College of Engineering, Design and Physical Sciences,
Electronic and Computer Engineering

Distributed Computing Systems Engineering

Interim Report

Conception and realization of a distributed and automated computer vision pipeline

Michael Watzko
1841795

Supervisor:
Dr. Paul Kyberd

Date of Submission: March 24, 2020

Abstract

The asterisk prevents this file from being labelled as a 'chapter.'

A short summary of what the project is about.

Declaration of Independence

TODO: hehe

Contents

Chapter 1

Introduction

Since the industrial revolution, humans strive for more automation in the industry as well as in the every day life. What was at first a cost saving measurement in factories, now also is a differentiation method for products. A new product must prove a higher level comfort to the customer than the previous generation as well as all the competitors. As such, the ambitions of the industry are focused on increasing the value of their products for the customer.

The automotive industry is one of the prime examples of this. Never was traveling from one place to another as comfortable as nowadays. Aspects like an elegant interior design, comfortable seats, air conditioning, entertainment systems and safety measurements need to be considered by car manufacturers to be competitive these days. The next luxury enhancement will be the autonomously driving vehicle. No longer shall the owner of a car steer it, but instead the car becomes his or hers personal chauffeur, driving the optimal route, the most comfortable way and being more reliable and safer than any human ever could.

The reason, autonomously driving cars are not common already, is their big complexity increase. Compared to already established technologies like parking assistants, entertainment systems or more efficient engine controllers, letting a computer reliably understand a certain traffic situation requires masses of input data and complex algorithms to process. As such, the problem itself becomes

massive and cannot be solved that easily. So the industry has no choice than to divide this into many small pieces and work out solutions step by step.

The MEC-View research project explorers one such step: whether and how to include external, steady mounted sensors in the decision finding process for partially autonomous vehicles in situations where onboard sensors are insufficient. To not disrupt traffic flow with non-human behavior, one needs to study and thereby watch human traffic. Automatically analyzing traffic from video footage requires a lot of computation power and can be further optimized by specialized hardware such as GPUs[1].

This thesis will conceptualize and realize a distributed and automated computer vision pipeline which is in this case used to analyzes traffic flow within video footage. Compared to an existing but highly manual workflow, the new system shall help to utilize the available hardware more efficiently by reducing idle times. Stage transitions and basic scheduling shall be automated to allow a user to plan and execute multiple projects ahead of time and in parallel.

[1]Graphics Processing Units

1.1 MEC-View

The MEC-View research project[1] - funded by the German Federal Ministry for Economic Affairs and Energy - aims to supplement the field of view of automated driving cars with road-side sensor data using 5G mobile communication. The sensor information is merged into an environment model on the so-called Mobile Edge Computing (MEC) server. This server is directly attached to the radio station to ensure low latency environment model updates.

The project is tested at an intersection in Ulm, Germany. Currently, there are 15 lidar and video sensors installed. Those sensors send their detections to the (MEC) server. A fusion-algorithm merges those detections into one environment model and sends it back to the (MEC) server and to the automated cars.

Additionally, general traffic flow is analyzed to learn about movement patterns. To do so, 4k video data is captured by an air drone from real world cross roads - not limited to the intersection in Ulm. On each frame of such a recording, cars are detected with a neuronal network. Detected cars are tracked throughout the video to compute the movement speed and position in time of each car. In an analysis of all vehicles, hot-spots of high and low traffic flow can be determined.

This thesis conceptualizes and implements a distributed and automated computer vision pipeline to increase the efficiency in video analysis management and execution. It is of concern for this thesis on how to retrieve the footage or what is further archived with the results of evaluated video footage. The focus is on utilizing available hardware resources to manage multiple projects simultaneously and to reduce the idle time of relevant hardware by slow human reaction and availability times. TODO: reaction time: like its done but not noticed for another x mins/hours, availability: work hours

Chapter 2

Aims and Objectives

This chapter will discuss the program which shall be implemented. To do so, the problem to solve must be understood. To gather requirements and understand the technical hurdles to overcome, this chapter is split into two sections. First, a rough glance over the current workflow is given, which is followed by a more detailed description for the desired workflow.

