FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Distributed job execution and evaluation system
Student:	Bc. Jan Mašek
Supervisor:	Ing. Tomáš Bartoň
Study Programme:	Informatics
Study Branch:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2020/21

Instructions

Design and implement a distributed system for local job execution and subsequent analysis of batch job results. The system should be able to run a job on a computing cluster with requested resources, display its status, and process and show its output in real-time. This output should be persistent for analysis purposes. Jobs are expected to be compute intensive with possibility to write on a scratch drive. Choose appropriate open-source technologies suitable for such tasks and justify your selection. Analyse potential security risks when running 3rd party (malicious) code. Demonstrate capabilities of the proposed system on assignments from the MI-PAA (Problems and Algorithms) course in at least two programming languages of your choice (e.g., Python, C++) and document your work.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague March 14, 2019

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Distributed execution and evaluation system

Bc. Jan Mašek

Department of Computer Systems
Supervisor: Ing. Tomáš Bartoň

May 9, 2019

Acknowledgements

I want to thank my supervisor Ing. Tomáš Bartoň for the enormous amount of help and his patience. I want to thank my family for their incredible support.

Declaration

In Prague on May 9, 2019 ....

Citation of this thesis

Mašek, Jan. *Distributed execution and evaluation system.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019. Also available from: ⟨`https://gitlab.com/mashtak/ctu-thesis`⟩.

Abstrakt

DJEES je distribuovaný systém pro spouštění a vyhodnocování úloh. Při návrhu a implemntaci byli použity nejnovější technolgie. Hlavní částí systému je Mesos framework, který zajišťuje přidělování volných prostředků ve výpočetním clusteru. Výpočty jsou izolovány za použití Docker kontejnerů. Výstupy výpočtu jsou následně uloženy v databázi Cassandra. Systém kombinuje distribuované technologie tak, aby tvořili škálovatelné a odolné prostředí, kde může uživatel spustit svůj výpočet.

Klíčová slova Distribuované spouštění výpočtů, Mesos, Docker, Fluentd, Kafka, Cassandra, Java

Abstract

DJEES is a distributed system for job execution and evaluation. It was implemented using the latest technologies. The core of the system is a Mesos framework that manages the cluster resources. The jobs are isolated using Docker containers, and output is persisted into the Cassandra database. The system combines distributed fault-tolerant technologies to provide a scalable, fault-tolerant environment where users can run their jobs.

Keywords Distributed job execution, Mesos, Docker, Fluentd, Kafka, Cassandra, Java

Contents

List of Figures

Introduction

This thesis focuses on building distributed and scalable system using the latest technologies. The purpose of this system is to provide students and other users a versatile platform for running diverse jobs. It does not matter if the user needs to run a compute-intensive, long-running job, or another type of job, running such a job should be easy and should have minimal complexity. However, running a big number of diverse workloads is a big challenge in the matter of resources assignment, handling the dependencies, isolating jobs and monitoring them.

First, with the development of hardware virtualization, it was possible to divide the server into multiple virtual machines. However, hardware virtualization is a bit heavy-weight tool. In the virtual machine, there is a whole operating system running; the user has to take care of all the components of the operating system and the whole running environment of the application. Moreover, running a whole operating system has significant overhead. It means that multiple applications have to be run on the same virtual machine to minimize the overhead of running operating system. Running multiple application on the same virtual machine brings again challenges with resources and applications isolation. These factors increase the complexity of running applications in a virtual machine. However, with massive adoption of the containerization, we can see a reduction of the complexity of starting, running and migrating applications. With containers, the application can be packed together with its dependencies and run in an isolated environment. Moreover, the containers are light-weight - contains only dependencies, not a whole operating system, and are very easy and fast to run.

Proposed system benefits from the latest development of containerization and resource sharing, and provides users unified interface for running their tasks and utilizing available resources.

Chapter 1

Analysis and design

This thesis aim is to design and implement a system for job execution and subsequent analysis of the results. It is meant to be a universal computing environment where can any student submit a job and use the available resources.

1.1 Requirements

The analysis is divided into two areas: user requirements and system requirements. The user is the main element that decides whether the project adopted and considered as successful, or not. On the other side, there is a system maintainer. The system maintainer requires to keep operations costs low.

All the gathered requirements are further categorized as functional and non-functional.

1.1.1 User Requirements

Currently, the main goal is to build a system where jobs could be evaluated for the course MI-PAA (Problems and Algorithms). The jobs are considered to be compute-intensive, and a user can choose the programming language which is going to use to implement the solution. Therefore, the system should be language independent. Moreover, the system can be used for other courses and types of jobs with various resource requirements; therefore, the user should be able to specify the required resources per job.

It can be expected that there are going to be multiple jobs running on the same nodes. Jobs can have various dependencies that can be mutually exclusive; therefore the job isolation is required.

When a job is submitted the user wants to know if the job is running and what is its output. Then, when the job terminates, the user needs the job output for further analysis.

1.1.2 System Requirements

For this thesis, four servers were used; however, no assumption cannot be made about the hardware that is going to be used for the system in the future. Moreover, the load and the usage of this system can not be predicted; therefore, the system should be distributed, easy to scale and fault tolerant.

The system is going to run the user's code and commands. Running user's code brings security risks. It is impossible to check if the code is potentially dangerous; therefore the system should maximize the job isolation and minimize the system exposition to the jobs.

1.1.3 Functional requirements

The functional requirements are the requirements that specify what functionality should system offer.

- Submit the job
- Run job with requested resources
- Run language independent job
- Display job status
- Show output in real-time
- Persistent output

1.1.4 Non-functional requirements

The functional requirements characterize the system. They state the properties that system should have.

- Distributed system
- Scalability
- Fault-tolerance
- Job isolation

1.2 Architecture design

The core of the system is a resource manager. It gathers information about all the nodes in the system, manages the resource allocations and schedules the jobs on a specific node. The resource manager is usually tightly coupled with the job executor who runs the code on the specified node. Therefore, the resource manager should support the chosen run method or at least provide an interface for implementing required job executor.

The proper resource manager, together with job executor, must be able to facilitate the following requirements.

- Submit the job
- Run job with requested resources
- Display job status

Moreover, to fulfill the non-functional requirements, they should support running in distributed mode and be easy to scale and fault-tolerant.

The non-functional requirement *Job isolation* requires a particular focus on how the jobs are run in the system.

The jobs cannot be run directly on the node because they would be hard to isolate; therefore, virtualization could solve the problem with job isolation. Moreover, it allows isolating not only the job but also the dependencies. There are two types of virtualization, hardware-level virtualization, and operating-system-level virtualization.

Hardware virtualization provides virtualization of the underlying hardware resources. It brings a reasonable level of security because virtualized systems share only the hardware that is isolated by the virtualization platform. On the other hand, running a job in a virtualized system means starting the whole system including the operating system and running it which can be significant time and resources consuming.

Operating-system-level virtualization, also known as containerization, virtualizes kernel space and isolates the container in user space. The great advantage of this approach is that it is lightweight. The container contains only the job, and it's dependencies because it shares the rest with the host. The most significant disadvantage of the containerization is that it is difficult to fully isolate the container and prevent it from accessing the host machine and other processes. However, containerization seems to be a good compromise between complexity and job isolation.

Containerization is also going to allow fulfilling the requirement *Run language independent job* because each container can have different dependencies and libraries attached.

The requirement *Show output in real-time* means that there has to be pipeline taking the job's output that is running inside the container and deliver it to the user. However, the requirement *Persistent output* defines that the output has to be persisted to persistent storage. These two requirements can be solved independently, one pipeline for the showing output in real-time and one for persisting the output. Alternatively, they can be solved together, one pipeline that stores the output to the storage and the user queries storage that provides real-time data. The first approach does not rely on storage that means the output is not delayed by storing and loading data to and from storage. However, in case of failure, there can be an inconsistency between real-time data and persisted data. Moreover, two pipelines add complexity to the system; therefore the system is going to use one pipeline that first saves the output to the storage. The output is going to be served to user from storage.

Storing data to storage can be complex operations that can be time and resources consuming. Therefore, it is important to decouple the outputs extracting and storage. The message queue suits well for solving this problem.

Then, the data from the message queue are consumed by a consumer that stores the data to storage. A stream processor can solve this problem.

Since no predictions cannot be made about the output of jobs and about the number of jobs that are going to be run on the system, the pipeline should be designed as high-throughput.

To summarize the previous analysis, the resource manager should support the executor that can use containerized jobs. The container platform should support log forwarding. The message queue and the storage should be designed as high-throughput, fault-tolerant, distributed and scalable.

Figure 1.1 shows the basic overview of proposed system.

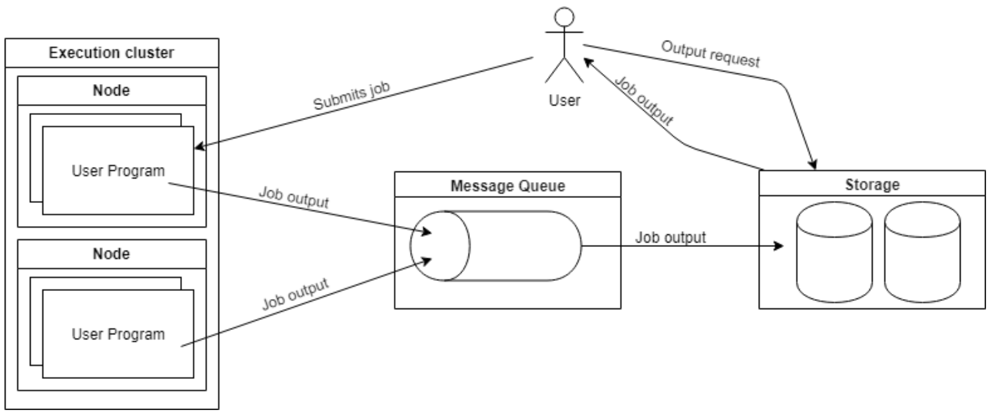

Figure 1.1: Design of the system

1.3 Methodology for evaluating project

In the previous text, the requirements were defined. However, there is a huge amount of technologies and projects with various quality. The criteria for choosing the right component of the system and better orientation in projects are defined.

1.3.1 Criteria

When building a new system, choosing the right projects is an essential part of the design. Many technical requirements have to be met (performance, compatibility, etc). However, there are factors that are not visible on the first look that can significantly influence the complexity of building and running system. These factors are closely related to people who develop the project, who run the system, or simply people who have experience with the component and who are open to share the experience and improve the project.

1. License
2. Developers' activity
3. Ecosystem
4. Production usages
5. Community
6. Documentation

1.3.1.1 License

The project license is one of the most important aspects in choosing the right component. The right license provides the certainty that the project can be evolving freely in the future and that it can be adopted by other community if it is needed. Moreover, there are sometimes functions that are not properly documented or are not working as expected. With the open-source license, the developer can take a look into the code and determine how the function works and how to use it. The preferred license is one of the open-source licenses. There are various licenses with different formulations and conditions. The main institution that is focused on interpreting open source licenses is the Open Source Initiative (`https://opensource.org`). The main open-source licenses are following.

- Apache License 2.0
- MIT license
- Mozilla Public License 2.0
- GNU General Public License (GPL)
- BSD License

1.3.1.2 Developers' activity

The repository activity can provide a good insight into the project's community and background. It is necessary to choose the project that is alive, that has a community which is developing the project and is fixing the issues and security problems. The project repository is the right place to determine the project state. The main metrics are the number of contributors, commits frequency and recent activity.

1.3.1.3 Ecosystem

This criterion is difficult to measure, but the main purpose is to choose projects with a strong and wide ecosystem that provides good versatility and extensibility. The good ecosystem does not provide only the project but also the tools for monitoring, configuration and connecting to other components. These tools reduce the complexity of starting and running the project.

1.3.1.4 Production deployments

Although the exact number is impossible to obtain, a rough estimate of the production usages is a good indicator of the project maturity. Usually, it is sufficient to determine if there are zero, few or many companies that run the project in a production environment. Running the project in a production environment means that there are people who believe in the project and are willing to spend time and money on running it and on building their business around it. Moreover, production usage guarantees that there are people that need to keep the project up-to-date and there is a high probability that they are going to share their experience and push the project forward.

1.3.1.5 Community

The community and the content made by the community greatly help and simplify project adoption and system development. If there is a problem, there is a high probability that somebody has already been solving it or that the community could help with it. There are several types of content. It can be discussions on the internet, presentations from conferences, additional tools that simplify the work with a project or open source projects built on top of the project.

1.3.1.6 Documentation

Although the source code can greatly help to understand the functionality, the key component of the project is documentation. It allows the developer to grasp the project architecture, reduces the project integration complexity and speeds up solving problems.

However, the quality of the documentation is hard to determine until the developer start using the project and start solving specific problems.

Chapter 2

State of Art

This chapter focuses on describing current technologies and project that are available and could help create the system. These projects are described in various depth. The investigation of the projects is performed into the depth that is necessary for choosing a suitable project.

2.1 Resource Manager

A resource manager is the core of the system. It monitors all the nodes and their resources. When there is a new job, the resource manager matches the requested resource with the job resources and send the job to the node.

2.1.1 Mesos

Apache Mesos is an open-source project that was developed at the University of California, Berkeley. It was published in [1]. "It is a platform for sharing commodity clusters between multiple diverse cluster computing frameworks"[1]. Frameworks are programs that Mesos offers resources, and that can accept them and run their job on the node. These frameworks have to communicate with Mesos using Mesos API.

