BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

CONFIGURABLE PARALLEL EXECUTION OF SYSTEM TESTS WITHIN THE STRIMZI PROJECT

KONFIGURATEĽNÁ PARALELNÁ EXEKÚCIA SYSTÉMOVÝCH TESTOV V RÁMCI PROJEKTU STRIMZI

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. MAROŠ ORSÁK**
AUTOR PRÁCE

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

BRNO 2022

Abstract

In recent years, many companies adopted the Kubernetes system and architecture of Microservices. This technology opened up a lot of new possibilities not just for a big companies, but also for a small users of the containers. Since Kuberentes is container-orchestration system a new idea about how to orchestrate the containers came – the Operator pattern. and with it appeared the operator pattern. One of these operators is developed under project Strimzi. This project gathers together several tools, which takes care of Apache Kafka deployed on top of Kubernetes. Since Kafka is a vast system, horizontally scalable middleware product, you can imagine that the installation of such system is a relatively complex action. Therefore, one of the biggest challenges of the Kubernetes environment is how to effectively and quickly test systems such as Strimzi and at the same time verify integration with other similar products. Additionally, the resources available at Kubernetes are much more demanding and thus rank among the most complex 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 create a mechanism within Strimzi system tests, which runs in parallel against only a single Kubernetes cluster. Furthermore, we proposed a brand new architecture of 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 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

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

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.

.....
Maroš Orsák
September 23, 2021

Acknowledgements

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

Contents

Chapter 1

Introduction (TODO)

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. In the past, when we wanted to speed up problem solving, we wanted to create something that will eliminate the time we spent on calculations. And so we invented the computer, which at the beginning knew relatively nothing to do. However, after a few years, all this changed and computer was able to solve problems that took a person many days. Today we live in a time when computers has significantly improved execution time by solving different problems using parallelism.

- 1. odstavec úvaha a motivácia paralelizácie...dnešný trend...
- 2. odstavec pár slov ku Strimzi...
- 3. bottle-nech prístupu ku aktuálnemu testovaniu
- 4. návrh na vyriešenie aktuálnych problémov
- 5. implementácia a ohodnotenie experimentov čo sa zistilo...
- 6. štruktúra diplomky...

Chapter 2

Preliminaries

This chapter provides a fundamentals of technologies used across the whole thesis. Notable technologies used are Kubernetes[1], Apache Kafka[2] and Strimzi[3], which are described in details. Note that author has published description of this technologies in his bachelor's thesis [14]. On the other hand, the author tried to explain the obscure things better and more clearly, and at the same time keep the things that were very well described. Furthermore, some ideas related to the Kubernetes were taken from the *The Kubernetes book* [18]. The section 2.4 describes the e2e 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* [16] and *How system tests work* [15] and has taken over a few sections that were necessary.

2.1 Kubernetes

A few years ago came a new concept of container management. This concept has opened the door to many products to simplify their management of 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 master and slave architecture. Master node controls Kubernetes resources. On the other hand, the slave 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]. And in each direction, we have eliminated certain disadvantages based on empirical knowledge.

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

[2] Apache Kafka – distributed messaging system originally founded 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/)

1. **Physical** — The first phase of how to deploy applications was to execute the program on the physical computer. Nevertheless, this approach was not as practical 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 environment, one has to buy five physical servers, which could cost a significant amount of money. All of this led to the formation of the follow-up phase.

2. **Virtualization** — The next phase has solved problems like scalability, security, and also price. This scenario of the applications can run on a single machine without sharing memory, which means it is isolated and encapsulated from the outside world. Furthermore, you can run four 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 security is much higher. However, resource consumption is still high since each virtual machine includes an Operating System. At the same time, the management of these entities is not easy if we imagine production with hundred virtual machines. Sometimes applications need to share information, and with that comes another phase.

3. **Containerization** — In the last phase, containerization is considered as a lightweight 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 container images instead of using virtual machine images

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

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. That is the point where 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.

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

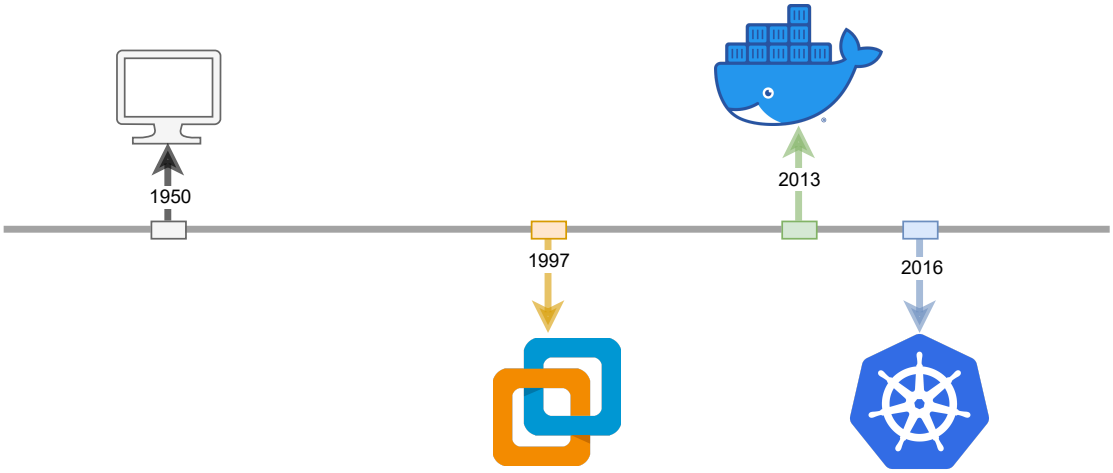


Figure 2.1: Evolution of virtual technologies

Afterwards, the lightweight era comes with an idea whose functionality was based on containerization [11]. Finally, we have a manager who takes care of the overall management of the individual containers and guarantees their reliability, scales it pretty effectively and more. This is what we call container orchestration system [1]. It has the following entities:

(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

These Linux hosts can be virtual machines, bare-metal servers in the data centre, or private or public cloud instances. Production environments typically have more than one master node running because the need of High Availability[6]). Kubernetes services from the biggest cloud providers such as Azure Kubernetes Service (AKS), Amazon Elastic Kubernetes service (EKS), and Google Kubernetes service (GKS) has 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* taking care of all controllers where each of these controllers runs as a separate process. *Node controller* 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.