2.1 Current Workflow

Currently, to analyze a video for the trajectories of recorded vehicles, the following steps are executed manually:

1. Upload the input video to a new directory on the GPU server
2. Execute a shell script with the video as input file and let it run (hours to days) until completed. The shell script invokes a Java Program - called TrackerApplication - with parameters on what to do with the input file and additional parameters.
3. The intermediate result with raw detection results is downloaded to the local machine and opened for inspection. If the detection error is too high,

the camera tracking has a drift or other disruptions are visible, the previous step is redone with adjusted parameters.

4. Upload the video and intermediate result to a generic computing server and run data cleanup and analysis. This is achieved with the same Java Program as in step 2, but with different stage environment parameters.

5. Download the results, recheck for consistency or obvious abnormalities. Depending on the result, redo step 2 or 4 with adjusted parameters again.

6. Depending on the assignment, steps 4 and 5 are repeated to incrementally accumulate all output data (such as statistics, diagrams and so on).

Because all those steps are done manually, the user needs to check for errors by oneself. Also, if a execution is finished or has failed early, there could be hours wasted until noticed, if the check intervals are too far apart, such as during nights or weekends.

2.2 Desired Workflow

The desired workflow shall be supported through a TODO: rich user interface. This user interface shall provide an overview of all active projects and their current state, such as running computation, awaiting user input, failed or succeeded.

To create a new project, a predefined pipeline definition shall be selected as well as a name chosen. Because only a handful of different pipeline definitions are expected, the creation of such does not need to happen through the user interface. Instead, it is acceptable to have to manually edit a configuration file in such rare circumstances.

Once a project is created, the user wants to select the path to the input video. This file has to be been uploaded to a global resource pool at this point. The upload and download of files shall therefore also be possible through the user interface. Because a video is usually recorded in 4k (3840 x 2160 pixels), encoded

with H.264 and up to 20 minutes long, the upload must be capable of handling files which are tens of gigabytes large.

Once a pipeline is started, it shall execute the stages on the most fitting server node until finished, failed or a user input is required. Throughout, the logs of the current and previous stage shall be accessible as well as uploading or downloading files from the current or previous stages workspace. In addition to the pipeline pausing itself for user input, the user shall be able to request the pipeline to pause after the current stage at any moment. When resuming the pipeline, the user might want to overwrite the starting point to, for example, redo the latest stage.

Mechanisms for fault tolerance shall detect unexpected program errors or failures of server nodes. Server nodes shall be easily installed and added to the existing network of server nodes. Each server node might provide additional hardware (such as GPUs), which shall be detected and provided.

For the ease of installation and binary distribution, Docker Images shall be used for running the Java Program for analyzing the videos as well the to be implemented management software.

2.3 Deliverable Requirements

TODO: .delete? From the desired workflow, the following requirements can be extracted (shortened and incomplete due to early project stage):

- TODO: Rich user interface
- Storage management for global resource files as well as stage based workspaces
- Pipeline definition through configuration files
- Handling of multiple projects with independent progress and environment
- Reflecting the correct project state (running, failed, succeeded, paused)
- Log accumulation and archiving

- Accepting user input to update environment variables, resuming and pausing projects as well as uploading and downloading files into or from the global resource pool or a stages workspace.
- Assigning starting stages to the most fitting server node
- Detecting program errors (in a stage execution)
- Cope with server node failures
- Docker Image creation for the Java Binary as well as the program implementation, preferred in an automated fashion.

Chapter 3

State of the art

In this chapter, programs solving similar problems, as described in the desired workflow, or dealing with a subset of the problem are looked into. The reason for this is to use well established or suitable programs as middle-ware to reduce implementation overhead. Where this is not possible, one might be able to gather ideas and learn about proven strategies to use or pitfalls to avoid while implementing custom solutions.

3.1 Similar solutions

This sections focuses on programs trying to provide somewhat similar workflows.

3.1.1 Hadoop MapReduce

For big data transformation, Hadoop MapReduce[2] is well known. With MapReduce, the input data is split into blocks of data and distributed onto Mappers. Mappers then apply the business logic and output intermediate results in form of key/value pairs. After shuffling, the Reduce stage will combine values from key/-value pairs with the same key. In the final output, each key is unique accompanied with a value.

This strategy has proven to be very powerful to process large amount input

data because Mappers and Reducers can work independently on their data-sets and therefore scale very well when adding further instances.