Mesos consists of two components: a scheduler and an executor. Mesos uses resource offers to assign resources. When a node has available resources, it notifies the master with the allocation module. The allocation module, according to scheduling policy and resource request, creates the resource offer and send it to the suppliant (Framework) that can decide whether it uses the resources or not. If it accepts the resources, it sends the task to the master that will forward the task to the executor. The figure 2.1 shows the communication.

Mesos utilizes ZooKeeper to make the master with Allocation module fault-tolerant. Mesos natively supports Docker and default Mesos containers.

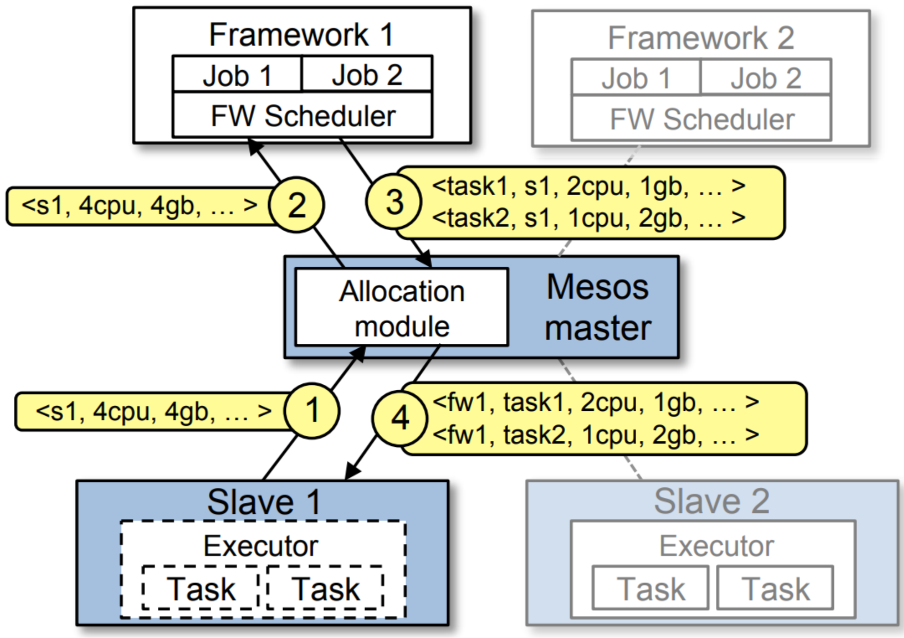

Figure 2.1: Mesos resource offer and offer acceptance [1]

Mesos was massively commercialized because the Mesosphere built a DC/OS (the Distributed Cloud Operating System) on it. DC/OS is an operating system that incorporates many project and tools to provide a versatile environment to users DC/OS is open-source; however, some parts are available only for paying customers. There is a huge community around Mesos and DC/OS. The developers are very active and many companies run Mesos or DC/OS in the production environment.

Around Mesos, there is a huge ecosystem of the tools and frameworks that can be used out-of-box. Documentation of the Mesos and the tools in the ecosystem varies from repository to repository. For example, Mesos has very good and exhaustive documentation, Marathon as well; however, Metronome has very brief documentation. Marathon and Metronome are Mesos frameworks.

Currently, there are more than 50 companies that use Mesos in production environment [12].

2.1.2 Yarn

The purpose of YARN (Yet Another Resource Negotiator) was to decouple the programming model from the resource management, and delegates scheduling functions in Hadoop's compute platform [2].

Yarn allows running other applications on Hadoop cluster, for example, Dryad, Giraph, Hoza, Spark, etc. In Yarn, jobs are submitted via a public interface to Resource Manager. Job specific logical plan of execution, requesting resources, generating physical resources and coordinating the execution of the plan is delegated to an Application Master. On each node, there is a daemon called the Node Manager. Resource Manager dynamically allocates the resource of the Node Managers and provide them to the Application Master. Then the job is run on allocated nodes. Figure 2.2 shows the Yarn architecture. In 2013, Yarn was used in a production environment in Yahoo!, running

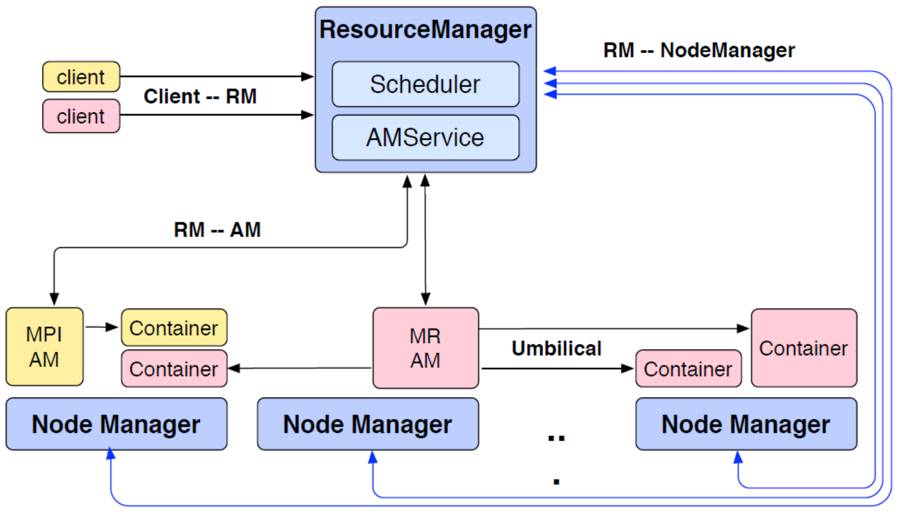

Figure 2.2: YARN Architecture with two application running [2]

on 2500 node grid [2].

The YARN is resource manager designed mainly for the Hadoop environment. It means it works great with MapReduce and HDFS; however, it is less versatile.

2.1.3 Kubernetes

The Kubernetes evolved from Google's internal systems Borg and Omega, and Google released it as open-source. All three systems shared the same goal - effective machines sharing, increasing resource utilization and thereby reducing costs [13].

Kubernetes was developed with a strong focus on the experience of developers. Its goal is to provide an easy way to deploy and manage a complex distributed system. The core of the Kubernetes is a shared persistent store watching for changes to relevant objects that are accessed through a domain-specific REST-API [13].

The Kubernetes cluster consists of nodes (VM or physical machine). Each node contains pods. A pod is a group of one or more containers (e.g., Docker containers) that share storage and network (an IP address and port space). A container is an isolated unit with a limited interface which runs container image. A container image is a package that consists of the application and its dependencies.

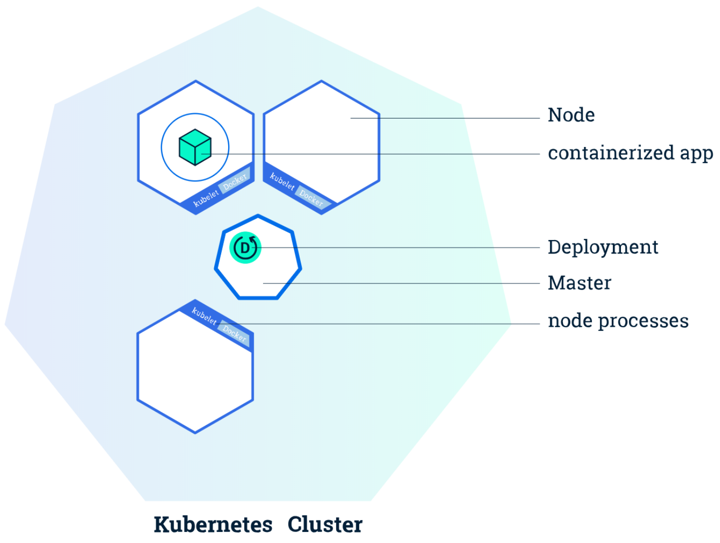

Figure 2.3: Kubernetes cluster diagram. Taken from `kubernetes.io`

Kubernetes is primary a container orchestration tool; it means it is mainly focused on running containers.

Kubernetes is a very popular project with a live community, a big ongoing development and many contributors (more than 2000 in the main repository). The documentation is on a good level, and more than 900 companies use it in production environment [14].

2.1.4 PBS/Torque

Torque (The Terascale Open-source Resource and QUEue Manager) is a fork of OpenPBS and currently maintained by Adaptive Computing.

"A TORQUE cluster consists of one head node, and many compute nodes. The head node runs the pbs_server daemon, and the compute nodes run the pbs_mom daemon. Client commands for submitting and managing jobs can be installed on any host (including hosts not running pbs_server or pbs_mom)." [15]

"The head node also runs a scheduler daemon. The scheduler interacts with pbs_server to make local policy decisions for resource usage and allocate

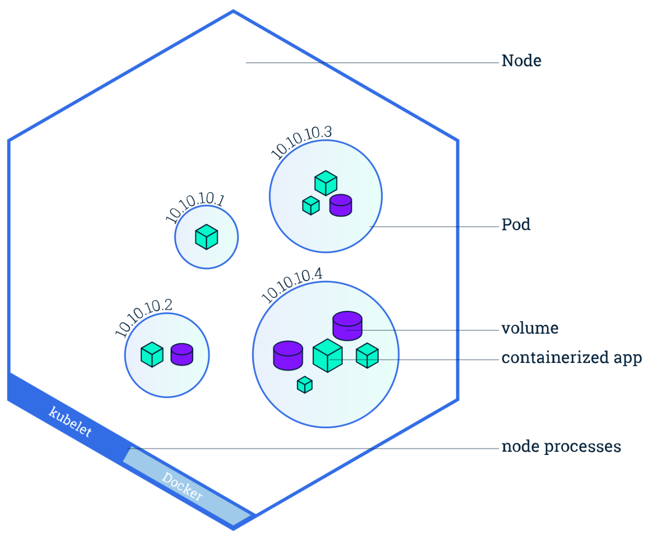

Figure 2.4: Kubernetes node diagram. Taken from `kubernetes.io`

nodes to jobs. A simple FIFO scheduler and code to construct more advanced schedulers is provided in the TORQUE source distribution. Most TORQUE users choose to use a packaged, advanced scheduler such as Maui or Moab."[15]

The project documentation looks good. Unfortunately, there is very little development in the repository. In the year 2018, there were only two commits made. Moreover, it has only 37 contributors (compared to Mesos that has 294 contributors).

2.1.5 Nomad

Nomad is a resource manager and scheduler. Nomad was inspired by Google Borg and Google Omega (Kubernetes' ancestors), and it is designed as distributed, scalable and highly available.

Nomad is a general purpose; therefore, it can run Docker (containerized applications), standalone, batch and scheduled applications. It uses Consul (consistent distributed key-value storage) for service discovery.

In Nomad, there are two types of entities – Client and Server. A client is a machine that tasks can be run on. Servers are responsible for accepting the jobs, managing clients and assigning tasks. In each region, there is a leader server which is elected using leader election.

Nomad's repository has a similar activity and contributors amount as Mesos. Documentation is on a good level. However, Nomad is architecturally much simpler. It does not depend on other systems (like Zookeeper) and does not need scheduling framework (like Marathon or Metronome for Mesos)[16].

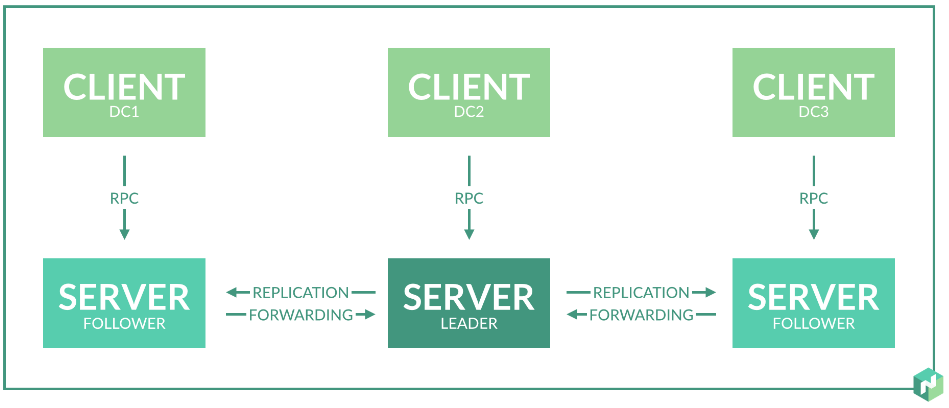

Figure 2.5: Nomad architecture [3]

[17] suggest that there are at least 20 companies that use Nomad; therefore it can be considered as a mature project.

2.2 Containerization

Although Docker can be run on Windows too, this text is focused on running containers in Linux environment.

Containerization (also called OS-level virtualization) is a technique that isolates application in user space. The main operating system primitives for isolation are namespaces and cgroups.

Namespaces allow process isolation in terms of system resources, networking, and file system. It means that an isolated process does see nor interfere with other running processes in the system. CGroups provide a way to limit resources usage as CPU and memory.

However, when a new container is started there are many complex operations that can be parametrized. Therefore, containers runtimes were created. The purpose of runtimes is to provide a certain level of abstraction to developers and reduce the complexity of a running container.

[18] suggests dividing runtimes on Low-level and High-level depending on their capabilities and the level of abstraction. However, the problem of these categories is that they cannot be exactly defined because different runtimes implement different capabilities. The main idea is that Low-level runtimes take care of creating namespace and cgroups, mounting file system and, generally, running the container. The High-level runtimes take care about transport and management of container images and delegates complex operation to Low-level Runtime. Figure 2.6 shows the diagram of the components that run a container.

