\url{...}}`.

☐ URL adresses in citations are only present, if necessary – article is cited like an article (author, title, where and when was it published), not using URL.

☐ Citations have author, title, publisher (conference title), year of publishing. If a citation does not have either of these, there is a good explanation for this special case and my supervisor agreed.

☐ If there is anything taken over from some other work in the program source code, it is properly cited therein in conformance with the license.

☐ If an essential part of the source code of the program is taken over, this is mentioned in the text of the thesis and the source is cited.

Typography

☐ No line extends past the right margin.

☐ There is no single-letter preposition at the end of a line (fixed using unbreakable space ~).

☐ Number of image, table, equation, citation is never a first item of a new line (fixed using unbreakable space ~).

☐ There is no space before a numeric reference to a footnote (like this[2], not like this [3]).

---

[2]footnote example
[3]another footnote example

Language

☐ I used spellchecker and there were no typos in the text.

☐ I had someone else read my thesis (at least one person), that knows czech / slovak / english well.

☐ Someone who knows english well checked the abstract in a czech or slovak written abstract thesis.

☐ No part of the text is written in second person (you).

☐ If first person is used (i, we), a subjective matter is being described (i decided, i designed, i focused on, i found out, etc.).

☐ There are no colloquialisms in the text.

☐ There are no *default* words in the text.

Result is on a data medium, i.e. software

☐ I have a non-rewritable data medium ready.

- CD-R,
- DVD-R,
- DVD+R in ISO9660 format (with RockRidge and/or Jolliet extension) or UDF,
- SD (Secure Digital) card in FAT32 or exFAT format, the card is set to write-protected mode

☐ If the result is online (service, application, . . .), URL is visible in introduction and conclusion.

☐ The medium contains the following mandatory items:

- source codes (e.g. Matlab, C/C++, Python, . . .)
- libraries necessary for compilation,
- compiled solution,
- PDF containing a technical report,
- text source code (LaTeX),

and the following optional items after consulting your supervisor:

- relevant (e.g. testing) data,
- demo video,
- poster in PDF
- . . .

☐ Source codes are refactorized, commented and labelled with an authorship header so that others can tell what they actually are.

☐ Any and all snippets of code taken from another sources are properly cited – differentiated using a opening and in case of multiple lines of code a closing comment. Comments contain everything that the license on web (always try to find out what the license is – for example, Stack Overflow[4] has a very strict citation policy).

Submission

☐ Do I want to delay (by at most 3 years) the publication ? If so, I will submit an application (in IS) at least a month prior to the submission of the academic work, and I'll include attitude of the company that the intellectual property belongs to and needs to be protected.

☐ I have at least minimum number of standard pages (can be calculated using Makefile and by adding number of pages that images translate to). If I'm just under the minimum, I consulted my supervisor about it.

☐ If I want a two-sided print, I consulted my supervisor about it and I've used correct template settings for two-sided printing. Chapters begin on odd pages.

☐ Technical report is bound in a bookbindery (at least one print, both prints if I'm delaying the publishing).

☐ Title page is followed by the specification (in other words, downloaded from IS and inserted into the template)

☐ Abstract and keywords are uploaded in IS.

  • There are no ~ characters for non-breaking spaces in the abstract and keywords in IS.

☐ PDF of thesis (with clickable links) is in IS.

☐ Both prints are signed.

☐ One (both if I'm delaying the publishing) of the prints contains a data medium with my login written on it using a CD marker (CD marker can be borrowed in library, at Student affairs or when I'm submitting the work).

---

[4]https://stackoverflow.blog/2009/06/25/attribution-required/

Appendix D

LaTeX for beginners

This chapter contains commonly used LaTeX packages and commands, that you might need when you're developing a thesis.