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

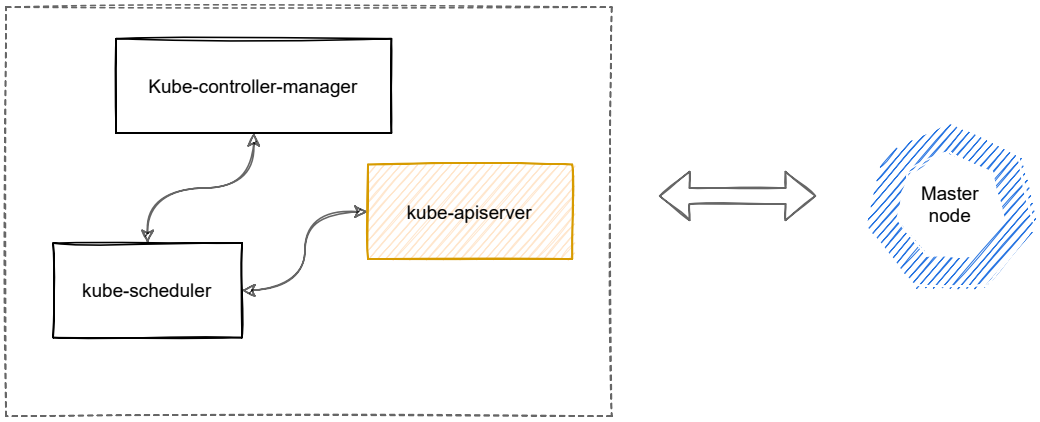


Figure 2.2: Representation of the Master node

Algorithm 1 Generic algorithm for each Kubernetes controller

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

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

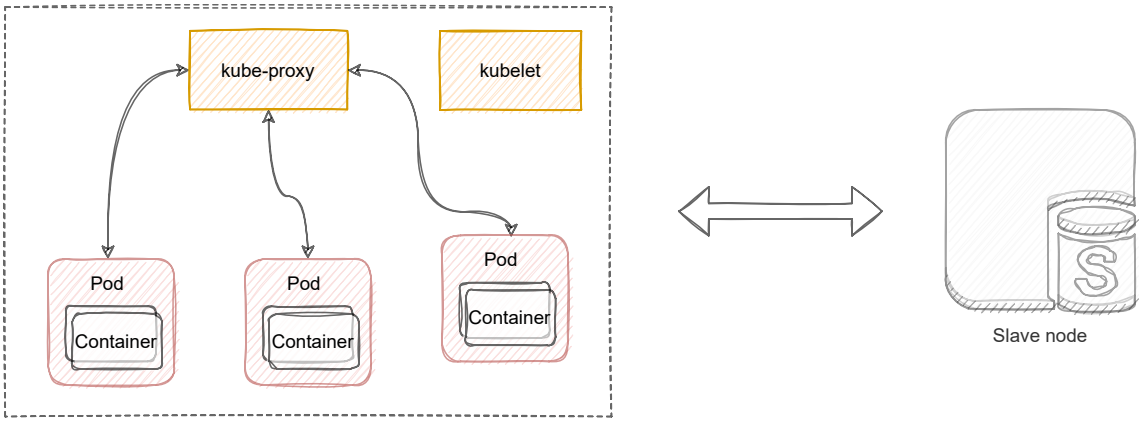


Figure 2.3: Representation of the Slave node

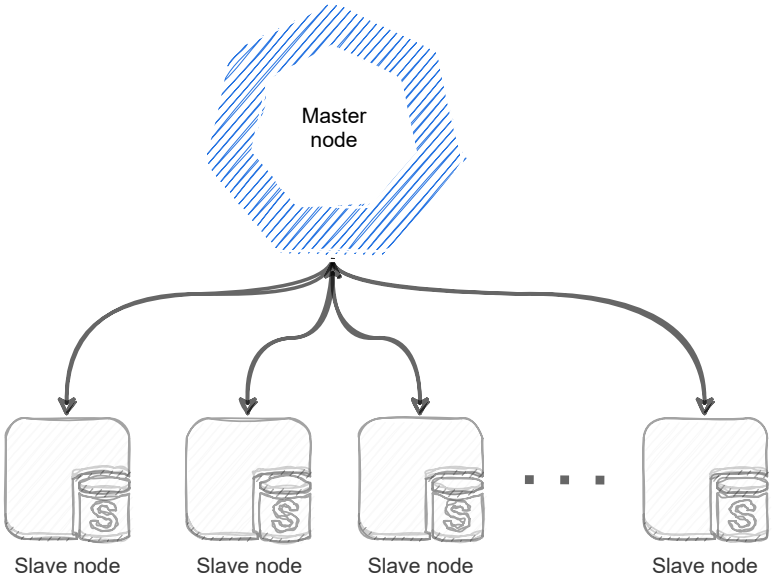


Figure 2.4: Relation between master and slave nodes

2.1.3 Common objects

1. **Pod** – is the atomic unit of the Kubernetes. For instance, in the VMware environment, the atomic unit is a virtual machine, and the Docker has a container. The term Pod originated from the Docker logo. If we think about it, Docker has one whale on his logo, and we call a group of such whales Pod or, in other words, Pod of whales. Deductively, we can find out the property of the Pod, that is, that one or more containers can run in it. These containers share storage, network and specification 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 a way how particular components communicate. Services provide reliable networking for a set of Pods. This means that if Pod fails and Kubernetes create a new Pod, the IP address is changed. Moreover, operations like scaling up or scaling down do the same. This is where Services comes 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 how one can expose your 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 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 one physical. *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 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 bind to a Pod. The main ideas behind volumes are the following: at first, assume a scenario when your Pod crashed, and the application will lose all 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 a great responsibility for the Rolling Update and Rollback operations available in Deployment. We will talk more about them in the next section.

2. **Deployment** – It is one of the most widely used application management controllers in the Kubernetes environment.

   Based on our knowledge, the skilful 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

[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

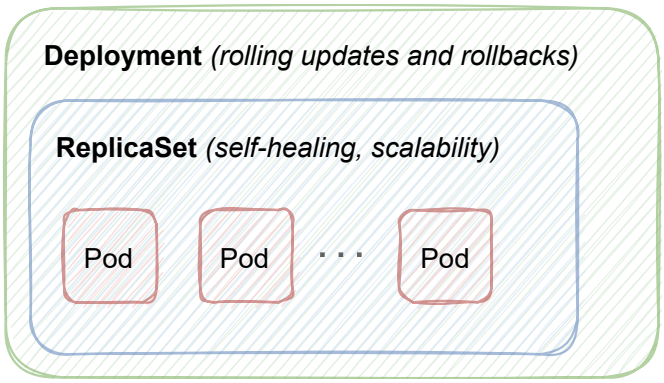


Figure 2.5: Hierarchy of Deployment, ReplicaSet and Pod inspired by The Kubernetes Book [18]

features at its disposal. Noteworthy, 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 us the ability to define such an object in YAML files, such an object can then be edited and updated, which will trigger 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 like 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:

   - **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 not care about order).
   - **scaling operation** – ensures that each new Pod is installed before 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.

   With that being, 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 StatefulSet. When the Pod is created, Statefulset immediately creates an associated volume and attach this Volume to the Pod, which shows Figure 2.6. This guarantees that Pod can keep all the information even in the event of an unexpected failure.

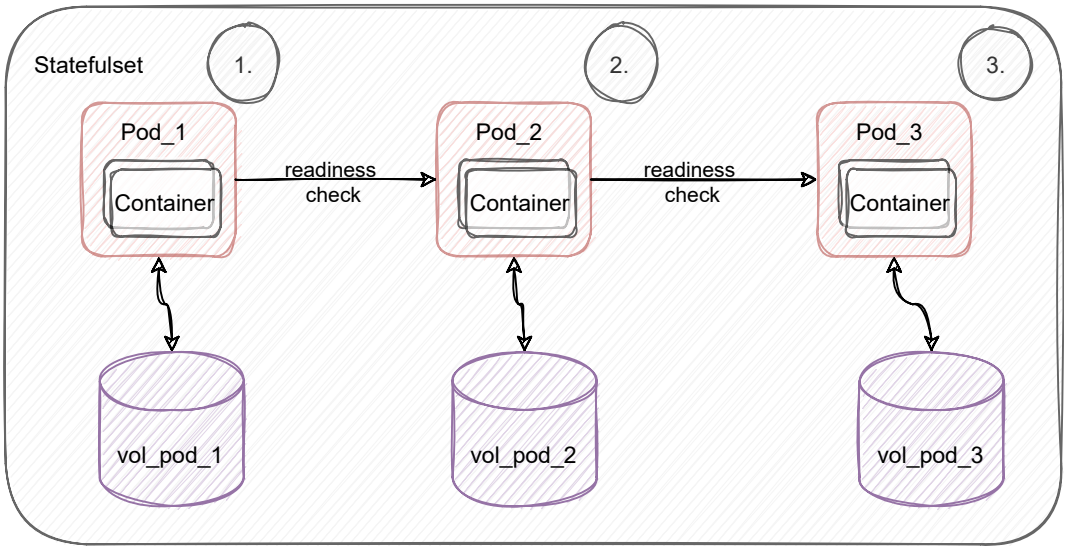


Figure 2.6: StatefulSet ordered creation of Pods

2.2 Apache Kafka

This section explains and defines the basics of the Apache Kafka system. The description is based on two books *Designing Event-Driven Systems* [22] and *Real-Time Data and Stream-Processing at Scale* [13]. Moreover, the Kafka streams subsection is based on *Mastering Kafka Streams and ksqlDB Building real-time data systems* [12]. We also used Kafka's documentation as the most up-to-date reference [2]. In these books and documentation, there can be found a more detailed explanation of Kafka itself.