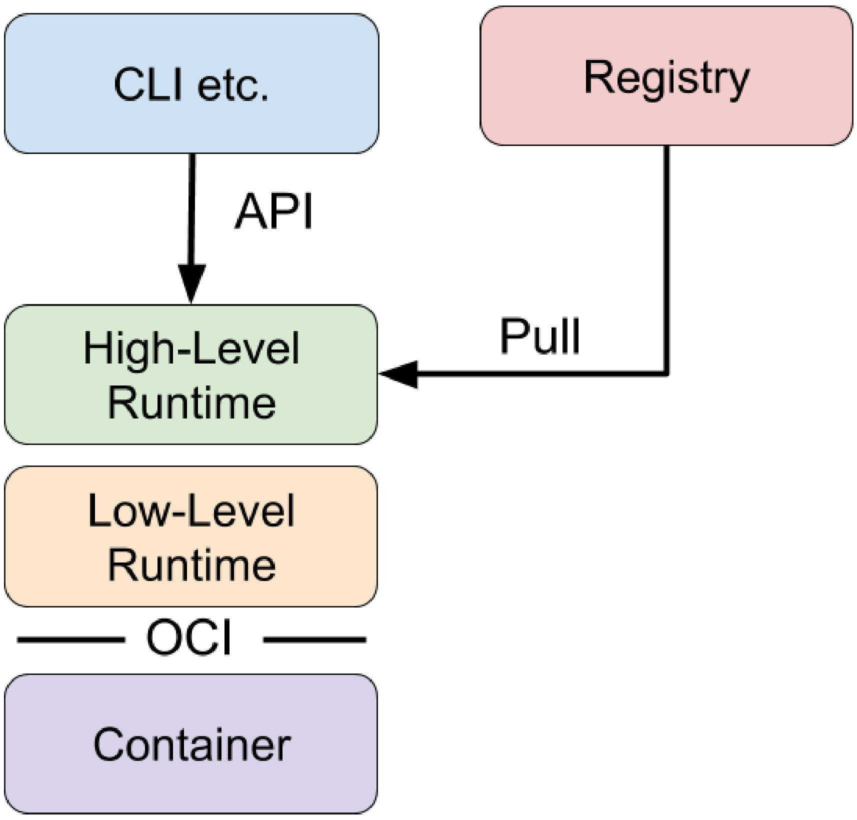

Figure 2.6: Container runtimes [4]

The leading organization that tries to standardize containers is the Open Container Initiative(OCI). Containerization is a popular technology and is developing fast. The following text provides a basic overview of the most popular container runtimes.

2.2.1 runc

Runc is one of the most popular Low-level runtimes. It expects a user to understand the low-level primitives; therefore it is mainly used by High-level runtimes. It was originally developed as part of Docker and it implements the Open Container Initiative (OCI) Runtime Specification.

2.2.2 rkt

RKT is alternative to runc developed by CoreOS.It provides low-level functions but also some high-level features. It allows building and downloading the image as well as running it.

2.2.3 systemd-nspawn

The name systemd-nspawn is an abbreviation of namespaces spawn. It means that only manages isolated processes; therefore, it cannot isolate resources. It does not provide an image repository.

2.2.4 containerd

Containerd is a runtime that was initially part of Docker too. It prepares the image and hand-over it to runc to run it.

2.2.5 Docker

Docker is currently on the most popular containerization technology. It provides an end-to-end user experience. It is composed of dockerd daemon, docker-containerd, and docker-runc. Docker-containerd and docker-runc are just packaged versions of containerd and runc [4].

2.2.6 Container Runtime Interface

Container Runtime Interface (CRI) is an interface defined by Kubernetes to provide interchangeability of container runtimes. Kubernetes is a popular open-source container orchestrator. The main High-level runtimes that implement CRI are containerd, Docker and cri-o.

2.2.7 CRI-O

CRI-O is a lightweight CRI runtime made as a Kubernetes specific high-level runtime. It supports the management of OCI compatible images and pulls from any OCI compatible image registry. It supports runc and Clear Containers as low-level runtimes [19].

2.2.8 LXC\LXD

LXC is a system container runtime designed to execute full system containers, which generally consist of a full operating system image. An LXC process, in most common use cases, will boot a full Linux distribution such as Debian, Fedora, Arch, etc., and a user will interact with it similarly to how they would with a Virtual Machine image [20].

LXD is based on LXC; however, it provides additional capabilities. It exposes REST API that can be used for managing containers.

2.2.9 gVisor

gVisor is a project by Google. It works by emulating the Linux kernel in userspace. This means that any syscall that is called by the container process is proxied through gVisor which then does the necessary work. gVisor stops the container process from directly communicating with the host kernel [21]

2.2.10 Kata Containers

Kata Containers takes a very different approach to container isolation. Instead of relying on the standard namespaces, cgroups and capabilities combination to isolate the container process, Kata runs each container in a stripped down QEMU virtual machine using the KVM hypervisor [21].

2.3 Message Queue

Message queues enable decoupling message architecture and provide scalability. Publisher does not have to be aware of subscribers, they do not have to send/receive at the same; they do not have to even run at the same time. In the system, a message queue is used to decouple Job executor producing output and the consumer that saves the output to the database.

2.3.1 Kafka

Apache Kafka was designed as a scalable publish-subscribe messaging system. It was first developed by LinkedIn to solve a general problem of delivering extreme high volume event data to diverse subscribers [22].

Nowadays, it is a distributed streaming platform. It means that it combines a queuing system and a publish-subscribe system.

Kafka organizes messages as a partitioned commit log on persistent storage so both real-time subscribers (e.g., online services) and offline subscribers (e.g., Hadoop) can read the messages at arbitrary pace [22].

In Kafka, the category or feed name to which record is published is called topic. The topic is Multi-subscriber (zero, one or many consumers) and can be partitioned. A partition is an ordered, immutable sequence of records. The record consists of a key, a value, and a timestamp. Each record in the partition has a sequential id number called offset. All records are persisted for the configurable retention period. After the retention period, the records are discarded.

A partition is the main unit of scaling and distribution. The topic can be too big to fit on a single server so it can be divided into multiple partitions that can be hosted on different servers. Moreover, a partition can be replicated across a configurable number of servers. Each partition has one ”leader” and followers that can take over in case of leader failure.

Producers are responsible for choosing a topic and partition. They can use a round-robin way to balance load or use other technique. Consumers can be assigned to consumer groups. Then each published record is delivered to one instance of each subscribing consumer group.

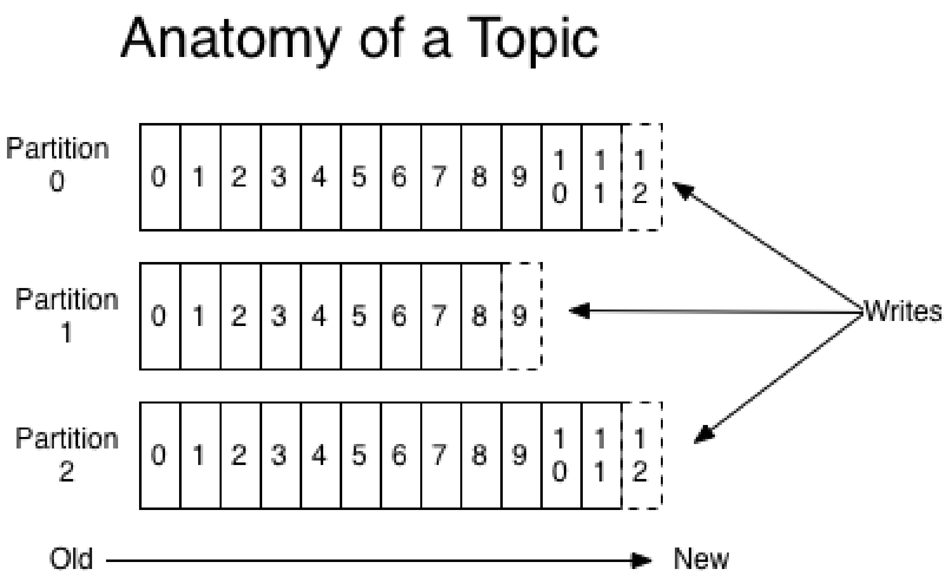

Figure 2.7: Topic's structure [5]

Kafka gives the following high-level guarantees [5].

- If the two messages are sent from the same producer to the same topic partition, the first sent message is going to have lower record offset than the later one.
- A consumer sees the record in the order they are stored into the log.
- If the replication factor is N, the system is going to tolerate N-1 failures without data loss.

The main Kafka's design goal is to handle high throughput (millions of messages).

Kafka is a widely used the popular messaging system. It has a big community of users and developers and good documentation.

2.3.2 Rabbit MQ

RabitMQ is an efficient and scalable implementation of the Advanced Message Queuing Protocol (AMQP) [23]. AMQP is a specification that defines the semantics of an interoperable messaging protocol. The Advanced Message Queuing Protocol (AMQP) is an open standard that defines a protocol for systems to exchange messages. AMQP defines the interaction that happens between a consumer/producer and a broker, and also the representation of the

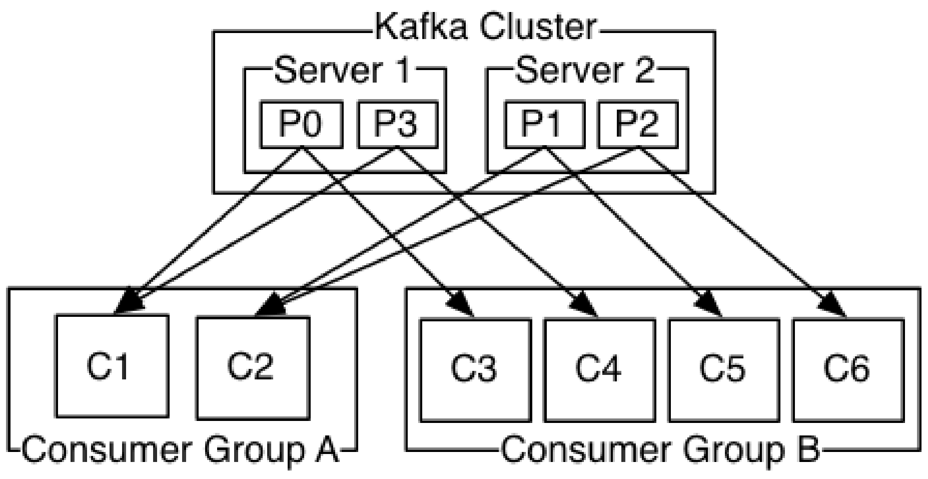

Figure 2.8: Consumer groups in Kafka [5]

messages and commands that are being exchanged. Because of the message format specification, it is truly interoperable, and there are implementations in a wide range of languages [6].

The core concepts of AMQP are following: [6]

- *Broker* Receives messages from publishers and deliver them to consumers or to another broker.
- *Virtual host* Virtual division in a broker, allowing segregation.
- *Connection* Network connection (TCP)
- *Channel* Logical connection. There can be multiple channels in one connection.
- *Exchange* Initial destination for all messages. It applies routing rules (e.g. direct(point-to-point), topic (publish-subscribe) and fanout (multicast))
- *Queue* Place where the messages wait to be consumed.
- *Binding* Virtual connection beween exchange and queue.

RabbitMQ is optimized for empty-or-nearly-empty queues, and the performance degrades significantly if the messages get accumulates. The reason is that the main design goal is to handle messages in DRAM memory [23].

RabbitMQ provides many exchange types which can be used to create complex routing logic.

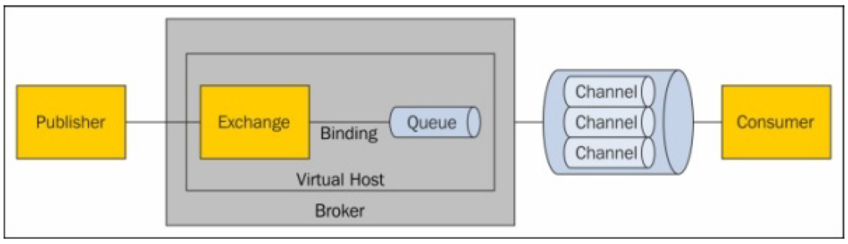

Figure 2.9: AMPQ architecture [6]

Since there can be run various jobs with various size of the output in the, it is important that the Message Queue can work effectively if there is a long queue of messages. Therefore, RabbitMQ is not well suited for the system.

2.3.2.1 Zero MQ

ZeroMQ is a library. It runs on different architectures and has support for more than 20 programming languages [24].

ZeroMQ is different from other message queuing systems because it does not need a broker - the central message server that provides asynchronous communication. If the application wants to send the message asynchronously, it can simply delegate it to ZeroMQ library that will queue the message in an asynchronous I/O thread [24].

ZeroMQ is an interesting technology; however, it is not well suited for the system. It is a library that can be used to build brokers and protocols. However, log forwarders are simple, light-weight programs that are designed to forward the message and move on.

2.3.3 Active MQ

ActiveMQ is open source messaging and message-oriented middleware.

Multiple ActiveMQ brokers can coordinate with each other to work as a single entity so that it is capable of providing expandability to meet requirements of large-scale systems. Unlike Apache Kafka, ActiveMQ does not allow multiple consumers to poll messages from a queue in parallel. It is because of ActiveMQ's centralized architecture which can potentially limit the queuing performance [7].