Useful packages

Students usually encounter the same issues. Some of them can be solved using the following LaTeX packages:

- `amsmath` – additional equation typesetting options,

- `float, afterpage, placeins` – image placement,

- `fancyvrb, alltt` – change the properties of Verbatim environment,

- `makecell` – additional table options,

- `pdflscape, rotating` – rotate a page by 90 degress (for image or table),

- `hyphenat` – change how words break,

- `picture, epic, eepic` – direct image drawing.

Some packages are used in this very template (in the lower section of fitthesis.cls file). It is also advised to read the documentation for individual packages.

A table column aligned to left with a fixed width is defined as „L" in the template (used as „p").

To reference a place within text, use command `\ref{label}`. Depending on the placement of this label, it will be a number of chapter, subchapter, image, table or a similar numbered element. If you want to reference a specific page, use command `\pageref{label}`. To cite a literature reference, use command `\cite{identifier}`. To reference an equation, you can use command `\eqref{label}`.

Symbol – (dash) is used generated using two minus signs (like this: `--`) in LaTeX.

Commonly used LaTeX commands

I highly recommend you check the source text of this chapter and see how the following examples are created. The source text even contains helpful comments.

Example table:

Table D.1: Assessment table

Name		
Name	Surname	Assessment
Jan	Novák	7.5
Petr	Novák	2

Example equation:

$$\cos^3 \theta = \frac{1}{4} \cos \theta + \frac{3}{4} \cos 3\theta \qquad \text{(D.1)}$$

and two horizontally aligned equations:

$$3x = 6y + 12 \qquad \text{(D.2)}$$
$$x = 2y + 4 \qquad \text{(D.3)}$$

If you need to reference an equation from the text, you can use command `\eqref`. For example, to reference the equations above (D.1). If you want to align the equation number vertically, you can use command `split`:

$$
\begin{aligned}
3x &= 6y + 12 \\
x &= 2y + 4
\end{aligned}
\qquad \text{(D.4)}
$$

Mathematical symbols (α) and expressions can be placed even in text $\cos \pi = -1$ and can also be in a footnote[1].

Image D.1 displays a wide image comprised of multiple smaller images. Standard raster image is inserted in the same way as image D.2.

Figure D.1: **Wide image.** Image can be comprised of multiple smaller images. If you want to address the partial images from text, use packagae `subcaption`.

---

[1]Formula in a footnote: $\cos \pi = -1$

Figure D.2: Good text is a bad text, that has been changed countless times. You have to start somewhere.

Sometimes it is necessary to attach a diagram that does not fit on an A4 page. Then it is possible to insert one A3 page and fold it into the thesis (so-called Engineering fold, similar to Z-fold, where two folds are created – face down and face up). Switching is performed as follows: `\eject \pdfpagewidth=420mm` (210mm to switch it back).

Other frequently used commands can be found above in the text, because a single practical example of correct use is better than ten pages of examples.

Appendix E

Examples of bibliographic citations

The czplain style is based on the style created by mr. Pyšný [20]. This appendix contains a set of supported type of citations with specific examples of bibliograhpic citations.

The next pages of the appendix contain examples of bibliographic citations of the following pubplications and their parts:

- Article in a periodical literature (magazine) (str. 73),

- monographic publication (str. 74),

- conference proceedings (str. 75),

- conference proceedings entry or book chapter (str. 76),

- manual, documentation, technical report and unpublished materials (str. 77),

- academic work (str. 78),

- web page (str. 79),

- and web site (str. 80).

Items are color-coded depending on whether or not they are required or optional:

- required element according to the standard

- optional element according to the standard

- required element for online information sources according to the standard

- element that is not specified in the standard, but is available and optional within the template's bibliographic style

Required items are only stated if they exist.