If the implementation were to be based on Hadoop MapReduce to achieve the desired workflow, it could be done like the following:

- Each video is split into many frames and each frame is applied to a Mapper
- A Mapper tries to detect all vehicles on a frame and outputs their position, orientation, size and so on
- The Reducer then tries to link the detections of a vehicle through multiple frames
- The final result would be a set of detections and therefore all positions for each vehicle in the video

But at the moment, this approach seems to be unfitting due to at least the following reasons:

1. It is not always trivial to reasonable link the detections of a vehicle. For example, a vehicle can be hidden behind a tree for a few frames until visible again. In addition, MapReduce requires the combination to be performed per common key. Until one is trying to link the detections of multiple frames, there is no common identifier that could easily be used as key. The position of a moving vehicle cannot be used as key, neither can the color or size, because of the noise of the camera, deviation in detection output and perspective distortions. The current implementation of the TrackerApplication is archiving this by finding similarities between detections, but for the Mapper it would be required to express this as a deterministic key.
2. MapReduce is great in combining many machines to solve a big computational problem. But at the moment, this is neither a desired nor given condition. At the moment, there are a handful of very powerful workstations

with specialized hardware. Therefore it is perfectly acceptable and sometimes required, when each workstation works through a complete video at a given time instead.

3.1.2 Build Pipelines

Build pipelines such as GitLab[3] and Jenkins[4] can also distribute the execution of stages onto other server nodes. In a common use-case, such build pipelines are used to build binaries out of source code, after a new commit into a SCM[1] repository was made. At IT-Designers GmbH GitLab as well as Jenkins are commonly used for scenarios exactly like this. A pipeline definition in GitLab CI/CD [5] or in a Jenkinsfile [6] describe stages and commands to execute. Each stage can be hosted on another node and be executed sequential or in parallel to each other.

Although this seems to be quite fitting for the desired workflow, there are two issues. First of all, such a pipeline does not involve any user input besides an optional manual start invocation. The result is then determined based on the state of the input repository. Second, such a pipeline is designed to determine the output (usually by compiling) whereas each run is independent from the previous and a repeated run shall provide the same result as the previous did. Usually, a new run is only caused by a change of the input data. However, the desired workflow differs in this aspects. A redo of a stage can depend on the result of the previous stage, for example, if the results are poor or the the stage failed. Instead of having multiple complete pipeline runs per project, the desired workflow uses a pipeline definition as base for which the order can be changed. Also, intermediate results need to influence further stages, even if repeated.

[1]Source Code Management

3.1.3 Camunda

Camunda[7] calls itself a "Rich Business Process Management tool" and allows the user to easily create new pipelines by combining existing tasks with many triggers and custom transitions. Camunda is focused upon visualizing the flow and tracking the data through a pipeline. The Camundas Process Engine[8] also allows user intervention between tasks.

One of the main supporting reason for it Camunda is the out of the box rich graphical user interface for process definition and interaction. Through its API[9], Camunda also allows custom external workers to execute a task. But it misses the capability to control which task shall be processed on which worker node which is required by the desired workflow. It does also not provide any concept on how to allocate and distribute resources. The user interface - while being rich overall - is quite rudimentary when it is about configuring tasks and would therefore require custom plugins to be developed for more advanced user interactions.

Camunda is also not designed to reorder stages or insert user interactions at seemingly random fashion. The user itself is considered more as a worker that gets some request, "executes" this externally and finally marks the request as accepted or declined. Mapping this to the desired workflow does not feel intuitive. Finally, there is also no overview of task executors, no centralized log accumulation and no file up- or download for global project resources.

3.1.4 Nomad

Nomad[10] by HashiCorp is a tool to deploy, manage and monitor containers, whereas each job is executed in its own container. It provides a rich REST API and can consider hardware constraints on job submissions. Compared to Kubernetes[11], which is similar but more focused on scaling containers to an externally applied load, it is very lightweight. It is also available in many Linux software repositories - such as for Debian - which makes the installation very easy.

Because there were no grave disadvantages found (depending on a third party

library can always be considered be a disadvantage for flexibility, error-pronous and limit functionality) Nomad is being considered as a middle-ware to manage and deploy stages. Others[12] seem to be using Nomad to manage and deploy containers for similar reasons. Nonetheless, further testing and prototyping will be required for a final decision.