ActiveMQ is not suitable for the system because it does not allow consumer scaling that is important for the system.

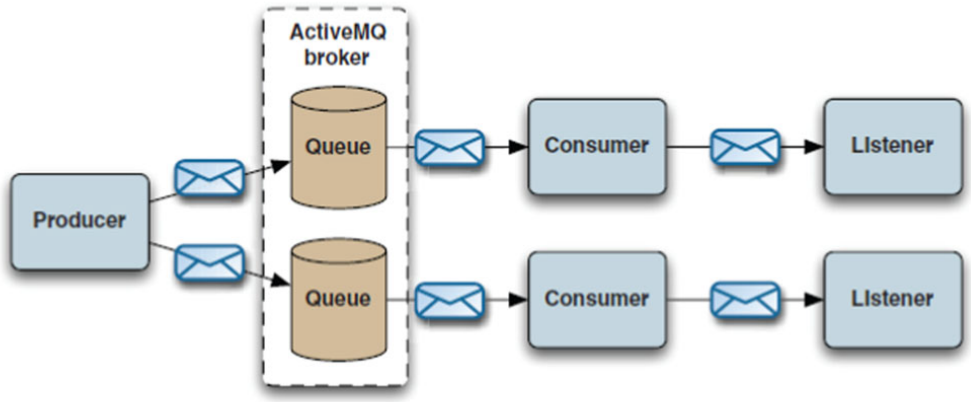

Figure 2.10: ActiveMQ architecture [7]

2.3.4 Apache Pulsar

Pulsar is a multi-tenant, high-performance solution for server-to-server messaging. Pulsar was originally developed by Yahoo, it is under the stewardship of the Apache Software Foundation[25].

Pulsar combines streaming and queuing into unified messaging model and API. It means it can be used in the system where both streaming and queuing are needed.

Pulsar offers following consuming modes.

- *Exclusive subscription* Only one subscriber can be subscribed to subscription.
- *Failover subscriptions* Multiple subscribers can be subscribed to the subscription; however only one consumes the messages and others will consume in case of the consuming subscriber.
- *Shared subscriptions* Multiple subscribers can be attached to the same subscription. Messages are delivered in a round-robin distribution.

A Pulsar instance consists of at least one Pulsar cluster. Clusters can replicate the data among themselves. A cluster consists of at least one Pulsar broker, Zookeeper and set of bookies. Pulsar broker exposes REST API for administrative tasks and topic lookup and dispatcher. The dispatcher is an asynchronous TCP server used for all data transfers. Bookies are instances of Apache BookKeeper. Apache BookKeeper is a distributed write-ahead log that is in Pulsar used as persistent message storage. Figure 2.11 shows the architecture of Pulsar.

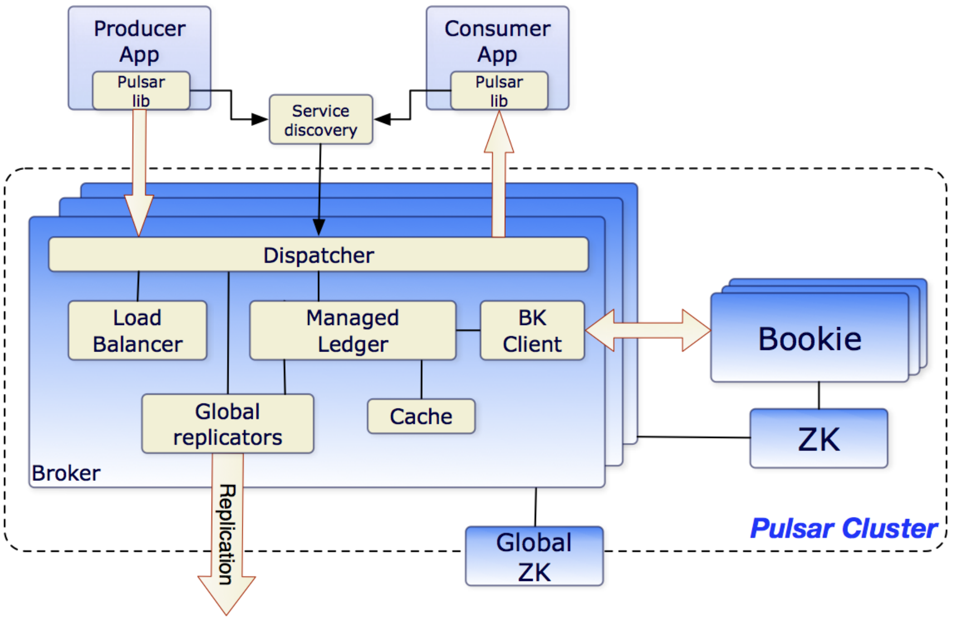

Figure 2.11: Apache Pulsar architecture. Taken from `https://pulsar.apache.org/docs/`

2.4 Stream Processor

The Stream processor was originally defined as a possible solution of consuming messages from Message Queue. However, adding Stream processor increases the operational complexity and can be solved by simple consumer application. The Stream processors that were considered are following.

- Spark
- Storm
- Flink

2.5 Storage

In general, data storage can be divided into File system storage and database. File system storage is not suitable for the system because the system is not only going to save the job's output but also the information about the tasks.

The following properties can classify databases.

- Consistency
- Availability

- Partition-tolerance

In 2000, these properties were specified by Brewer [26]. They can be used to characterize the type of database because all three properties cannot be fulfilled at the same time.

Consistency means that all clients see the same data at the same time. It means that by updating data on one node, the data on all nodes are updated.

Availability means that the client can read and write any time regardless of the state of the system.

Partition tolerance is the property that guarantees that the database works despite message loss or partition failure.

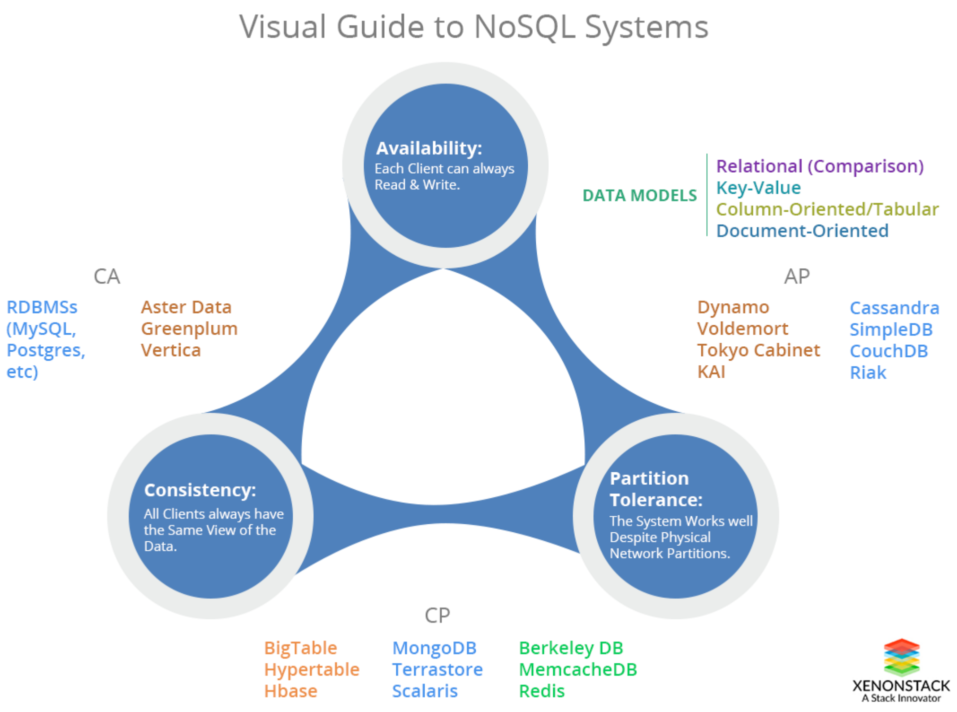

Figure 2.12: Database properties with examples [8]

In the previous text, the following requirement for the database were defined.

- High-throughput
- Fault-tolerant
- Distributed
- Scalable

Considering the CAP theorem, Partition-tolerance is important in order to fulfill the requirement on Fault-tolerance. High-throughput requirement shows the need for Availability. However, the high-throughput is necessary only for writing to the database. For the reading, Consistency is more important (to provide user consistent up-to-date results). Therefore, the ideal database should be CP or AP with the possibility of tuning Availability a Consistency.

In addition to the CAP theorem, the databases can be categorized using the way they store data. In addition to traditional Relational model (databases that do not use the relational model are frequently called NoSQL databases), databases can be divided into the following categories. [27].

- *Key-Value* The data are stored as pair Key-Value.
- *Wide Column or Column Families* Data are stored by columns; therefore some rows may not contain part of the columns which results in bigger flexibility in the data definition.
- *Document-oriented* Document is generally set of fields with attributes. A document can be represented in various formats like XML, JSON or BSON.
- *Graph-oriented* The data are stored in a graph-like structure.

Considering the data model, Document-oriented and Wide Column seem to be suitable for the system. In the database, there will be information about the job and job outputs. These data should be kept together so they can be queried easily. Therefore, both Document and Wide Column (or Column Families) can effectively store this information.

2.5.1 Cassandra

Cassandra is an open source distributed, decentralized, fault-tolerant, eventually consistent, linearly scalable, and a column-oriented data store [28]. Regarding CAP theorem, Cassandra is classified as an AP system. However, Cassandra consistency can be tuned for each driver; therefore it can be turned into a CP system easily while increasing latency.

Cassandra's data management topology can be described as a ring. To determine where (on which node) the data should be saved, the hash value of the primary key is computed; the hash is called token. Each node in topology is responsible for a range of tokens. Therefore, data are saved to the node that takes care of the range of tokens where the token falls.

Replication in Cassandra occurs in a transparent manner. If the replication is set to 4, then four nodes will have copies of each row in the range. The first replica is always saved to the node with the range where the token falls. Other nodes are chosen using Replication Strategy which can be changed. By default, Cassandra offers Simple Strategy and Network Topology Strategy. Simple Strategy assigns the replicas to consecutive nodes in the ring. Network Topology Strategy also considers different user-defined data centers and racks.

When any write operation is requested, first the change is written into CommitLog. CommitLog is component which tracks all write requests and is able to replay it in case of a problem. After the CommitLog, the change is written to MemTable is an in-memory representation of a column family. It can be thought of as cached data. When the system has gathered enough updates in memory or after a certain threshold time, flush the data to a disk in a structured file called SSTable [28]. Figure 2.13 shows the write operation.

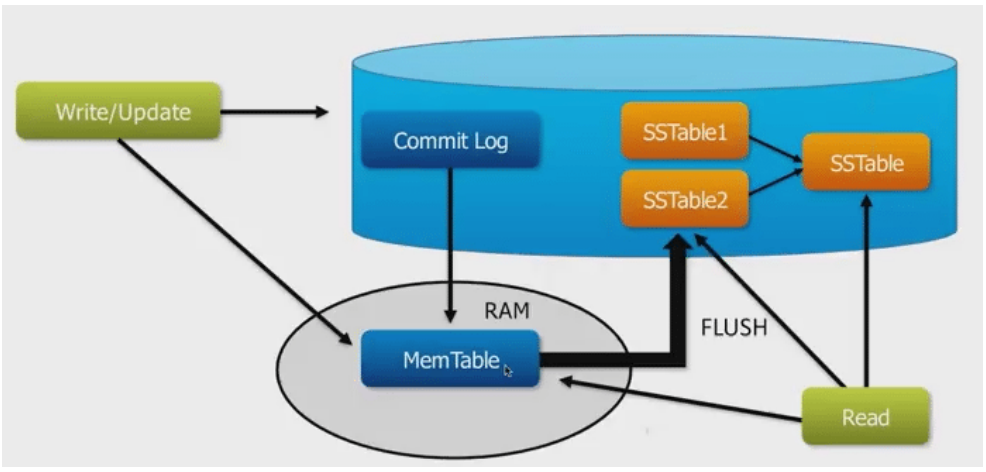

Figure 2.13: Write operation in Cassandra. Taken from `https://www.edureka.co/blog/introduction-to-cassandra-architecture/`

2.5.2 Elasticsearch

Elasticsearch is an open-source search engine built on top of Apache Lucene, a full-text search-engine library [29]. Elasticsearch is part of Elastic Stack and Graylog.

The Elastic Stack is a popular platform for collecting, analyzing and visualizing various data (mainly log files) from various sources. The Elastic stack consists of Elasticsearch, Logstash and Kibana. Graylog is similar product as Elastic stack. Graylog collects logs and provides real-time search and analysis. It is built on MongoDB and Elasticsearch.

Elasticsearch was designed as highly-scalable; it means it can be run on multiple nodes in a cluster. The cluster is uniquely identified by a unique name. Each node in the cluster have a unique name and participates in storing data, indexing and searching capabilities.

The main organizational unit in Elasticsearch is an index. The index is a collection of documents that share somewhat similar characteristic (e.g., tweets from Twitter can be in Elasticsearch saved under /twitter index). The basic unit of information is called a document. The document is expressed in JSON.

Because the index can contain a large amount of data, Elasticsearch provides an ability to split the index into multiple pieces called shards. On index creation, the user can specify the number of shards. The number of shards can be changed later but it is no trivial. Shards are a basic unit of replication. They are independent; therefore, can be each shard searched in parallel.