The bibliography file contains records in the following form:

```
@Article{Doe:2020,
    author              = "Doe, John",
    title               = "How to cite",
    subtitle            = "Article citation",
    journal             = "Writing theses and dissertations",
    journalsubtitle     = "Formal aspects",
    howpublished        = "online",
    address             = "Brno",
    publisher           = "Brno University of Technology,
                           Faculty of information technology",
    contributory        = "Translated by Jan NOVÁK",
    edition             = "1",
    version             = "version 1.0",
    month               = 2,
    year                = "2020",
    revised             = "revised 12. 2. 2020",
    volume              = "4",
    number              = "24",
    pages               = "8--21",
    cited               = "2020-02-12",
    doi                 = "10.1000/BC1.0",
    issn                = "1234-5678",
    note                = "This a made up citation",
    url                 = "https://merlin.fit.vutbr.cz"
}
```

Article in a periodical literature - @Article

Record items

Element	BibTeX item	Example
Author	author	Doe, John
Article title	title	How to cite
Article subtitle	subtitle	Article citation
Periodical literature title	journal	Writing theses and dissertations
Periodical literature subtitle	journalsubtitle	Formal aspects
Type of medium	howpublished	online
Edition	edition	1
Version	version	version 1.0
Secondary author(s)	contributory	Translated by Jan NOVÁK
Place of publication	address	Brno
Publisher	publisher	Brno University of Technology, Faculty of information technology
Month	month	2
Year	year	2020
Volume	volume	4
Number	number	24
Pages	pages	8-21
Revision	revised	revised 12. 2. 2020
Date of citation	cited	2020-02-12
Series title	series	Guidelines for writing theses and dissertations
Number in series	editionnumber	42
Digital object identifier	doi	10.1000/BC1.0
Standard number	issn	1234-5678
Notes	note	This is a made up citation
Availability	url	https://merlin.fit.vutbr.cz

Bibliographic citation

Doe, J. How to cite: Article citation. *Writing theses and dissertations: Formal aspects* [online]. 1st ed., version 1.0. Translated by Jan NOVÁK. Brno: Brno University of Technology, Faculty of information technology. February 2020, vol. 4, num. 24, p. 8–21, revised 12. 2. 2020, [cit. 2020-02-12]. Guidelines for writing theses and dissertations, no. 42. DOI: 10.1000/BC1.0. ISSN 1234-5678. This is a made up citation. Available at: https://merlin.fit.vutbr.cz

Monographic publication - @Book, @Booklet (book, brochure)

Record items

Element	BibTeX item	Example
Author	author	John von Doe
Title	title	How to cite
Subtitle	subtitle	Monographic publication citation
Type of medium	howpublished	online
Edition	edition	1
Secondary author(s)	contributory	Translated by Jan NOVÁK
Place of publication	address	Brno
Publisher	publisher	Brno University of Technology, Faculty of information technology
Month	month	2
Year	year	2020
Revision	revision	revised 12. 2. 2020
Date of citation	cited	2020-02-12
Pages	pages	220
Series title	series	Guidelines for writing theses and dissertations
Number in series	editionnumber	2
Standard number	isbn	01-234-5678-9
Notes	note	This is a made up citation
Availability	url	https://merlin.fit.vutbr.cz

Bibliographic citation

VON DOE, J. *How to cite: Monographic publication citation* [online]. 1st ed. Translated by Jan NOVÁK. Brno: Brno University of Technology, Faculty of information technology, February 2020, revised 12. 2. 2020 [cit. 2020-02-12]. 220 p. Guidelines for writing theses an dissertations, no. 2. ISBN 01-234-5678-9. This is a made up citation. Available at: https://merlin.fit.vutbr.cz

Conference proceedings - @Proceedings

Record items

Element	BibTeX item	Example
Author*	author	Čechmánek, Jan
Editor*	editor	Čechmánek, Jan
Title	title	How to cite
Subtitle	subtitle	Conference proceedings citation
Type of medium	howpublished	online
Edition	edition	1
Secondary author(s)	contributory	Translated by Jan NOVÁK
Place of publication	address	Brno
Publisher	publisher	Brno University of Technology, Faculty of information technology
Month	month	2
Year	year	2020
Volume	volume	4
Number	number	24
Pages	pages	8-21
Revision	revised	revised 12. 2. 2020
Date of citation	cited	2020-02-12
Series title	series	Guidelines for writing theses and dissertations
Number in series	editionnumber	2
Digital object identifier	doi	10.1000/BC1.0
Standard number	isbn or issn	01-234-5678-9
Notes	note	This is a made up citation
Availability	url	https://merlin.fit.vutbr.cz

*Either author or editor is stated.