Apache Kafka is a event streaming platform that offers many features like high performance, distribution, commit log service[8], and more. It publishes and subscribes 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. This technology is widely used by many big companies like LinkedIn, Spotify, Netflix, and Uber.

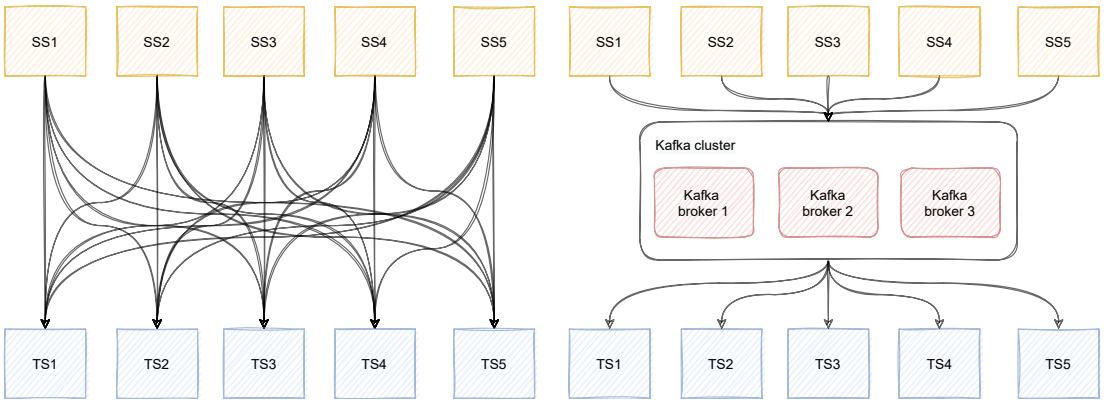
2.2.1 Motivation [14]

In the past, companies had applications or systems that share many data. Moreover, the application was able to provide some valuable information to another application. So, there was one source system and one target system. But what about adding some 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.

Therefore has twenty-five links, which is not effective i.e 2.7 (quadratic complexity). That is why Kafka was invented. Let 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 is bind to the Kafka broker, and all data are delivered by a single link. You can see the updated system in the Figure 2.7.

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

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



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

Figure 2.7: How to make system more efficient with Kafka

2.2.2 Fundamental concepts

In this subsection, we describe fundamental concepts of Apache Kafka such as Producer, Consumer, Kafka broker, Kafka cluster and so on. The description is based on the *Real-Time Data and StreamProcessing at Scale* [13] and partially [2].

1. **Producer** – are 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 world. Kafka also provides the implementation of these clients in several languages such as Java, Go, C++, Python and many others. Noteworthy, Kafka also provides the client with a higher abstraction, which means that it is no longer necessary to create the producers themselves, but those entities are encapsulated in the client. These are, for example, Kafka Streams for stream processing or Kafka Connect API for data integration.

2. **Consumer** – Unlike producing messages, a consumer or group of consumers tries to obtain messages. At the same time, as a producer, 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 lets them know where it is when reading the data from topic. The method that consumers use is called polling[10]. The consumer group behaves as an individual logical unit. Kafka does not support reading from one specific partition with two or more consumers simultaneously. The reason why this concept was created is based on straightforward questions. *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 comparing with the traditional messaging based on AMQP protocol like ActiveMQ or RabbitMQ.

3. **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

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

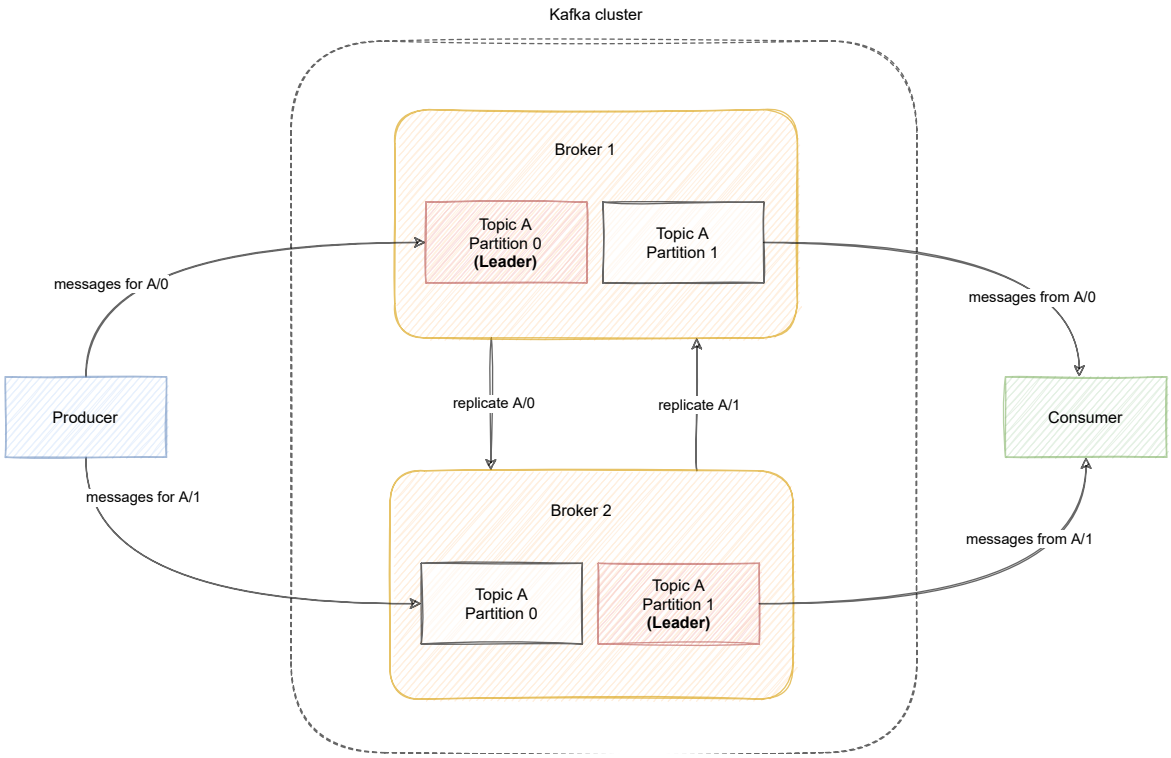


Figure 2.8: Kafka topic partition replication scenario in Kafka cluster inspired by *Real-Time Data and StreamProcessing at Scale* [13]

takes care about 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 Kafka cluster, we always know who is the cluster controller. The cluster controller takes care of fundamental operations such as assigning partitions to brokers or monitoring for the failure of Kafka brokers. One broker in the Kafka cluster always owns the topic partition. This broker is called the leader of this topic partition. Of course, this topic partition can be replicated into several Kafka brokers, which will result in its replication and thus data redundancy. On the other hand, if the leader Kafka broker fails, the one who has the replicated topic partition will take control and become the new partition leader. If the leader Kafka broker fails, the one with the replicated topic partition can 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 topic are replicated.

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 in a specific topic. A replication factor is a number, which defines how many replicas will be available on the other brokers from 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 a command-line interface. (In the Section 2.3, we will also talk about an alternative way of creating resources.)