Elasticsearch's data replication is based on the primary-backup model. Based on documentID, the primary shard of the replication group is determined using routing. The primary shard is responsible for validating the operating and forwarding it to other replicas. Elasticsearch keeps the list of shards which received all the operations and can serve the actual data. When the node receives the read request, it forwards the request to the relevant shards. By default, Elasticsearch round robin between the replicas. Elasticsearch is a document-oriented database with a powerful engine for indexing and searching documents.

2.5.2.1 MongoDB

MongoDB is high performance and very scalable document-oriented database. It stores data in BSON format. BSON keep documents in an ordered list of elements, every element as three components: a field name, a data type and a value. BSON was designed to be efficient in storage space and scan speed which is done for large elements in a BSON document by a prefixed with a length field. All documents must be serialized to BSON before being sent to MongoDB; they're later deserialized from BSON by the driver into the language's native document representation [30].

MongoDB uses sharding to provide horizontal scalability. Sharding is a method for distributing data across multiple machines. There can be a mixture of sharded and unsharded collections in the database. MongoDB partitions the collection using the shard key. Shard key is an identificator of the document in the collection. It can consist of immutable field od document or fields that exist in every document in the collection. To perform Read/Write operation, routing service, called mongos, is used 2.14 MongoDB provides two sharding strategies, Hashed Sharding and Ranged Sharing.

MongoDB has a limit of maximum BSON document size 16 MB.

2.5.3 ArangoDB

ArrangoDB is a native multi-model database. It offers document-oriented storage, graph-oriented storage, and key-value storage. It is licensed under

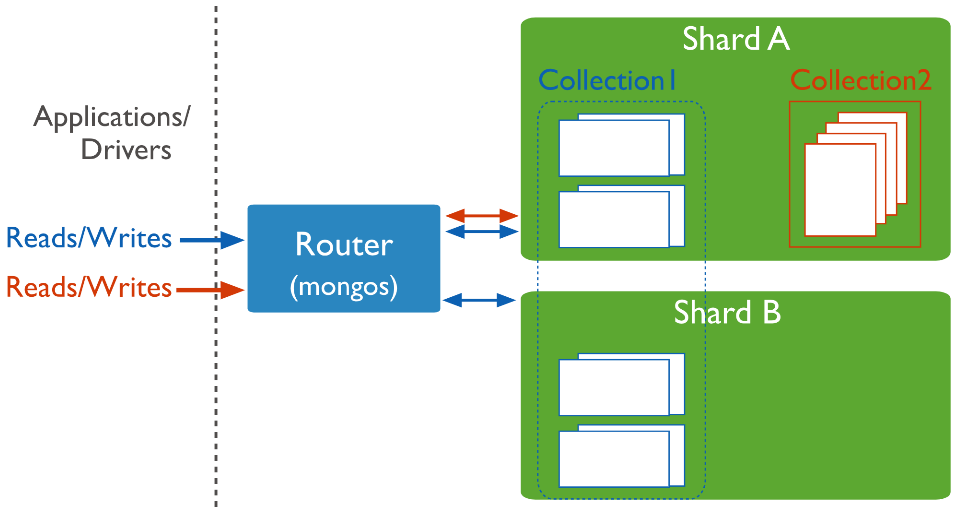

Figure 2.14: MongoDB collection sharding and routing [9]

open source Apache License 2.0 license.

The Cluster architecture of ArangoDB is a CP master/master model with no single point of failure [10]. There are 3 roles that can instance of ArangoDB have. They are Agents, Coordinators, and DBServers. Figure 2.15 shows the architecture of the cluster.

Agents perform leader election and keep the configuration of the server. Together, they form the Agency. Agency held the configuration of the server. Agency is a highly-available resilient key/value store and uses Raft Consensus Protocol to coordinate instances.

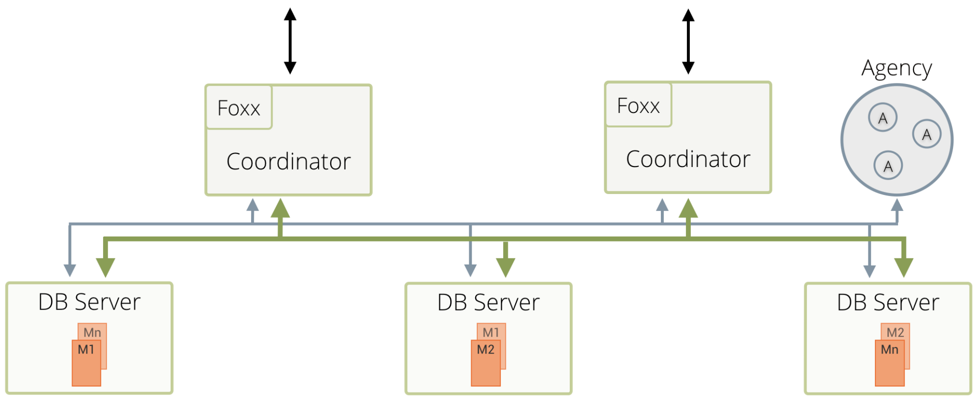

Figure 2.15: ArangoDB cluster architecture [10]

Coordinators are the endpoints that are exposed to the clients. They coordinate cluster tasks and run Foxx services.

DBServers are instances where data are hosted. The data are sharded and are synchronously replicated. DBServer can be a leader or follower of the data shard.

Foxx services are an interesting feature of ArangoDB that offers developer run microservice providing the database data directly on Coordinator.

2.5.4 HBase

HBase, an Apache open-source project, is a distributed fault-tolerant and highly scalable, column-oriented, NoSQL database built on top of HDFS. HBase is used for real-time read/write random-access to very large databases [31].

Regarding the CAP theorem, it is considered to be a CP system. To run HBase, there have to be a Hadoop File System (HDFS). This requirement increases the complexity of using it in the system. Therefore; it is not further considered a suitable option.

2.5.5 Amazon DynamoDB

DynamoDB is a database that supports key-value and document storage. However, it is a proprietary technology that disqualifies it from being used in the system.

2.5.6 CouchDB

CouchDB is highly available, partition tolerant, and eventually consistent Document Storage. It uses JSON to store the data and JavaScript to query the data.

2.5.7 Couchbase

Couchbase Server (previously known as Membase) is an open source, distributed Documents storage. It stores data as items, each of which has a key and a value. Key is a unique identifier and value can be binary or JSON.

2.5.8 Druid

Apache Druid is currently open source distributed data store. It is currently being incubated by The Apache Software Foundation. Druid is, similarly as Elasticsearch, focused on data analytics and search systems.

Chapter 3

Implementation

The system will be called DJEES (an abbreviation of Distributed Job Execution and Evaluation System). Firstly, the system was developed on the laptop. Then, it was deployed to a cluster that consist of four nodes each node with the following resources. The nodes have names storm1, storm2, storm3, and storm4.

- CPU: AMD Opteron Processor 6344 - 12 cores
- RAM: 32 GB
- Disk: Hard drive WDC WD1003FBYX - 1 TB

All the nodes run Ubuntu 16.04.

This chapter is divided into two parts. In the first part, the projects that are used in the DJEES are selected, and, then, the implementation is described.

3.1 Projects selection

3.1.1 Resource manager

The main candidates were Mesos, Kubernetes, and Nomad. The Mesos was chosen because it is not limited to running only Docker containers. It means Mesos supports running custom frameworks for example for Kafka or Cassandra. Although these frameworks were not used because of the deprecated versions, custom frameworks were considered as project advantage. Frameworks were also considered as Mesos advantage over Nomad.

3.1.2 Scheduler

The scheduler is generally Mesos framework. Every framework that is run on Mesos clusters has to use Mesos APIs to communicate with Mesos Master and to schedule task.

3.1.2.1 Singularity

Singularity targets to be more generic middle part that allows a various application to be run on Mesos cluster. Singularity combines long-running tasks and job scheduling functionality and provides a uniform way to run applications. Singularity currently supports following process types [11].

1. Web Services
2. Workers
3. Scheduled Jobs
4. On-Demand Processes

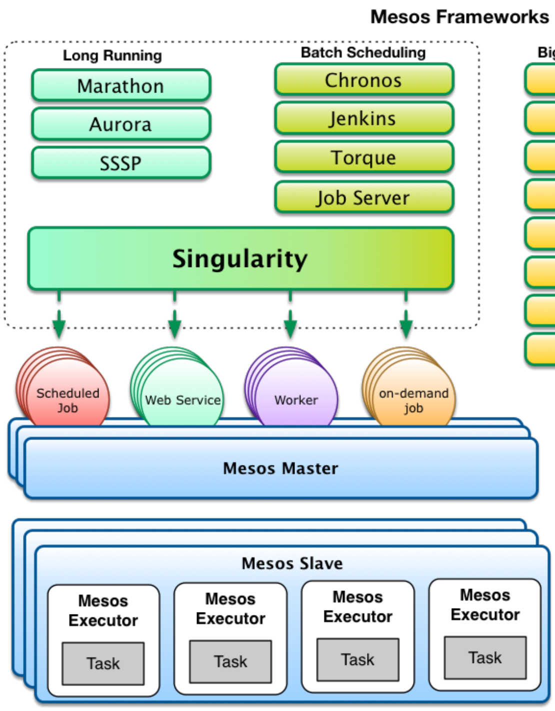

Figure 3.1: Singularity functionality [11]

The Singularity was my first choice when prototyping the system. However, I did not found API documentation, and the documentation was overall confusing.

3.1.3 Aurora

Apache Aurora is Mesos Framework that supports long-running services, cron jobs, and ad-hoc jobs. Aurora claims that it lets using an Apache Mesos cluster as a private cloud. It means that it production grade tool that is used by many top companies (Twitter, Uber, PayPal and many more). However, Aurora is quite difficult to configure, and it does not expose any Http API (or at least there is no documented Http API). Therefore, the Thrift API has to be used. It increases the complexity of implementing and running this framework.

3.1.3.1 Marathon

Marathon is a container orchestration platform for Mesos. It is mainly focused on running services. It schedules services to the nodes, monitor them and restarts them if it is needed. Marathon provides various options for fine-tuning the service. Marathon offers out of the box UI and well documented REST API. Therefore, Marathon was chosen to run the services in the Mesos.

3.1.4 Metronome

Metronome is a Mesos framework for scheduled jobs. It uses Zookeeper to keep information about jobs. It has a simple REST API that is well-documented. Therefore, it was chosen for submitting user jobs to the Mesos cluster.

3.1.5 Containerization platform

There are quite a lot of containerization projects; however, Docker currently dominates the marker. [32] calculates Docker market share at 83%. The second is CoreOS RKT with 12%, 3rd is Mesos containers with 4% and the last mentioned are Linux Containers LXC with 1% market share. Generally, Docker is more widespread; therefore, it can be expected that many users already have some experience with Docker. Moreover, Docker can be built on Windows too. Since it can be expected that the users will have to prepare the container to run the job, the Docker containers are considered as the main platform for running the user jobs.

Docker is an open source project that builds on many long-familiar technologies from operating systems research: LXC containers, virtualization of the OS, and a hash-based or git-like versioning and differencing system, among others [33].

Docker shares with the host only the kernel. It means all the dependencies have to be packed inside the container. The main file for the constructing new container is *Dockerfile*. In the Dockerfile, the user specifies how the container should be created. User can specify the source image, commands that have to be run to install programs or services, the files that should be copied inside the container. Then the container can be built and pushed to Docker Registry. Docker Registry is an open source server for storing and distributing Docker images. The system maintainer can use Registry provided by Docker called Docker Hub or deploy own instance of the Docker Registry. The Docker was already chosen as the main containerization technique in the previous text. The main argument for selecting the Docker is its popularity, widespread, and overall good user experience. Moreover, it is easy to use and is well supported by Mesos.

3.1.6 Logs extraction

The problem of extracting output from the container can be solved in several ways. There can be another process inside the container that will forward the output to the endpoint outside the container. However, this approach means exposing the endpoint to the running process and potential security risk. Another approach is a shared file between container and host system. However, this approach exposes filesystem (or part of it) to the container and is security risk too. Moreover, both approaches set the requirements on the user code and reduce the usability and potentially user experience. Therefore, this is system is going to use the log forwarders that are connected directly to the container socket and forward stdout and stderr of the container.

The process of log forwarding consists of two steps. First, a special driver that is supported by Docker forward logs to the service or server. Then, the service or server forwards the log messages to the targeted system. The targeted system is a message queue.

3.1.6.1 Docker drivers

Docker currently supports many logging drivers. Moreover, a custom plugin for logging can be installed into Docker. However, the following text describes the drivers can be potentially used in the system.

3.1.6.1.1 Syslog Syslog is a standardized protocol for event notification messages; it is standardized in [34] (also specified as RFC 5424). It uses client-server architecture.

3.1.6.1.2 Journald Journald is logging driver that is able to send the container logs to the systemd journal.

3.1.6.1.3 Gelf Gelf is an abbreviation of Graylog Extended Format. Graylog is a popular open-source logging solution. It is build using Elasticsearch and MongoDB projects. It can be used with various tools such as Graylog, Logstash, and Fluentd.

3.1.6.1.4 Fluentd Fluentd driver sends logs to the Fluentd collector.

3.1.6.1.5 Splunk Splunk is a product for monitoring and analyzing generated data. The Docker Splunk logging driver sends container logs to HTTP Event Collector in Splunk product. However, Splunk is a commercial product; therefore, it is not suitable for the system.