Bibliographic citation

ČECHMÁNEK, J. *How to cite: Conference proceedings citation* [online]. 1st ed. Translated by Jan NOVÁK. Brno: Brno University of Technology, Faculty of information technology, February 2020, vol. 4, num. 24, p. 8–21, revised 12. 2. 2020 [cit. 2020-02-12]. Guidelines for writing theses and dissertations, no. 2. DOI: 10.1000/BC1.0. ISBN 01-234-5678-9. This is a made up citation. Available at: `https://merlin.fit.vutbr.cz`

Conference proceedings entry or book chapter - @InProceedings, @InCollection, @Conference, @InBook

Record items

Element	BibTeX item	Example
Author	author	John von Doe
Entry title	title	How to cite
Entry subtitle	subtitle	Article citation
Parent document author	editor or organisation	Smith, Peter
Parent document title	booktitle	Conference proceedings on writing theses and dissertations
Parent document subtitle	booksubtitle	Formal aspects
Type of medium	howpublished	online
Edition	edition	1
Version	version	version 1.0
Parent document secondary author(s)	contributory	Translated by Jan NOVÁK
Place of publication	address	Brno
Publisher	publisher	Brno University of Technology, Faculty of information technology
Month	month	2
Year	year	2020
Volume	volume	4
Number	number	24
Chapter	chapter	5
Pages	pages	8-21
Revision	revised	revised 12. 2. 2020
Date of citation	cited	2020-02-12
Series title	series	Guidelines for writing theses and dissertations
Number in series	editionnumber	2
Standard number	isbn or issn	1234-5678
Notes	note	This is a made up citation
Availability	url	https://merlin.fit.vutbr.cz

Bibliographic citation

DOE, J. How to cite: Article citation. In: SMITH, P., ed. *Conference proceedings on writing theses and dissertations: Formal aspects* [online]. 1st ed., version 1.0. Translated by Jan NOVÁK. Brno: Brno University of Technology, Faculty of information technology, February 2020, vol. 4, num. 24, chap. 5, p. 8–21, revised 12. 2. 2020 [cit. 2020-02-12]. Guidelines for writing theses and dissertations, no. 2. ISSN 1234-5678. This is a made up citation. Available at: https://merlin.fit.vutbr.cz

Manual, documentation, technical report and unpublished materials - @Manual, @TechReport, @Unpublished

Record items

Element	BibTeX item	Example
Author (person or organisation)	author	Brno University of Technology, Faculty of information technology
Title	title	Manual for writing theses and dissertations
Subtitle	subtitle	Manual citation
Type of medium	howpublished	online
Document type	type	User manual
Document number	number	3
Edition	edition	1
Secondary author(s)	contributory	Edited by Jan NOVÁK
Place of publication	address	Brno
Organisation or institution	organization or institution	Brno University of Technology, Faculty of information technology
Month	month	2
Year	year	2020
Revision	revised	revised 12. 2. 2020
Date of citation	cited	2020-02-12
Pages	pages	220
Notes	note	This is a made up citation
Availability	url	https://merlin.fit.vutbr.cz

Bibliographic citation

Brno University of Technology, Faculty of information technology. *Manual for writing theses and dissertations: Manual citation* [online]. User manual 3, 1st ed. Edited by Jan NOVÁK. Brno: Brno University of Technology, Faculty of information technology, February 2020, revised 12. 2. 2020 [cit. 2020-02-12]. 220 p. This is a made up citation. Available at: https://merlin.fit.vutbr.cz