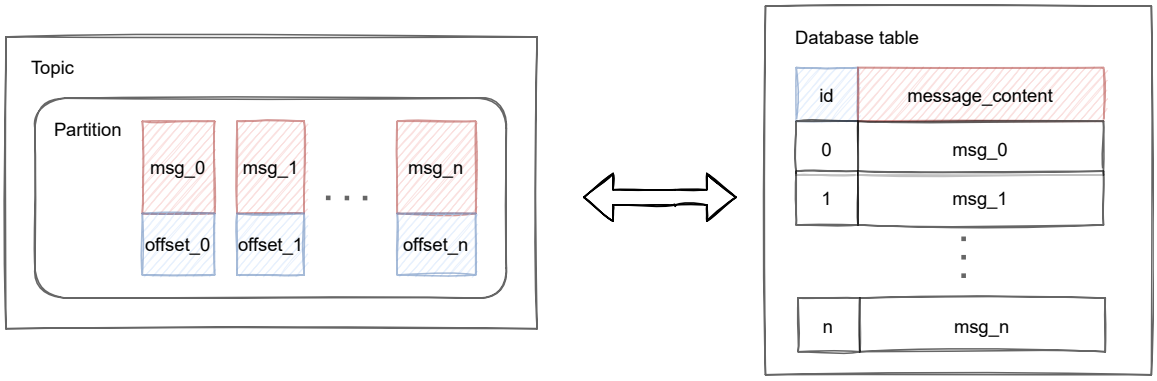


Figure 2.9: Equivalence of Kafka topic and database table

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 an entities that splits 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 offset, which serves as an id for the specific message. 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 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, e.g. one to 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.

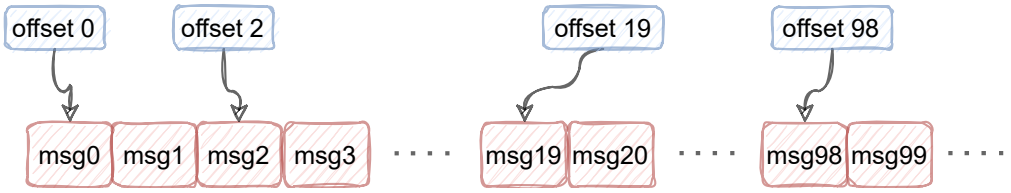


Figure 2.10: Partition offset

2.2.3 Kafka Streams

It is a stream processing tool created by the Kafka community that does not suffers from 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 payoff, and it is writing many lines of code. Unfortunately, we cannot classify these APIs into stream processing APIs because they do not contain primitives that would classify them there, such as *Local* and *Fault-tolerant* state and a set of transformers that work with data (a transformer is an operator that transforms data).

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

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.

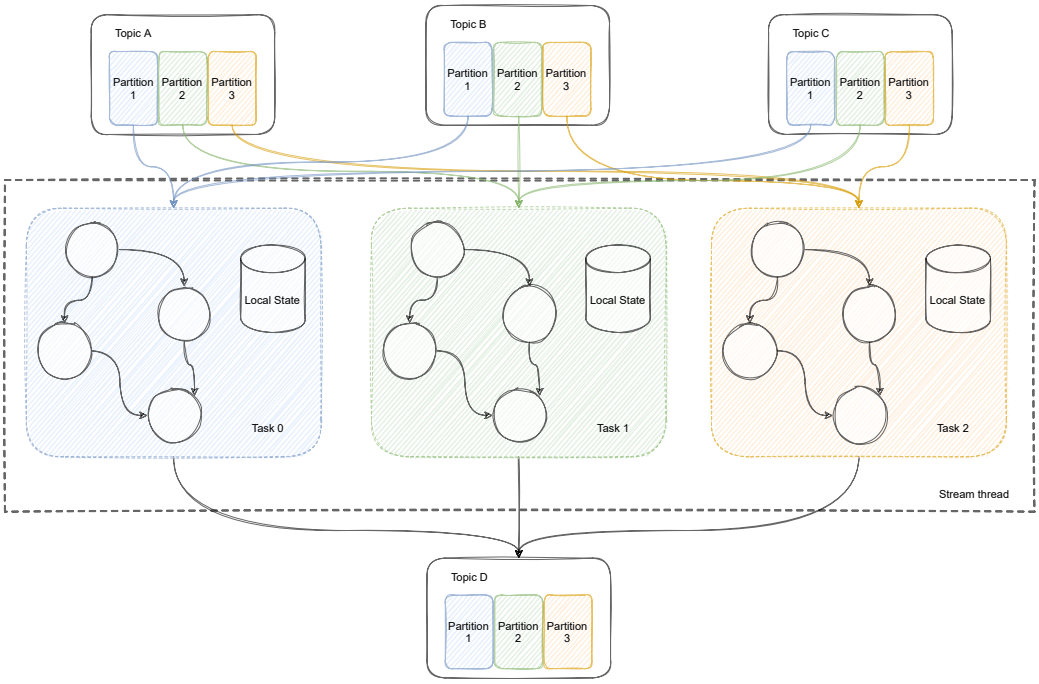


Figure 2.11: Kafka Streams with local state stores inspired by Kafka Documentation [2]

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 fell 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 to run the given error is corrected, Kafka will rebalance again. These sentences only prove that Kafka Streams is a fault-tolerant and provides reliability.

One of the main differences between other similar systems is the processing model that Kafka Streams offers. These systems, such as *Apache Spark Streaming*[12] or *Trident*[13], use micro-batching, which occurs very much in neural networks where work is divided into

[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.

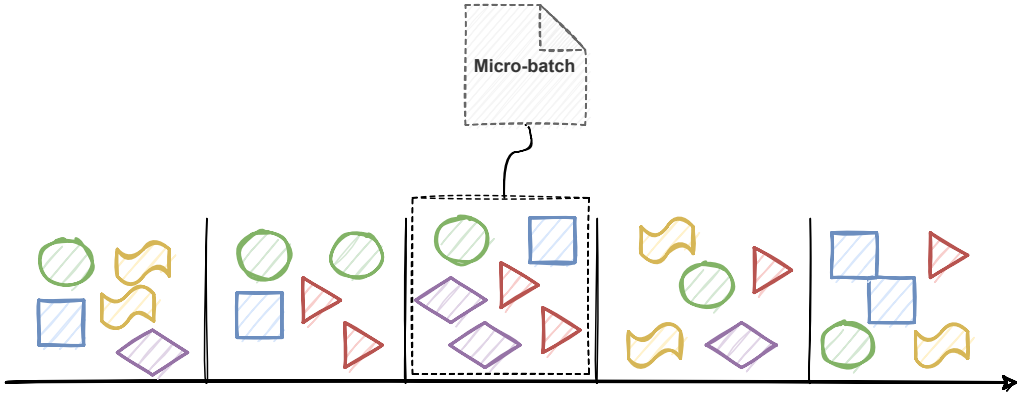


Figure 2.12: Micro-batching processing (typical for different systems) inspired by [12]

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.

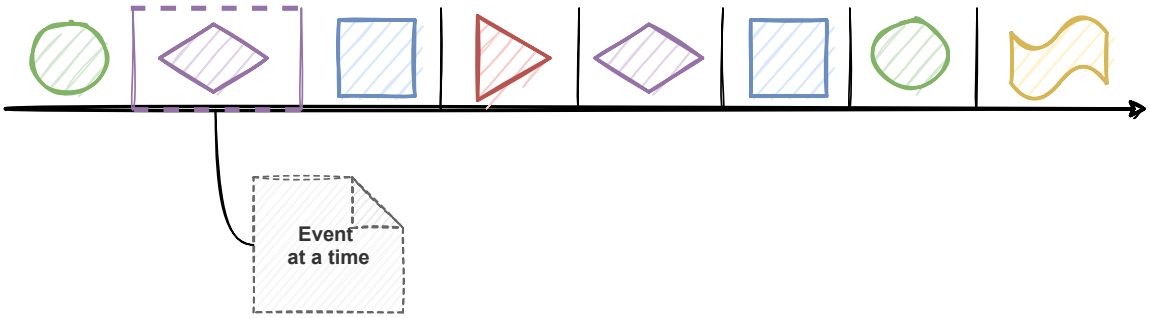


Figure 2.13: Kafka Streams uses the event at a time processing inspired by *Mastering Kafka Streams and ksqlDB Building real-time data systems* [12]

Kafka Streams is thus a set of libraries that offer developers incredible power over data processing. Additionally, it have a model of parallelism, where the logical unit partitions. 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 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).

We have 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 is asking: „*How to move data from Kafka to datastore or vice versa?*“. At the same time, 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 very quickly and solve the data integration problem thanks to the Kafka Connect component.