3.1.6.2 Log forwarding systems and services

The Docker logging driver can forward the log message to various log forwarding or log processing services. In the proposed system, the service objective is to send the log message to the message queue. There is an enormous amount of projects for log forwarding with various architecture. The most popular projects are following.

3.1.6.2.1 Logstash Logstash is part of The Elastic Stack. It is open source, and its primary function is to collect the data from various sources and sends them to Elasticsearch. However, Logstash has pluggable architecture so it can be used with various projects.

3.1.6.2.2 Logagent Logagent is open-source, light-weight log shipper. It claims that uses low memory and low CPU. It workes as Syslog listener.

3.1.6.2.3 Fluentd Fluentd is an open-source platform for data collection. It is also presented as a unified logging layer. It can process various messages and forward them to various systems. The main source of versatility is its Pluggable Architecture.

3.1.7 Log forwarding solution

Suitable drivers for the system are Syslog, GELF, and Fluentd. However, in this thesis, the decision was made between Logstash and Fluentd. The difference between Logstash and Fluentd is in the way they handle the output. Fluentd uses tags to determine the action; Logstash uses if-else statements. Logstash and Fluentd both use similar pluggable architecture. [35] suggests that Fluentd might be faster and uses less memory. However, the differences were not further investigated, and Fluentd was chosen because of personal preference and experience from creating a prototype (Fluentd is easy to run and configure).

3.1.8 Output queuing

Apache Pulsar looks interesting; however, Fluentd nor Logstash do not have output plugin for Apache Pulsar. Kafka was chosen as the best option because it is designed as high throughput and it supports the consumer scaling.

3.1.9 Data storage

The main arguments for Cassandra were that it is highly scalable, fault-tolerant and high-throughput. Moreover, it allows fine-tuning of the queries consistency. Therefore, the writing queries can be optimized for speed and reading queries for consistency.

3.2 Configuring and connecting projects

The whole process of the configuration is described in detail in Appendix B. For the project development and necessary functionality testing, project Minimesos (`https://minimesos.org`) is available. It is an excellent ready-to-use tool.

Figure 3.2 shows the final project architecture and the output flow between the parts of the system.

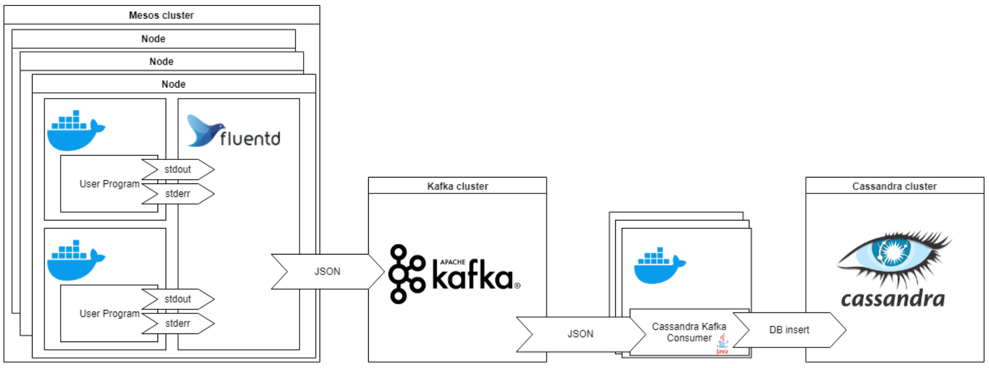

Figure 3.2: System overview

The number of servers does not allow to divide the services among the nodes; therefore, the system components are running simultaneously on the nodes. It means that there are usually running Zookeeper, Mesos-master, Mesos-slave, Kafka and Cassandra on the same node to simulate the distributed environment. This setup is not ideal; however, it should not significantly affect constructing or testing the system.

3.2.1 Zookeeper and Mesos

As a first step, the Zookeeper have to be installed. Mesos and both frameworks that are going to be used (Metronome and Marathon) depend on it. After that, Mesos can be deployed. Mesos consists of two components master and agent (previously called as a slave). The master takes care of resource offering to frameworks and assigning jobs to agents. The agent has to be deployed on each node that is intended for running the jobs. The agent is responsible for running the framework executor on his node. If there are multiple master instances leader is elected using Zookeeper; therefore, the master must be configured with the addresses of the Zookeeper nodes.

The agent uses Zookeeper to determine the leading master which should offer resources. Therefore, the agent must be configured with the addresses of the Zookeeper nodes too. Moreover, the agent has to be configured with the supported containerization methods. By default, it only uses Mesos containers. Furthermore, the offered resources can be configured. There are four distinct resource types of resources that the agent offers and that have to match to run a job.

- CPU
- Memory
- Disk
- Port (by default only ports 31000 - 32000 are offered)

3.2.2 Docker and log forwarder

Mesos supports the Docker from version 0.20.0 (version 1.7.2 is used in this project). However, to run the Docker container on the node, there have to be Docker installed. Therefore, the Docker must be installed on each node that is intended for running the jobs. Moreover, the log forwarder is tightly coupled with log forwarder. The system is using Fluentd log forwarder that is distributed as service called td-agent.

3.2.3 Kafka

There is a project that runs Kafka as a Mesos framework that is called mesos/kafka, but the project is not maintained, and it would be difficult to configure and run.

The Kafka was also considered to be run as a Marathon service inside the container. However, Kafka is a complex project, and it is difficult to configure to run inside the docker container. More about the running the Kafka inside the Docker can be found on `https://rmoff.net/2018/08/02/kafka-listeners-explained/`. Besides, running the Kafka inside the Docker using

Mesos increases the complexity of monitoring and debugging it. Therefore; Kafka was installed as service directly on the nodes.

Kafka uses Zookeeper to keep information about topics and partitions. Therefore, Kafka has to be configured with addresses of the Zookeeper nodes.

In addition, when Kafka is started, a new topic for the output messages have to be created. In the DJEES, the topic is called metronome because the jobs in the system are scheduled by Metronome framework. In Kafka, each topic has two properties - replication factor and number of partitions. In the DJEES, the metronome partition is configured with replication=2, to provide fault-tolerance, and partitons=3 to provide customers' scalability.

3.2.4 Connecting Fluentd with Kafka

Fluentd uses plugin architecture. There many various input and output plugins. DJEES utilizes standard forward input plugin. For sending messages to Kafka, there is a couple of output plugins. DJEES uses kafka2 plugin. However, this plugin contains a bug that prevents the plugin from injecting tag and formatted timestamp into the message. Tag is important because it is used to determine the job that produced the message. The timestamp is used for messages ordering. Therefore, the right format with the specified precision is crucial for determining the correct message order. Because of these reasons, the filter plugin is used to inject tag and timestamp to the message. All three mentioned plugins are already included in package td-agent3. It means that they can be turned on by proper configuration.

3.2.5 Cassandra

Cassandra was also installed as from Cassandra repository. Firstly, there was an idea of running Cassandra as a Marathon service inside the Docker. However, running a distributed database inside the Docker brings a lot of challenges (similar to running Kafka inside Docker); therefore, Cassandra is run directly on the node.

When started, Cassandra uses a ring topology. Therefore, Cassandra should be configured with seed addresses. Seeds are the nodes that are already connected to the ring. When the instance of Cassandra is started, it initializes the gossip with the seed node and joins the ring topology. However, the topology is established only when the instance starts, and the instance keeps the configuration over the instance restarts. Therefore, it is necessary to remove Cassandra data before joining a new ring.

Then the keyspace and tables have to be created. When creating the keyspace, two properties have to be defined. The replication strategy defines how the data are replicated among the nodes. The replication factor defines how many copies of the data there will be in the Cassandra cluster. DJEES is configured to use SimpleStrategy and replication_factor=2 to provide fault-tolerance.

Then, tables can be defined inside the keyspace. Each table has columns with the defined type. Moreover, for each table, the primary key has to be defined. The primary key can be a compound of the partition key and clustering columns. The unique identification of the row is a combination of the partition key and clustering columns. Rows with the same partition key are kept in the same table partition and are ordered using clustering columns.

In the DJEES, there are two keyspace, runs and runInformation. The jobs outputs are stored in the keyspace runs in tables stdout and stderr. The partition key is a field source that is a unique identifier of the job run. The clustering column is a timestamp of the log message. Because the Cassandra uses the timestamp to order the rows in partition, it is ensured that if the job run is queried, the log messages are returned in the correct order using timestamp. The information about the jobs is stored in keyspace runInformation in tables task and taskRun. Figure 3.3 shows the columns of tables task and taskRun. The partition keys are marked with asterisk and clustering column is marked with a hash symbol.

runInformation.task		runInformation.taskRun	
∗ user	text	∗ user	text
# name	text	∗ name	text
command	text	# runId	int
image	text	state	text
submissionTime	text	submissionTime	text
memory	int	finishedAt	text
cpus	float		
disk	int		

Figure 3.3: Tables runInformation.task and runInformation.taskRun

3.2.6 Marathon

Marathon is a Mesos framework that deploys services and applications on Mesos agents. It allows to monitor them and in case of failure restart them. In the DJEES, it is used to run Metronome, TaskSubmitter, and CassandraKafkaConsumer which are described in the following text.

For running Marathon, Mesosphere Docker image in version 1.5.1 was used. Marathon uses Zookeeper to provide fault-tolerance and elect a leader. Therefore, it has to be configured with the addresses of Zookeeper. Moreover, since it is a Mesos framework, it has to be also configured with Mesos path in the Zookeeper.

3.2.7 Persisting data from Kafka to Cassandra

The problem of moving log messages from the Kafka queue to Cassandra is solved by Java application. The application is called CassandraKafkaConsumer and uses Apache Kafka library called kafka-clients for consuming messages from Kafka and the Cassandra driver from Datastax for storing data to Cassandra. First, the class KafkaConsumer uses ConsumerFactory class to sign up for topic.

To sign up for the topic, Consumer needs at least one valid address of Kafka node, topic name, and the GroupId. GroupId identifies the groups and allows load balancing partitions consuming among the multiple instances of the CassandraKafkaConsumer.

For the connection to Cassandra, class CassandraConnector is used. It needs at least one valid address (in a driver called contactPoint) of Cassandra node and name of the datacenter that was Cassandra configured with. The CassandraConnector on startup prepares queries for inserting the log messages to the Cassandra.

When a new message is accepted, KafkConsumer class uses Gson converter to convert the message to Java object. Then, using the source field determines whether the log message should be inserted to table stdout or stderr. Finally, the object is persisted to the Cassandra using CassandraConnector class.

To run the CassandraKafkaConsumer on Marathon, CassandraKafkaConsumer has to be registered using Marathon REST API. In the DJEES, the Marathon is configured to run three instances of CassandraKafkaConsumer, each on a different node. Running the application on different nodes is ensured using constraint *unique hostname.*

3.2.8 Metronome

Metronome is a Mesos framework, but it is focused on one-off tasks and scheduled jobs. In the DJEES, it is used to schedule users jobs on the Mesos cluster. To run Metronome, the docker image was prepared and run using Marathon.

3.2.9 Job submitting

Submitting jobs to the system is solved by an application build on Java Spring framework that is called TaskSubmitter. It uses the Datastax Cassandra driver to connect Cassandra. The application uses Spring Boot server and layered architecture. The application was designed to simplify the task submitting.

TaskSubmitter uses two classes to represent the job. The first one is Task class that represents a general problem and holds the information about job name, command and container that should be run. The second one is class TaskRun that represents a single run of the job. It contains runId, state, submission time.

The primary endpoint is /runs/user that can be used to list all the users' tasks and to add a new task. The system does not currently support authentication and authorization; however, the design allows to implement it using Spring Boot Security easily.

To submit the new job the user has to specify the task name, requested resources, container (in the JSON called image) and the command. When the job is submitted, the job run must be triggered using Http Post method. The system generates runId that is used to identify the task run and obtain the run output.

DJEES currently supports only two states of the job run. When the job run is requested, it enters the state *submitted*. When it finishes, the state is changed on *finished*, and the column finishedAt is updated. Unfortunately, Metronome does not support webhooks; therefore, the job runs' states are updated when user requests list of the Job Runs. The information about the Job run can be obtained from Metronome using query parameter embed=historySummary. When the Job run enters the finished state, the Job run is removed from Metronome. The reason for this is that Metronome saves information about the run into the Zookeeper. It is expected that there are going to be a huge amount of jobs and job runs and it could have a significant effect on Zookeeper performance. Moreover, Zookeeper is used by critical parts of the system (Mesos, Marathon, Kafka). Furthermore, there is minimum extra information versus information that is saved in Cassandra.

Currently, TaskSubmitter offers the following functionality.

- List all user Jobs
- Submit a new Job
- List Job runs
- Get job run standard output
- Get job run standard error output

Marathon uses Http to check if the application is running. For this purpose, Task Submitter has /health endpoint that response to Http Get method.

Chapter 4

System analysis

4.1 Capabilities demonstration

For the demonstration purposes, the first assignment from the MI-PAA was chosen. The assignment is to implement simple recursive solving of the Knapsack problem. The solution was implemented in Python and C++ using simple recursion. To demonstrate possibilities, the solution in C++ was cloned from the Gitlab repository and run in the prepared image. The Python solution is packet inside the docker image and only the run script is called.