Academic work - @BachelorsThesis, @MastersThesis, @PhdThesis, @Thesis

Record items

Element	BibTeX item	Example
Author	author	Brno University of Technology, Faculty of information technology
Title	title	BiBTeX style for ČSN ISO 690 and ČSN ISO 690-2
Subtitle	subtitle	
Type of medium	howpublished	online
Document type	type	Dissertation
Place of publication	address or location	Brno
School	school	Brno University of Technology, Faculty of information technology
Year	year	2020
Date of citation	cited	2020-02-12
Pages	pages	220
Appendices	inserts	20
Standard number	isbn	01-234-5678-9
Supervisor	supervisor	Dytrych, Jaroslav
Notes	note	This is a made up citation
Availability	url	https://www.fit.vut.cz/study/theses

Bibliographic citation

NOVÁK, J. *BiBTeX style for ČSN ISO 690 and ČSN ISO 690-2* [online]. Brno, CZ, 2020. [cit. 2020-02-12]. 80 p., 20. p. apps. Dissertation. Brno University of Technology, Faculty of information technology. ISBN 01-2345-678-9. Supervisor DYTRYCH, J. This is a made up citation. Available at: https://www.fit.vut.cz/study/theses

Web page - @Webpage

Record items

Element	BibTeX item	Example
Author	author	Nováková, Jana
Page title	secondarytitle	Post citation
Site title	title	Web on writing theses and dissertations
Site subtitle	subtitle	
Type of medium	howpublished	online
Secondary author(s)	contributory	Edited by Jan NOVÁK
Version	version	version 1.0
Place of publication	address	Brno
Publisher	publisher	Brno University of Technology, Faculty of information technology
Day	day	12
Month	month	2
Year	year	2020
Time of publication	time	14:00
Revision	revised	revised 12. 2. 2020
Digital object identifier	doi	10.1000/BC1.0
Standard number	issn	1234-5678
Notes	note	This is a made up citation
Availability	url	https://merlin.fit.vutbr.cz
Path	path	Home; Art; The art of citation

Bibliograpic citation

NOVÁKOVÁ, J. Post citation. *Web on writing theses and dissertations* [online]. Edited by Jan NOVÁK. version 1.0. Brno: Brno University of Technology, Faculty of information technology, 2. february 1998 14:10. revised 12. 2. 2020 [cit. 2020-02-12]. DOI: 10.1000/BC1.0. ISSN 1234-5678. This is a made up citation. Available at: `https://merlin.fit.vutbr.cz` Path: Home; Art; The Art of Citation.

Web site - @Website

Record items

Element	BibTeX item	Example
Author (person or organisation)	author	Nováková, Jana
Site title	title	Web on writing theses and citations
Site subtitle	subtitle	
Type of medium	howpublished	online
Secondary author(s)	contributory	Edited by Jan NOVÁK
Version	version	version 1.0
Place of publication	address	Brno
Publisher	publisher	Brno University of Technology, Faculty of information technology
Day	day	12
Month	month	2
Year	year	2020
Time of publication	time	14:00
Revision	revised	revised 12. 2. 2020
Date of citation	cited	2020-02-12
Digital object identifier	doi	10.1000/BC1.0
Standard number	issn	1234-5678
Notes	note	This is a made up citation
Availability	url	https://merlin.fit.vutbr.cz

Bibliographic citation

NOVÁKOVÁ, J. *Web on writing theses and dissertations* [online]. Edited by Jan NOVÁK. version 1.0. Brno: Brno University of Technology, Faculty of information technology, 2. february 1998 14:10. revised 12. 2. 2020 [cit. 2020-02-12]. DOI: 10.1000/BC1.0. ISSN 1234-5678. This is a made up citation. Available at: `https://merlin.fit.vutbr.cz`.