Kafka Connect offers an incredible 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 datastore are changed, the data is automatically sent to Topic 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. Noteworthy, there are currently many such converters available, and it is only about the configuration.

All the mentioned Kafka components can be divided into three stages, which Kafka itself went through. 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. To one, the Kafka ecosystem would not

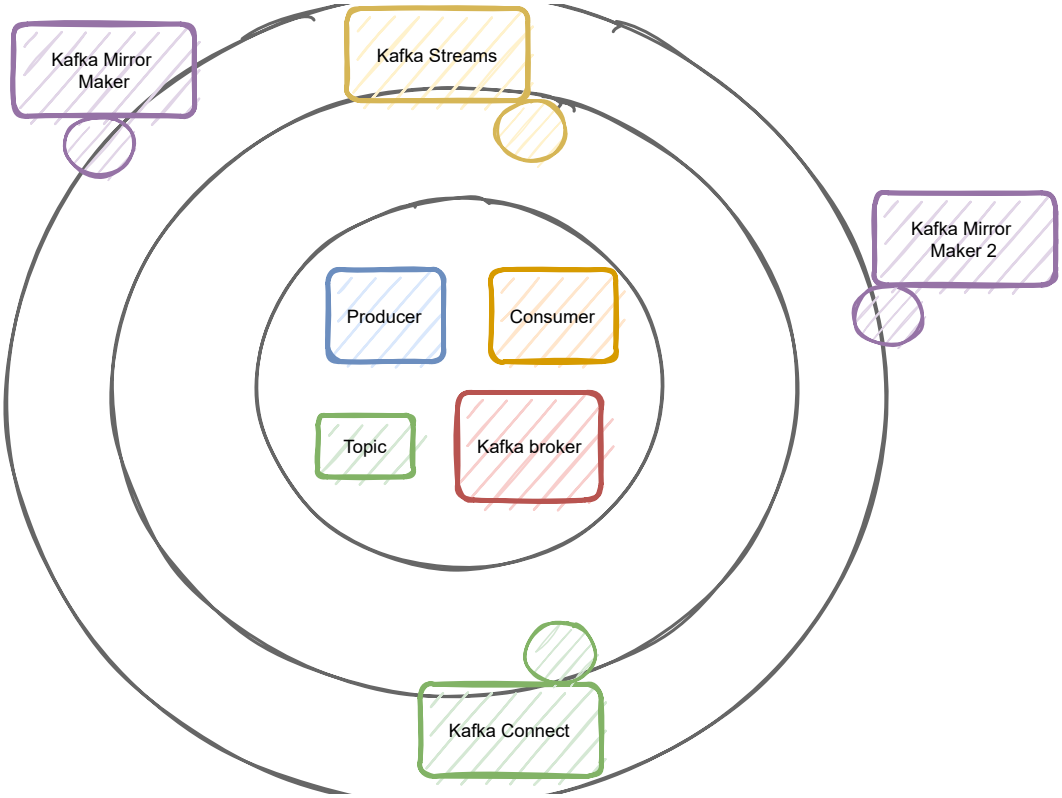


Figure 2.14: The entire Apache Kafka ecosystem.

seem small, where we have described its main elements. Figure 2.14 shows these stages 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.

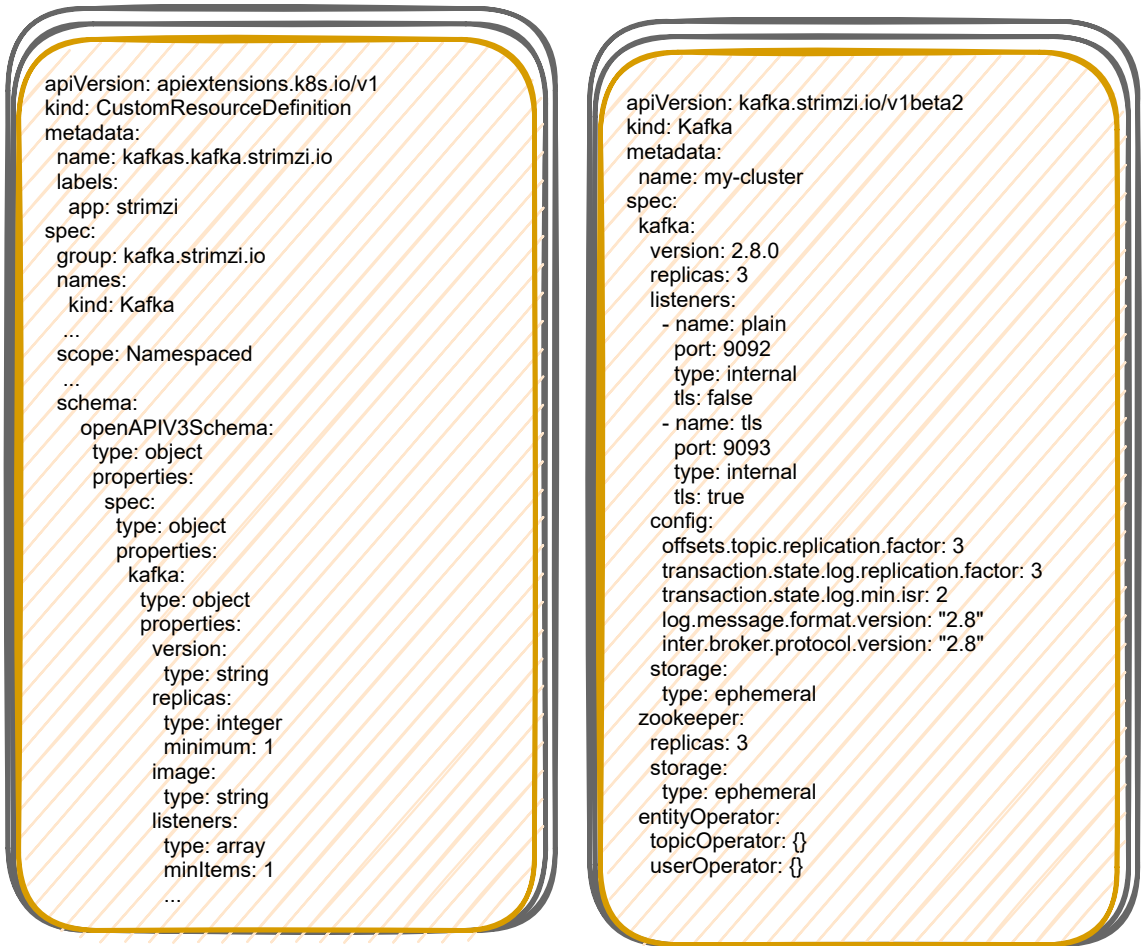
2.3 Strimzi

This section describes and defines the fundamental parts of the Strimzi product. Moreover, it explains the whole architecture with all Operators (i.e., Topic, User, Cluster). The description is based mainly on [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 work 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 this 2.1.2 algorithm. The main difference is that the Operator oversees Custom Resources. Custom Resource is an extension of the Kubernetes API. These CRs defines 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. At the same time, Custom Resource is already an instance of the class. Strimzi has a defined Custom Resource Definition for each Kafka component we described in section 2.2 except for clients. For example, for the KafkaBroker component, Strimzi has its Custom Resource Definition and others.



(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. Kafka Custom Resource Definition (Figure 2.15 – left side) 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** – In 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, which 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* we have a restriction that it must have at least one replica and similarly for other attributes. For imagination, the current version of Strimzi 0.25.0 Kafka Custom Resource Definition consists of 7000 lines of code.