4.1.1 Running C++

Figure 4.1 shows JSON for running the C++ code in the DJEES. Firstly, the code is downloaded from the Gitlab. Then, it is compiled, and finally, the binary is run. To submit the job, the JSON has to be sent to /runs/user endpoint using the Post method. The Linux utility curl was used for this purpose. Figure 4.2 shows the command that was used during the testing phase. When the job is submitted, it has to be triggered to run using the Post method on /runs/user/taskName endpoint. Figure 4.3 shows the command that can be used to trigger the job run and response from the system. The response contains generated identifier runId that can be used to obtain the real-time output. Figure 4.4 shows the command for getting the job standard output and the response from the system. Figure 4.5 shows the command for getting standard error output and the response from the system.

The time measurement was performed using C++ chrono library.

4.1.2 Running Python

The purpose of this test was to demonstrate that the system can run any language that the user supplies with its dependencies.

```
{
  "command":
    "git clone https://gitlab.com/dip-test-cases/paa-brute-force.git
     && cd paa-brute-force && g++ main.cpp -o run && ./run",
  "cpus": 1,
  "disk": 0,
  "image": "mashtak/c_cpp_image:1.0",
  "memory": 512,
  "name": "paacppbruteforce"
}
```

Figure 4.1: Json for running C++ algorithm

```
curl -H "Content-Type: application/json" -X POST -d@paa_c++.json \
storm2:31800/runs/masekja7
```

Figure 4.2: Command for submitting JSON to system

```
curl -X POST storm2:31800/runs/masekja7/paacppbruteforce

{
"user": "masekja7",
"name": "paapythonbruteforce",
"runId": 1,
"submissionTime": "2019-05-05T21:04:10.489Z",
"state": "submitted",
"finishedAt": null
}
```

Figure 4.3: Command for triggering job run and system response

The process of submitting a Python job is the same as for C++ program. The only difference is the JSON for submitting a job (Figure 4.6). The job run can be triggered using the name defined in JSON (paapythonbruteforce) and output is obtained the same way as in C++ version (Figure 4.7).

4.2 High load

The goal of this testing phase was to discover if the system isolates resources and give the same results. First, the test triggering 40 runs of the C++ recursive solving Knapsack problem. The same implementation as in 4.1.1 was

```
curl -X GET storm2:31800/runs/masekja7/paacppbruteforce/1/stdout

{
"ouput": [
      "-------knap_4.inst.dat--------",
      "0.000198077",
      "-------knap_10.inst.dat--------",
      "0.00352617",
      "-------knap_15.inst.dat--------",
      "0.177497",
      "-------knap_20.inst.dat--------",
      "6.39341",
      "-------knap_22.inst.dat--------",
      "27.4397"
],
"user": "masekja7",
"name": "paacppbruteforce",
"runId": 1,
"type": "stdout"
}
```

Figure 4.4: Command for obtaining job standard output and system response

```
curl -X GET storm2:31800/runs/masekja7/paacppbruteforce/1/stderr

{
"ouput": ["Cloning into 'paa-brute-force'..."],
"user": "masekja7",
"name": "paacppbruteforce",
"runId": 1,
"type": "stderr"
}
```

Figure 4.5: Command for obtaining job standard error output and system response

used; however, the only instance of size 22 was measured. The measurement was performed on DJEES with 3 Mesos agents running on 3 nodes. Figure 4.8 shows the command that was used to trigger the job runs and figure 4.9 shows the obtained results.

Unfortunately, the significant differences can be observed between the fastest and slowest solution (column Measured time). The fastest solution

```
{
  "command": "./test_script.sh",
  "cpus": 1,
  "disk": 0,
  "image": "mashtak/mi-paa_python_bruteforce:1.0",
  "memory": 512,
  "name": "paapythonbruteforce"
}
```

Figure 4.6: Json to run the Python program

spent 30.87 seconds performing the task; however, the slowest run spent with the same task 48.259 seconds. It means that the slowest run is more than 50% slower than the fastest. Moreover, if there is no load on the node, the job runs even faster (27.4397 seconds shown in 4.1.1).

The cause of big difference could be that other services are running on the nodes (Kafka, Cassandra) but the Mesos agent offers the full capacity. Another test was performed to verify it. The C++ Knapsack recursive solver was run together with the stress utility that can impose load on CPU. To perform this test, only one Mesos Agent was used. Figure 4.10 shows the JSON that was used to configure the job. The stress utility tries to stress all the 12 node CPUs, although, it has dedicated only one CPU. It stresses the CPU for 2 minutes.

The measured time when there was a stress utility running on the same node was 44.705 seconds. However, without load, the same job on the same server runs between 27 and 29 seconds. This test shows that CPU isolation does not work correctly and that simultaneously running jobs affect each other.

The problem of the job resource isolation lays in the way how the Mesos and Docker use CPU parameter. The problem of resources limitation is described in [36]. Mesos converts the CPU parameter to Docker's parameter cpu-shares. However, the cpu-share option defines only the priority of the container and the portion of the CPU time that the container receives. Therefore, when running the stress test and the C++ Knapsack solver on the same node with CPUs equal 1, both processes have the same priority; therefore C++ solver receives only 50% of CPU time which results in a significant slowdown of the C++ solver.

Docker offers parameter −−cpus that limits the usage of the available CPUs. This option would be more suitable for this system. It could be passed as additional Docker parameter to Metronome; however, this option was not implemented nor tested. This option was successfully tested using htop to monitor system resources. Therefore, it can be implemented to the system in the future.

4.3 Security

System security can be viewed from two perspectives. First is the security of the interface that is exposed to users. Second is running user code that could be potentially malicious.

The only interface that is meant to be exposed to the users is TaskSubmitter application. However, TaskSubmitter is not ready for a production environment. It contains protection against SQL injection because all the queries use Prepared Statements, but Task Submitter does not implement any Authentication nor Authorization; therefore it currently cannot be used for serving multiple users.

The security of running depends on container isolation. There can be two goals of the malicious code. It can be using system resources to attack the targets on the internet. The second possibility is that the attacker wants to gain control of the node potentially the whole system.

The first risk, attacking targets on the network, can be mitigated by setting up firewall and white-listing allowed website. For example, the container could only connect to the specified git server and the server with containers.

The second risk, gaining host root access is a big security problem that is not solved satisfyingly. In the paper [37], there was a huge amount of attack analyzed, and security mechanisms were created. Moreover, the new vulnerability CVE-2019-5736 was discovered recently (it is described in [38]). However, the more profound analysis of the container security is beyond the scope of this thesis. The main conclusion of this analysis is that containers in the default configuration cannot be currently considered as a secure way to run 3rd party programs. However, there are mechanisms and techniques to minimize the risk of gaining control of the host machine. The problem is that they are not standardized and setting the containers properly needs particular caution.

4.4 Scalability

The system is scalable in several ways. If only more computational power is needed, the node with Mesos agent can be added. The node will offer its resource; hence, increases available system computational power.

When adding new node the following steps have to be made.

- Install Docker
- Install td-agent(Fluentd)
- Configure td-agent to forward messages to Kafka
- Install Mesos agent

- Configure Mesos agent with Zookeeper to be able to reach Mesos master

Although the scaling computational power increases the overall computational power of the system, other parts of the system have to be able to handle increased traffic. However, determining the required amount is currently impossible because presumptions cannot be about the jobs the will be run in the system. There can be a few compute-intensive jobs that generate very few output messages or there can be many jobs and will generate a huge amount of output messages.

All of the system components are easy to scale. Adding nodes to Kafka or Cassandra means installing the service on the node and connecting it to the existing topology. When scaling CassandraKafkaConsumer, two things have to be done. First, there has to be enough partition in the Metronome topic. Otherwise, adding new CassandraKafkaConsumer would not have an effect. Generally, there should be at least the same number partition as is CassandraKafkaConsumers in the system. Second, adding more instance of the CassandraKafkaConsumer can be done changing Marathon JSON and updating it in Marathon.

Problematic is scaling Task Submitter and Metronome. Scaling TaskSubmitter means that users need to know the location of the server they should contact. It induces the need to have a load balancer that would divide the traffic among the instances of Task Submitter. Moreover, the Task Submitter needs to be configured with the address of the Metronome; therefore, TaskSubmitter has to be scaled simultaneously with Metronome.

4.5 Fault-tolerance

The fault-tolerance of the system have to analyzed be analyzing each component. Zookeeper, Mesos, Marathon, Kafka, and Cassandra are fault-tolerant by design (however, Kafka and Cassandra have to configured with replication factor at least 2). However, when the node running job stops, the job has to be restarted. CassandraKafkaConsumer is also fault-tolerant because Kafka can forward to the partition to other instance of the consumer. However, TaskSubmitter is not fault-tolerant. It is going to be restarted by Marathon if the application fails. However, the Marathon deploys the Task Submitter only to the node with the specified hostname. Therefore, if the node with the specified name is not accessible, the Task Submitter will not be redeployed.

4.6 Further development

There are many possibilities for extending the current system. Mesos is a versatile platform that can be extended in several ways.

Mesos resource managing is not limited only to CPUs, memory, and disk, but can provide GPUs as well. GPUs are used mainly in Machine Learning and

Artificial Intelligence. Moreover, there is an implementation of popular TensorFlow as Mesos framework (`https://github.com/douban/tfmesos`). TensorFlow is a framework that greatly simplifies developing Machine Learning and Artificial Intelligence applications.

Another possibility is exploring and integrating framework mesos-hydra (`https://github.com/mesosphere/mesos-hydra`). Mesos hydra is framework for running MPI applications in Mesos cluster. Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to run parallel applications on a wide variety of parallel computing architectures.

Finally, migrating to DC/OS should be considered. DC/OS is a distributed operating system based on Mesos that simplifies using Mesos and integrates many projects that were used in this project.

```
{
    "ouput": [
        "knap_10.inst.dat",
        "------brute_10------",
        "",
        "real\t0m0.139s",
        "user\t0m0.136s",
        "sys\t0m0.000s",
        "knap_15.inst.dat",
        "------brute_15------",
        "",
        "real\t0m1.986s",
        "user\t0m1.980s",
        "sys\t0m0.008s",
        "knap_20.inst.dat",
        "------brute_20------",
        "",
        "real\t0m59.668s",
        "user\t0m59.636s",
        "sys\t0m0.020s",
        "knap_22.inst.dat",
        "------brute_22------",
        "", "real\t3m35.005s",
        "user\t3m34.956s",
        "sys\t0m0.008s",
        "knap_4.inst.dat",
        "------brute_4------",
        "",
        "real\t0m0.029s",
        "user\t0m0.020s",
        "sys\t0m0.012s"
    ],
    "user": "masekja7",
    "name": "paapythonbruteforce",
    "runId": 5,
    "type": "stdout"
}
```

Figure 4.7: System output for Python program

```
for i in {1..40}; do  curl -X POST \
 storm2:31800/runs/masekja7/paacppbruteforce; done
```

Figure 4.8: Command for triggering 40 runs of the C++ solver

RunId	Submission time	Finished time	Time to finish	Measured time
1	18:34:42.802	18:37:32.868	0:02:50.066	33.692
2	18:34:42.881	18:37:19.680	0:02:36.799	47.624
3	18:34:42.963	18:36:42.243	0:01:59.280	31.652
4	18:34:43.034	18:36:45.308	0:02:02.274	36.420
5	18:34:43.099	18:36:44.389	0:02:01.290	37.858
6	18:34:43.176	18:37:19.067	0:02:35.891	47.146
7	18:34:43.239	18:36:46.618	0:02:03.379	34.637
8	18:34:43.307	18:37:21.091	0:02:37.784	46.854
9	18:34:43.371	18:37:25.816	0:02:42.445	31.212
10	18:34:43.440	18:37:25.916	0:02:42.476	30.960
11	18:34:43.510	18:37:21.394	0:02:37.884	46.334
12	18:34:43.580	18:36:49.343	0:02:05.763	38.299
13	18:34:43.651	18:37:28.749	0:02:45.098	31.106
14	18:34:43.721	18:36:44.542	0:02:00.821	40.662
15	18:34:43.789	18:37:21.908	0:02:38.119	47.238
16	18:34:43.853	18:36:39.809	0:01:55.956	37.329
17	18:34:43.913	18:37:23.141	0:02:39.228	47.179
18	18:34:43.975	18:37:23.334	0:02:39.359	46.696
19	18:34:44.046	18:37:29.542	0:02:45.496	30.959
20	18:34:44.114	18:37:25.518	0:02:41.404	30.870
21	18:34:44.178	18:37:29.951	0:02:45.773	31.881
22	18:34:44.247	18:36:47.544	0:02:03.297	37.693
23	18:34:44.315	18:36:45.349	0:02:01.034	42.286
24	18:34:44.384	18:37:21.601	0:02:37.217	47.264
25	18:34:44.447	18:36:47.781	0:02:03.334	41.163
26	18:34:44.522	18:37:35.689	0:02:51.167	36.388
27	18:34:44.580	18:36:42.345	0:01:57.765	31.630
28	18:34:44.646	18:36:42.650	0:01:58.004	38.211
29	18:34:44.713	18:37:28.758	0:02:44.045	32.365
30	18:34:44.780	18:37:20.484	0:02:35.704	47.999
31	18:34:44.845	18:36:43.250	0:01:58.405	38.683
32	18:34:44.913	18:37:28.134	0:02:43.221	31.962
33	18:34:44.979	18:36:41.928	0:01:56.949	31.696
34	18:34:45.045	18:37:31.367	0:02:46.322	34.675
35	18:34:45.109	18:36:46.350	0:02:01.241	39.760
36	18:34:45.171	18:37:18.976	0:02:33.805	48.259
37	18:34:45.230	18:37:28.542	0:02:43.312	36.900
38	18:34:45.295	18:36:42.516	0:01:57.221	38.006
39	18:34:45.363	18:37:23.026	0:02:37.663	46.991
40	18:34:45.425	18:37:22.517	0:02:37.092	46.258

Figure 4.9: Running 40 instances of the C++ Knapsack recursive solver

```
{
  "command": "stress --cpu 12 --timeout 120s",
  "cpus": 1,
  "disk": 0,
  "image": "containerstack/alpine-stress",
  "memory": 512,
  "name": "stress"
}
```

Figure 4.10: JSON for running stress utility

Conclusion

The current open-source technologies were analyzed. The methodology for evaluating and selecting open source technologies was defined. Suitable technologies were chosen using that methodology. The system for the distributed job execution and evaluation was implemented using selected technologies. It was proven that chosen technologies can be used to provide a scalable environment for running user jobs. The system is able to run a job on a computing cluster with requested resources. Display its status, and show its output in real-time. The output is persisted into the Cassandra database. Jobs are run in containers to provide isolation of the jobs. Containerization of jobs also allows running any programming language. Capabilities were demonstrated; however, further tests showed that CPU limitation does not work as expected because of the Docker implementation. Therefore; DJEES can provide inconsistent results in specific scenarios. However, a solution was found and can be implemented in the future.