On the other hand, we have Kafka Custom Resource (Figure 2.15 - right side), which includes parts worth of 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 installed,
- **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. Furthermore, a second listener will use encrypted communication using TLS technology and listen on port 9093.
- **spec.kafka.config** – these are additional configuration features that are added to Kafka,
- **spec.kafka.storage** [21] – the type of storage. We know two types of storage that Kubernetes has been supporting for a long time. In our case, it is ephemeral storage. Ephemeral storage is usually a directory somewhere in the operating system on our Kubernetes node. It works the same as the temporal directory. There are also risks associated with this, when the Kubernetes node may crash, and the data stored in the given 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 installed,
- **spec.entityOperator** – instance of Entity Operator with default configuration.,

2.3.1 Architecture

Strimzi architecture consists of two larger units, where the first unit is 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 then 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)

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. Whether it is scalability, metadata management or deployment itself, the answer to these problems was also in 2020 Kafka Improvement Proposal abbreviated KIP[14], which was accepted. Thus, Kafka 3.0 should already be without Zookeeper's dependency. His 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 them from the section 2.2, so their objective is clear. On the other hand, HTTP clients communicate with Kafka Bridge and thus connect the Kafka cluster and the clients themselves. It communicates by default via the REST API, and the user can

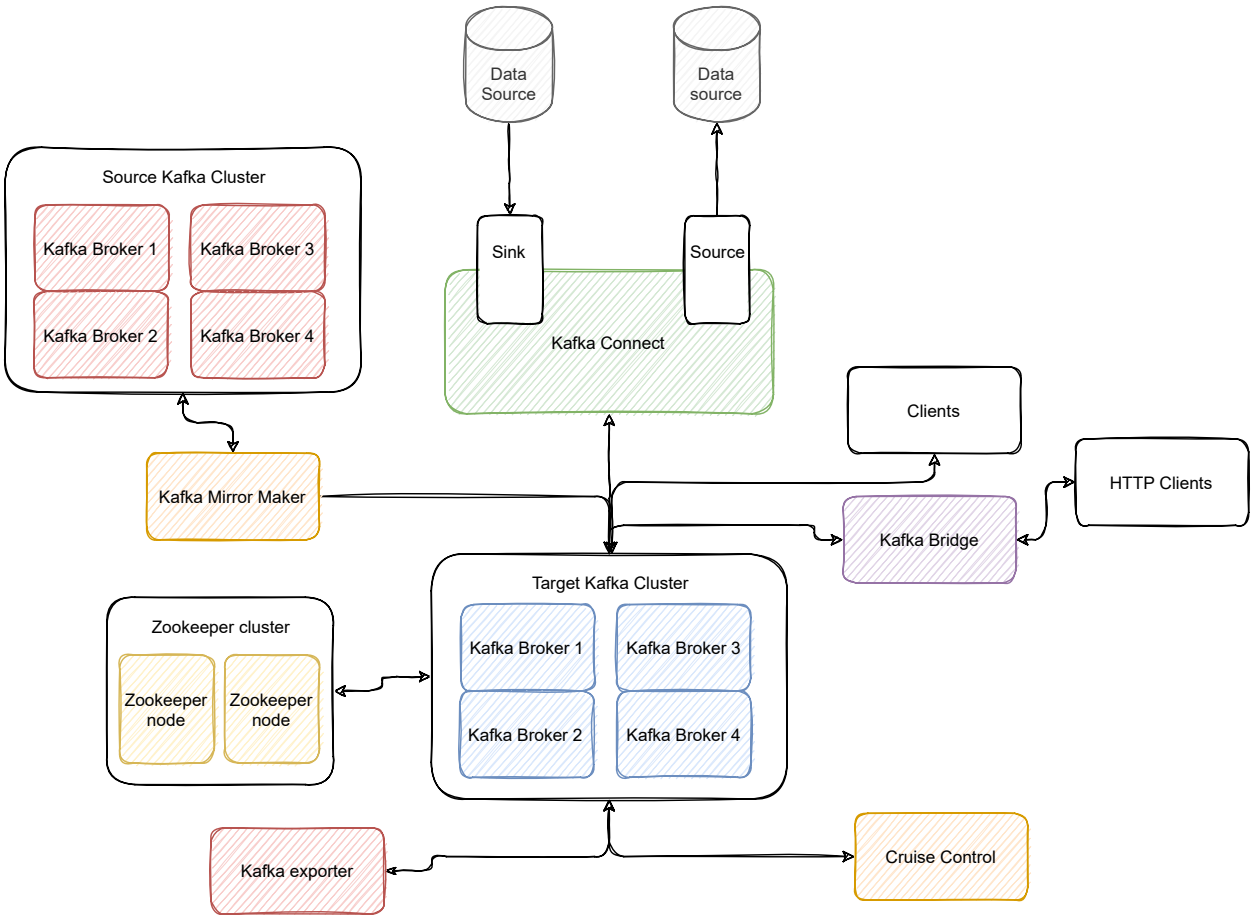


Figure 2.16: Strimzi Kafka architecture

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

create, delete, 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 data from the 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 pull-out, partitions status and many other metrics. Cruise Control creates an information model and analyzes when necessary to perform balancing and rearrange the load. Everything we have described shows the following 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, which manages Kafka, Kafka Mirror Maker, Kafka Mirror Maker 2, Kafka Connect, Kafka Rebalance and Kafka Bridge Custom Resources. Furthermore, since Kafka Custom Resource encapsulates the Entity Operator (Topic and User Operator running in the same Pod but 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 is Cluster Operator responsible.

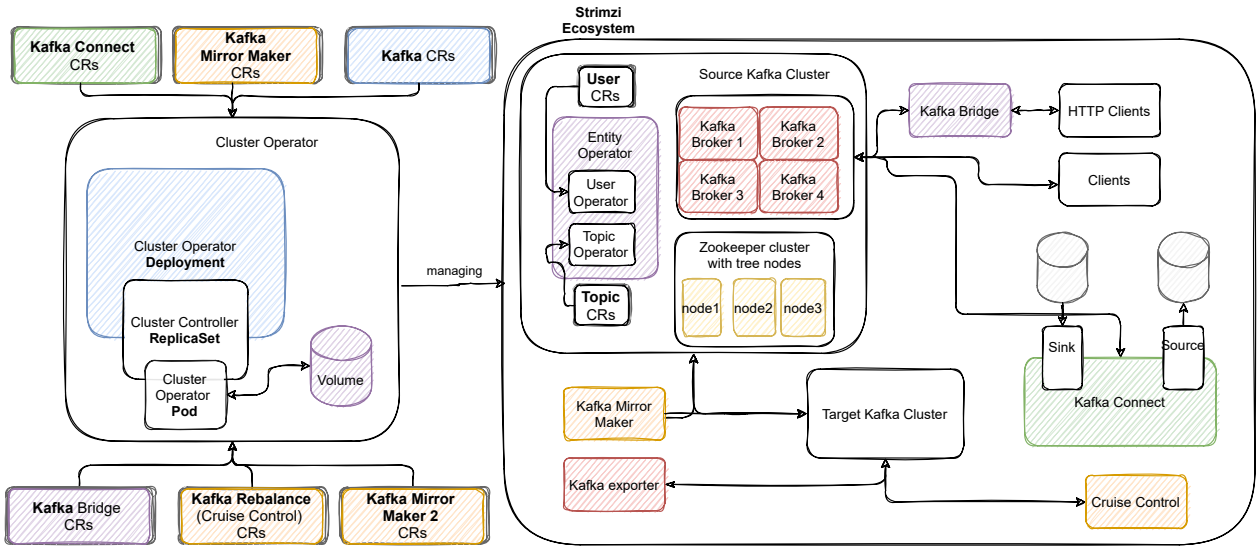


Figure 2.17: Strimzi Operators architecture with Strimzi ecosystem

Topic Operator takes care of creating, deleting, updating individual Topics. It is also necessary to mention that Topic Operator ensures synchronization between Custom Resource Topic and Topic (located inside Kafka container) and keep them in sync. *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 where the intersection is not empty.*

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

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 name. In other words, we can define read, 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. Noteworthy, when one creates Kafka Custom Resource, then immediately 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 control 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. We start with a short description of how we test the Strimzi product. Then in the 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.

It all starts with a regression, where we begin with unit tests, integration tests, and system tests. Of course, the most time-consuming is system tests, which in our case take about 40 hours. 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 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 stacks that take care of the total test case lifecycle. These stacks are responsible for storing all resources based on the test case. Furthermore, the deletion of these resources is transparent for the user as well as if it is a resource created in *@BeforeAll*[16] annotation. 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

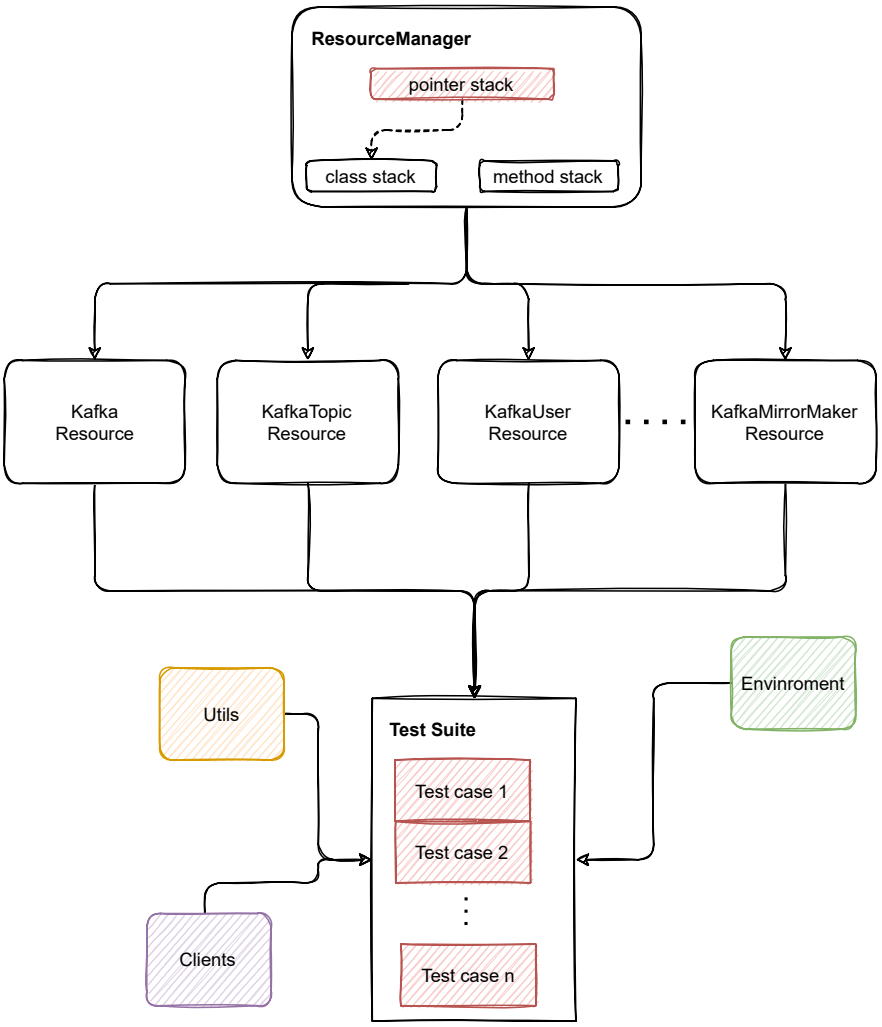


Figure 2.18: Strimzi system tests top-level component architecture

2.4.1 JUnit5 relation and execution of test cases

The entire implementation and management of the test lifecycle are in charge of 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 can be seen in the 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 have created a Kubernetes cluster 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 Kubernetes client `https://github.com/fabric8io/kubernetes-client`. This client provides a large 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 resource. In practice, we will also encounter the term CRUD methods. To illustrate, we can imagine getting all the namespaces on a given Kubernetes cluster. All namespaces are obtained using the command *client.namespaces().List();*.

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 the 2.4 Section, 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 deleting resources 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 the number of Test

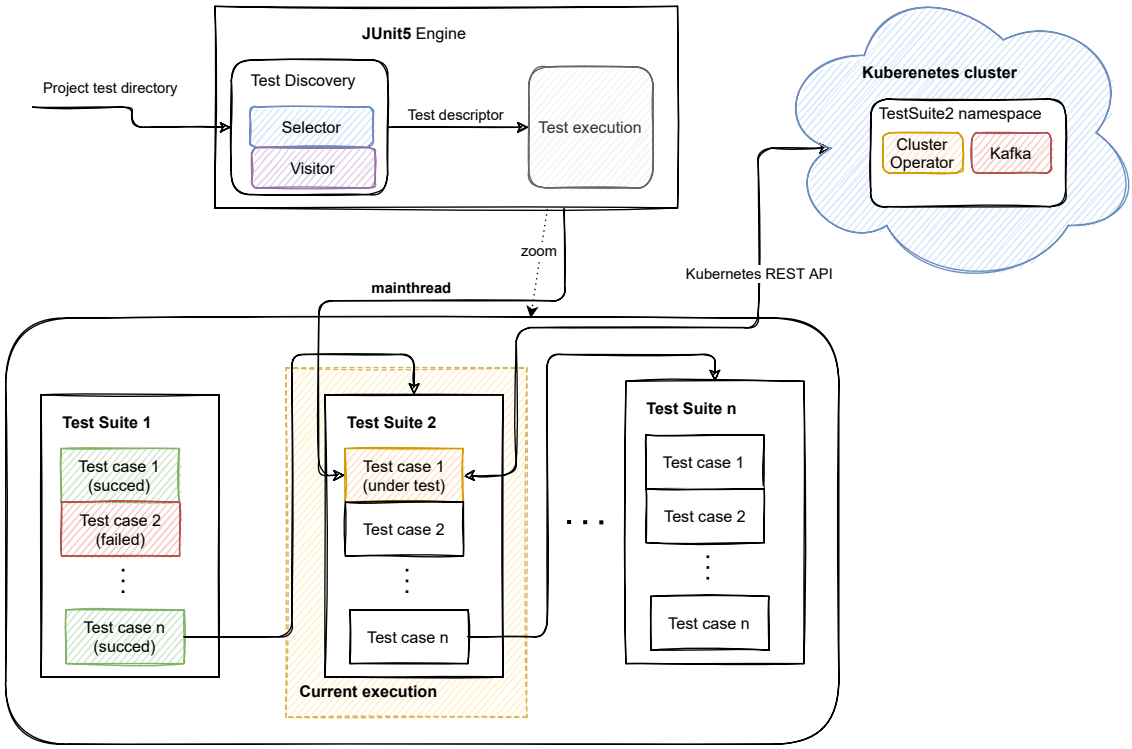


Figure 2.20: Strimzi system tests execution flow

suites where Test Suite 2 is currently executed and specifically Test Case 1. The attentive reader will somehow realize that the overall execution model is sequential due to Java's main thread, the primary thread identifier.

Chapter 3

Theory of paralelization

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)). This chapter is based on the following books *An Introduction to Parallel Programming* [17] and *The Art of Multiprocessor Programming* [9].

In the past, computers did not have an operating system. They could only execute one program at a time from start to end. The programmers of the time were as respected as the virtuoso in music and the arts. Writing such programs has been highly challenging. This problem was solved by developing operating systems that can run several processes (programs). Processes use the so-called variant of coarse-grained communication. Coarse-grained communication includes primitives such as sockets, signals, semaphores, shared memory, and files. This variant allows them to communicate with each other using signals, files or shared memory. These 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. If we connect all the parts, we will have a model called Sequential. This model is used by most of today's programming languages. Specifically, the sequential programming model is intuitive because it creates a sequence of operations that follow each other, thus making the expected result. However, this model has its limitations on performance and time consumption on specific tasks. During the twentieth century, technological advances brought a regular increase in the clock's speed, so that the software really „accelerated“ itself over time. However, 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 this parallelism is one of the great challenges of modern informatics.