DJEES is a complex and versatile system that offers many possibilities for further development. The main areas for the further development were addressed in Chapter 4.

Bibliography

[1] Hindman, B.; Konwinski, A.; et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*, volume 11, 2011, pp. 22–22.

[2] Vavilapalli, V. K.; Murthy, A. C.; et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, ACM, 2013, p. 5.

[3] Nomad, Architecture. Accessed: 2019-2-4. Available from: `https://www.nomadproject.io/docs/internals/architecture.html`

[4] Lewis, I. Container Runtimes Part 3: High-Level Runtimes. Accessed:2019-05-08. Available from: `https://www.ianlewis.org/en/container-runtimes-part-3-high-level-runtimes`

[5] Apache Kafka, Introduction. Accessed:2019-1-4. Available from: `https://kafka.apache.org/intro`

[6] Dossot, D. *RabbitMQ essentials*. Packt Publishing Ltd, 2014.

[7] Nguyen, C. N.; Lee, J.; et al. On the role of message broker middleware for many-task computing on a big-data platform. *Cluster Computing*, 2018: pp. 1–14.

[8] Gill, N. S. NoSQL Databases – Overview, Types and Selection Criteria. Accessed: 2019-05-05. Available from: `https://www.xenonstack.com/blog/nosql-databases/`

[9] MongoDB Documentation - Sharding. Accessed: 2019-2-13. Available from: `https://docs.mongodb.com/manual/sharding/`

[10] Cluster Architecture. Available from: `https://www.arangodb.com/docs/stable/architecture-deployment-modes-cluster-architecture.html`

[11] About Singularity. Accessed: 2018-10-16. Available from: `https://getsingularity.com/Docs/about/how-it-works.html`

[12] Companies that use Apache Mesos. Accessed: 2019-05-05. Available from: `https://stackshare.io/mesos`

[13] Burns, B.; Grant, B.; et al. Borg, omega, and kubernetes. *Queue*, volume 14, no. 1, 2016: p. 10.

[14] Companies that use Kubernetes. Accessed: 2019-05-05. Available from: `https://stackshare.io/kubernetes`

[15] TORQUE architecture. Available from: `http://docs.adaptivecomputing.com`

[16] Nomad vs. Mesos with Aurora, Marathon. Accessed: 2019-05-05. Available from: `https://www.nomadproject.io/intro/vs/mesos.html`

[17] Companies that use Nomad. Accessed: 2019-05-05. Available from: `https://stackshare.io/nomad`

[18] Lewis, I. Container Runtimes Part 2: Anatomy of a Low-Level Container Runtime. Accessed:2019-05-08. Available from: `https://www.ianlewis.org/en/container-runtimes-part-2-anatomy-low-level-contai`

[19] Lewis, I. Container Runtimes Part 4: Kubernetes Container Runtimes & CRI. Accessed:2019-05-08. Available from: `https://www.ianlewis.org/en/container-runtimes-part-4-kubernetes-container-run`

[20] rkt vs other projects. Accessed:2019-05-08. Available from: `https://coreos.com/rkt/docs/latest/rkt-vs-other-projects.html`

[21] Chia, T. Container Runtimes. Accessed:2019-05-08. Available from: `https://www.ayrx.me/container-runtimes`

[22] Wang, G.; Koshy, J.; et al. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, volume 8, no. 12, 2015: pp. 1654–1655.

[23] Dobbelaere, P.; Esmaili, K. S. Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, ACM, 2017, pp. 227–238.

[24] Akgul, F. *ZeroMQ*. Packt Publishing Ltd, 2013.

[25] Pulsar Overview. Accessed: 2019-05-05. Available from: `https://pulsar.apache.org/docs/en/concepts-overview/`

[26] Gilbert, S.; Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, volume 33, no. 2, 2002: pp. 51–59.

[27] Corbellini, A.; Mateos, C.; et al. Persisting big-data: The NoSQL landscape. *Information Systems*, volume 63, 2017: pp. 1–23.

[28] Neeraj, N. *Mastering Apache Cassandra*. Packt Publishing Ltd, 2013.

[29] Gormley, C.; Tong, Z. *Elasticsearch: The Definitive Guide: A Distributed Real-Time Search and Analytics Engine.* " O'Reilly Media, Inc.", 2015.

[30] Truică, C. O.; Boicea, A.; et al. CRUD operations in MongoDB. In *Proceedings of the 2013 international Conference on Advanced Computer Science and Electronics Information, Ed. Atlantis Press*, 2013.

[31] Vora, M. N. Hadoop-HBase for large-scale data. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 1, IEEE, 2011, pp. 601–605.

[32] Carter, E. 2018 Docker Usage Report. Accessed: 2019-05-05. Available from: `https://sysdig.com/blog/2018-docker-usage-report/`

[33] Boettiger, C. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, volume 49, no. 1, 2015: pp. 71–79.

[34] Gerhards, R. The syslog protocol. Technical report, 2009.

[35] https://logz.io/blog/fluentd logstash/. Fluentd vs. Logstash: A Comparison of Log Collectors. Accessed: 2019-05-05. Available from: `https://sysdig.com/blog/2018-docker-usage-report/`

[36] CPU and Memory Resources in Mesos Marathon with Docker. Accessed: 2019-05-05. Available from: `http://blog.wumuxian1988.com/2016/05/24/mesos-marathon-cpu-memory-constraint-meaning/`

[37] Lin, X.; Lei, L.; et al. A measurement study on linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACM, 2018, pp. 418–429.

[38] Iwaniuk, A.; Popławski, B. CVE-2019-5736: Escape from Docker and Kubernetes containers to root on host. Accessed: 2019-05-05. Available from: `https://blog.dragonsector.pl/2019/02/cve-2019-5736-escape-from-docker-and.html`

Appendix **A**

Acronyms

GUI Graphical user interface

XML Extensible markup language

API Aplication interface

JSON JavaScript Object Notation

BSON Binary JSON

REST Representational State Transfer

URL Uniform Resource Locator

Appendix B

Installation manual

B.1 Zookeeper

The Zookeeper should be installed as the first component. Depending on Linux distribution it can be installed from the repository or downloaded from the website. Then, it has to be configured. There is a sample configuration in folder conf in Zookeeper location. The file that configures Zookeeper should have name zoo.cfg. Zookeeper should be run on at least 3 nodes to provide fault-tolerance.

B.2 Docker

All the nodes that are going to execute user code must have Docker installed. Docker can installed from Docker repository. More information can be found at `https://docs.docker.com/install/`.

B.3 Fluentd

All nodes that will run users' jobs must have Fluentd installed to be able to forward the messages to Kafka. Fluentd can be installed using deb package `https://docs.fluentd.org/v1.0/articles/install-by-deb` as td-agent service. Then, the configuration file has to be copied from the project folder and the Fluentd have to be restarted.

```
copy configuration/td-agent.conf /etc/td-agent/
service td-agent restart
```

The configuration file contains the seed of Kafka brokers. The addresses of the Kafka brokers should be updated according to the cluster configuration.

B.4 Mesos

Mesos can be built from the source. The process of building is described at `http://mesos.apache.org/documentation/latest/building/`.

First, the master instances have to be run. They can be run using startup script ./bin/mesos-master.sh or it can be registered as system service. Mesos have to be configured with the address of Zookeeper.

Then, agents can be run. However, Docker has to be added to containerizers. In addition, the agent can be configured with exact resources it should offer.

```
echo 'docker,mesos'|sudo tee /etc/mesos-slave/containerizers
```

B.5 Kafka

Kafka can be set up as a system service using following commands.

```
wget "http://www-eu.apache.org/dist/kafka/2.2.0/kafka_2.12-2.2.0.tgz"
mkdir /opt/kafka
tar -xvzf kafka_2.12-2.2.0.tgz --directory /opt/kafka
--strip-components 1
rm -rf kafka_2.12-2.2.0.tgz
mkdir /var/lib/kafka
mkdir /var/lib/kafka/data
cp ./configuration/kafka/server.properties /opt/kafka/config
cp ./configuration/kafka/kafka.service /etc/systemd/system/
service kafka status
```

File server.properties contains information about Zookeeper nodes; therefore, it has to updated to reflect the addresses in cluster.

When Kafka runs, the topic can be created.

```
/opt/kafka/bin/kafka-topics.sh --create --topic metronome  \
--replication-factor 2 --zookeeper storm3 --partitions 3
```

B.6 Cassandra

Cassandra can be installed using deb package following the steps from `http://cassandra.apache.org/doc/latest/getting_started/installing.html#`

When Cassandra is installed, it automatically starts with a default configuration. Therefore; it has to be stopped and configuration has to be changed.

```
service cassandra stop
sudo rm -rf /var/lib/cassandra/data/system/*
cp ./configuration/cassandra.yaml /etc/cassandra/
service cassandra start
```

The file cassandra.yaml contains information about other Cassandra that it should use to establish topology. There has to be at least one running server that Cassandra should use.

Established ring topology can be verified using following command.

```
nodetool status
```

Then the keyspaces and tables can be defined. The easiest way is running docker image wtih cqlsh binary.

```
docker run --rm --network host -it cassandra /bin/bash
cqlsh

create keyspace runs with replication =
{'class':'SimpleStrategy', 'replication_factor' : 2};
create table runs.stdout ( source text, timestamp text, \
log text, PRIMARY KEY (source,timestamp) );

create table runs.stderr ( source text, timestamp text,\
log text, PRIMARY KEY (source,timestamp) );

create keyspace runInformation with replication = \
{'class':'SimpleStrategy', 'replication_factor' : 2};

create table runInformation.task (user text, name text,\
command text, image text, submissionTime text, memory int, \
cpus float, disk int ,PRIMARY KEY(user,name) );

create table runInformation.taskRun (user text, name text,
runId int, state text, submissionTime text, finishedAt text, \
PRIMARY KEY((user,name),runId) );
```

B.7 Marathon

Marathon can be simply run on required nodes using Docker. It is recommended to run at least two instances on different nodes to provide fault-tolerance.

```
docker run -d --network host --restart always \
mesosphere/marathon:v1.5.1 \
--master zk://0.0.0.0:2181/mesos \
--zk zk://0.0.0.0:2181/marathon
```

It has to be run with correct Zookeeper addresses.

B.8 Metronome

Metronome can be simply run by submiting configuration JSON to Marathon. JSON has to be configured with correct Zookeeper and Metronome nodes addresses.

```
curl -H "Content-Type: application/json" -X POST \
-d@start-up_commands/marathon_metronome.json \
localhost:8080/v2/apps
```

B.9 CassandraKafkaSubmitter

CassandraKafkaSubmitter is run by Marathon too; therefore, it can be run same way as Metronome. Howver, JSON file has to configured with correct Kafka and Cassandra adresses.

```
curl -H "Content-Type: application/json" -X POST \
-d@start-up_commands/cassandraKafkaConsumer_marathon.json
localhost:8080/v2/apps
```

B.10 TaskSubmitter

TaskSubmitter is run by Marathon too; therefore, it can be run same way as Metronome. Howver, JSON file has to configured with correct Metronome address.

```
curl -H "Content-Type: application/json" -X POST \
-d@start-up_commands/marathon_tasksubmitter.json \
localhost:8080/v2/apps
```

Then the functionality can be verified by submitting any of JSON files in task_submitter_example_runs to the TaskSubmitter endpoint.

Appendix C

Contents of enclosed CD

```
readme.txt ...................... the file with CD contents description
src ................................... the directory of source codes
  DJEES .......................................... DJEES directory
    CassandraKafkaConsumer ............ CassandraKafkaConsumer
    implementation
    TaskSubmitter .................. TaskSubmitter implementation
    configuration ............................ Configuration files
    containers ......................... Prepared Docker containers
    start-up commands ............. Commands for starting DJEES
    task_submitter_example ....................... Jobs examples
  thesis ............. the directory of LaTeX source codes of the thesis
text ........................................ the thesis text directory
  thesis.pdf ........................ the thesis text in PDF format
```