3.1 Amdahl's law

If we imagine ourselves as a team that would like to migrate from a single-processor program to a multi-processor program, it would be perfect to be sure that if we embark on the parallelisation of such a system, it will pay off. Moreover, many people live in a bubble, where they think 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 home table. In this case, it is a sequential approach. However, adding four identical tables (so there will be five) will take five times more time for one. Suppose four friends come to help him (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 will 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. This implies that the acceleration will not be close to 5-times, but it will probably be almost 3-times. This kind of analysis is crucial for concurrent computation, and thanks to Mr Amdahl, 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 what the program is correct. 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

The first aspect is 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.

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

[1]**Deadlock** – is a situation where two processes or threads enters a waiting state because a requested system resource is held by another waiting process, which in turn is waiting for another resource held by another waiting process (toto prepisat)

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. While at Non-uniform memory access, the time will be different. In the UMA 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 object, which has its own global address space.*

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. 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.
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 returns to the *Terminate* state if it was previously aborted abnormally (i.e., using inter-process communication) or complete its execution.

[2] **Cache coherence** – this is a situation where one of the processors obtains a value from shared memory and makes a change in its cache memory and 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.

3.4 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 are changing it.*

Fortunately, as programmers, we can eliminate such errors. 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 the 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 briefly stated 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 lock whenever one wants to modify a thread or read from a shared resource. Then one modifies the source and finally release the lock. Acquire lock is an Atomic operation performed as 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. One 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** – In case more readers want to read from a shared resource can. 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 read the value subsequently 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 the wait queue. to 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, and in the next paragraph, we argue why.

ThreadPool is an object that creates and manages several threads, also called worker threads. What is so interesting about *ThreadPool* is that 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 entire load on resources. However, this is nice, but if we want to submit one asynchronous task, then in the main thread, we want the future result. Thus, we created 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 closing *Future*. It also provides access to asynchronous operations, so most languages 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 an 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, 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* [17] or *The Art of Multiprocessor Programming* [9].

Chapter 4

Proposal of parallel approach

TODO: popis kapitoli...a uvod co bude zahrnovat...

4.1 Bottlenecks of current approach

If we remember the knowledge we acquired in Section 2.4, then one realises that the time required for a given test set is extremely time-consuming. It is easier to maintain the correctness of the program of the sequence computing model, but the benefit that parallelism offers is incomparable. Nevertheless, one has to ask oneself whether it is possible and whether it pays off. 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 by analysing whether it contains any shared variable against other tests. 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 such a test will have to run in an isolated environment. The manual analysis we performed found that 250 tests must be 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, and if we scale it to 16 CPUs, the acceleration will be almost 3x. 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 the non-use of 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 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 must be shared for all test suites. Using this approach eliminates a lot of 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 or the entire test suite to run in isolation.

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

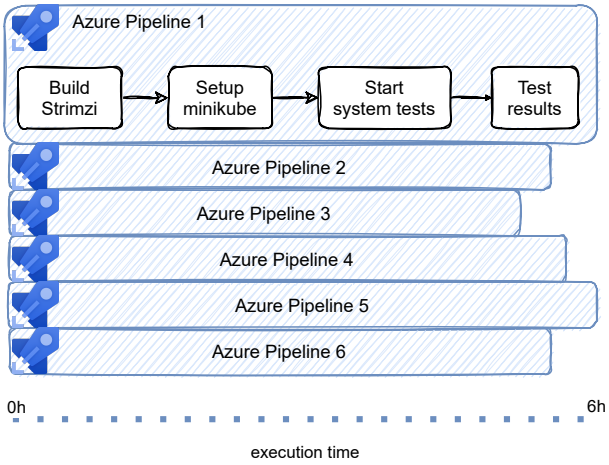


Figure 4.1: Azure pipelines in form parallelism used to execute our system tests

to create a resource, we do it using *KafkaResource.kafkaEphemeral(...).done()* 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 separation pipelines [1]. In Figure 4.1 one can see such decomposition. The attentive reader might realise why we cannot run such 40 or 100 Azure pipelines and thus eliminate the total execution time of the tests. Unfortunately, we are limited only to run six Azure Pipelines simultaneously. By this limitation, the total regression of the tests takes approximately 6 hours, which is not very satisfactory. Similarly, we try to reduce the time at the Jenkins pipeline when using the OpenStack and Amazon Web Services infrastructure. However, this Strimzi product must be verified for multiple configurations when running a Kubernetes separation 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.

What is also a duty to mention is that we are limited to the number of processes (i.e., pipelines) that always use the separation Kubernetes cluster. On Amazon Web Services and Openstack infrastructures, we are not limited to the computing resources we use. This is a fact that we must use and thus think about how parallelisation will lead the

[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.

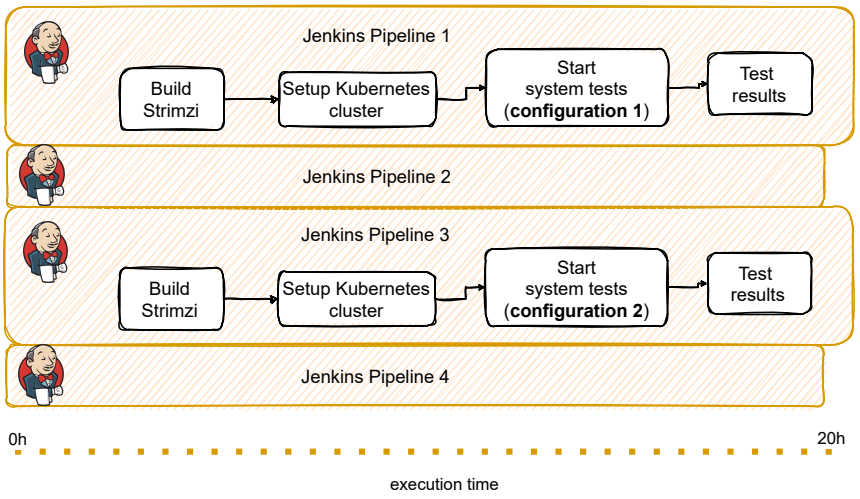


Figure 4.2: Jenkins pipelines in a form parallelism used to execute our system tests

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 infrastructures' 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

4.2.2 Writing own Junit5 Engine

4.2.3 JUnit5 paralelization

4.3 Architecture changes

4.4 Method wide paralelization

4.5 Class wide paralelization

4.6 Algorithms

Algorithm 4 Parallel algorithm for creation all resources inside *Resource manager*

Input: extensionContext, resources

1: **for each** $resource \in resources$ **do**
2: $type \leftarrow findResourceType(resource)$
3: $type.create(resource)$
4:
5: // here starts critical section
6: $all\_resources.computeIfAbsent((test\_name), k- > newStack <> ())$
7: $all\_resources.get((test\_name).push(deleteResource(resource)$
8: // here ends critical section
9:
10: **if** wait for resource readiness **then**
11: **for each** $resource \in resources$ **do**
12: $type \leftarrow findResourceType(resource)$
13: wait for resource readiness
14: **end for each**
15: **end for each**

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

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

[11] 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.

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

[13] 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.

[14] 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/.

[15] 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/.

[16] 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/.

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

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

[19] 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/.

[20] 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/.

[21] 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/.

[22] 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 [19], 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 Raggedhttps://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 [20].

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 [10], 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 signifficantly larger, nor signifficantly 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:

$$\cos^3\theta = \frac{1}{4}\cos\theta + \frac{3}{4}\cos 3\theta \tag{D.1}$$

and two horizontally aligned equations:

$$3x = 6y + 12 \tag{D.2}$$
$$x = 2y + 4 \tag{D.3}$$

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`:

$$\begin{aligned} 3x &= 6y + 12 \\ x &= 2y + 4 \end{aligned} \tag{D.4}$$

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 [19]. 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. 63),
- monographic publication (str. 64),
- conference proceedings (str. 65),
- conference proceedings entry or book chapter (str. 66),
- manual, documentation, technical report and unpublished materials (str. 67),
- academic work (str. 68),
- web page (str. 69),
- and web site (str. 70).

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.