3.1.5 dCache

"The goal of this project is to provide a system for storing and retrieving huge amounts of data, distributed among a large number of heterogenous server nodes, under a single virtual filesystem tree with a variety of standard access methods"[13]. dCache seems to be able to solve the storage access and distribution concern for the stages and sever nodes. When using dCache, one could store the global resources distributed between the server nodes. Built-in replication would prevent access loss on a node or network failure and an export through NFS[2] allows easy access for Linux based systems[14].

But the installation is complex and requires many services to be setup correctly, such as postgresql and many internal services such as zookeeper, admin, poolmanager, spacemanager, pnfsmanager, cleaner, gplazma, pinmanager, topo, info and nfs. The documentation is also rather outdated and incomplete which meant, early tests with a prototype setup took days to setup and behaved rather unstable (probably due to a wrong configuration). It is to be seen, whether such an complex and heavy system is actually required or if there are feasible alternatives.

3.1.6 Further mentions

The following list shall acknowledge programs that behave similar to the previously mentioned strategies. Programs that are listed here, were looked into, but not in-depth because miss-fits were detected early on (listed in no specific order):

[2]Network File System

- **Quartz**[15] is a Java based program to schedule jobs. Instead of doing so by using input, Quartz executes programs through a timetable and in certain intervals.

- **Luigi**[16] also executes pipelines with stages and is written in python. The advertised advantage is to define the pipeline directly in python code. But, this is at the same time the only way to define pipelines which contradictions with the existing Java TrackerApplication implementation.

- **Calery**[17] is focused on task execution through message passing and is written in Python. Intermediate results are expected to be transmitted through messages. Because there is no storage strategy and python adapter-code would have been required, Calery was dismissed.

- **IBM InfoSphere**[18] provides similar to Camunda a rich graphical user interface but for data transfer. Dismissed due to commercial nature.

- **qsub**[19][20] is a CLI[3] used in HPC to submit jobs onto a cluster or grid. Dismissed due to an expected high setup overhead, non-required multi-user nature and the fact, that it only provides a way to submit jobs.

- **CSCS**[21] High Throughput Scheduler (GREASY). Dismissed for similar reasons as qsub, although it is more light weight and hardware agnostic (it can consider CUDA/GPU requirements).

- **zsync**[22], similar to rsync, is a file transfer program. Zsync allows to only transfer new parts when a file that shall be copied already exists in an older version on the target. This tool might be useful when implementing a custom resource distribution strategy is required.

- **OpenIO**[23] provides a distributed file system, is already provided as Docker image and provides a simple to use CLI. Because the NFS export is only

[3]Command Line Interface

available through a paid subscription plan, it was dismissed from further investigation.

- **SeaweedFS**[24] provides a scalable and distributed file system. The most interesting aspects are that it is rack-aware as well as natively supports external storage such as Amazon S3. When adding server nodes from the cloud this could allow all nodes to access the same file system while using rack-aware replication to reduce bandwidth usage and latency. A local test also proved that it is easy to setup, but because it cannot hot-swap nodes and was not able to recover when the seeweed master node became unreachable it was dismissed.

- **Alluxio**[25] provides a distributed file system but was dismissed because it itself requires a centralized file system for the master and its fallback instances

- **GlusterFS**[26] is another tool to provide a distributed file system with replication. It was bought by IBM but is nonetheless available through the software repository of many Linux distributions such as Debian. A local test showed that the setup is very easy and no adjustments of configuration files are required. However, the replication mechanism requires that an integer multiple of nodes of the replica value are assigned to the file system. This makes GlusterFS hard to use in a scenario, where adding and removing nodes are expected to happen frequently. It was therefore dismissed.

3.2 Docker Integration

As describe before (see section 2.2), for easy deployment, the implementation as well as the stages shall be executed inside Docker[27] containers. This allows easier isolation of the stages and workspaces from each other and other host programs. Because one needs to communicate with the Docker daemon, this in-

creases the complexity for the implementation. But by using third party libraries, the increase in complexity can be limited.

Chapter 4

Schedule

The following figure shows schedule for further and past work:

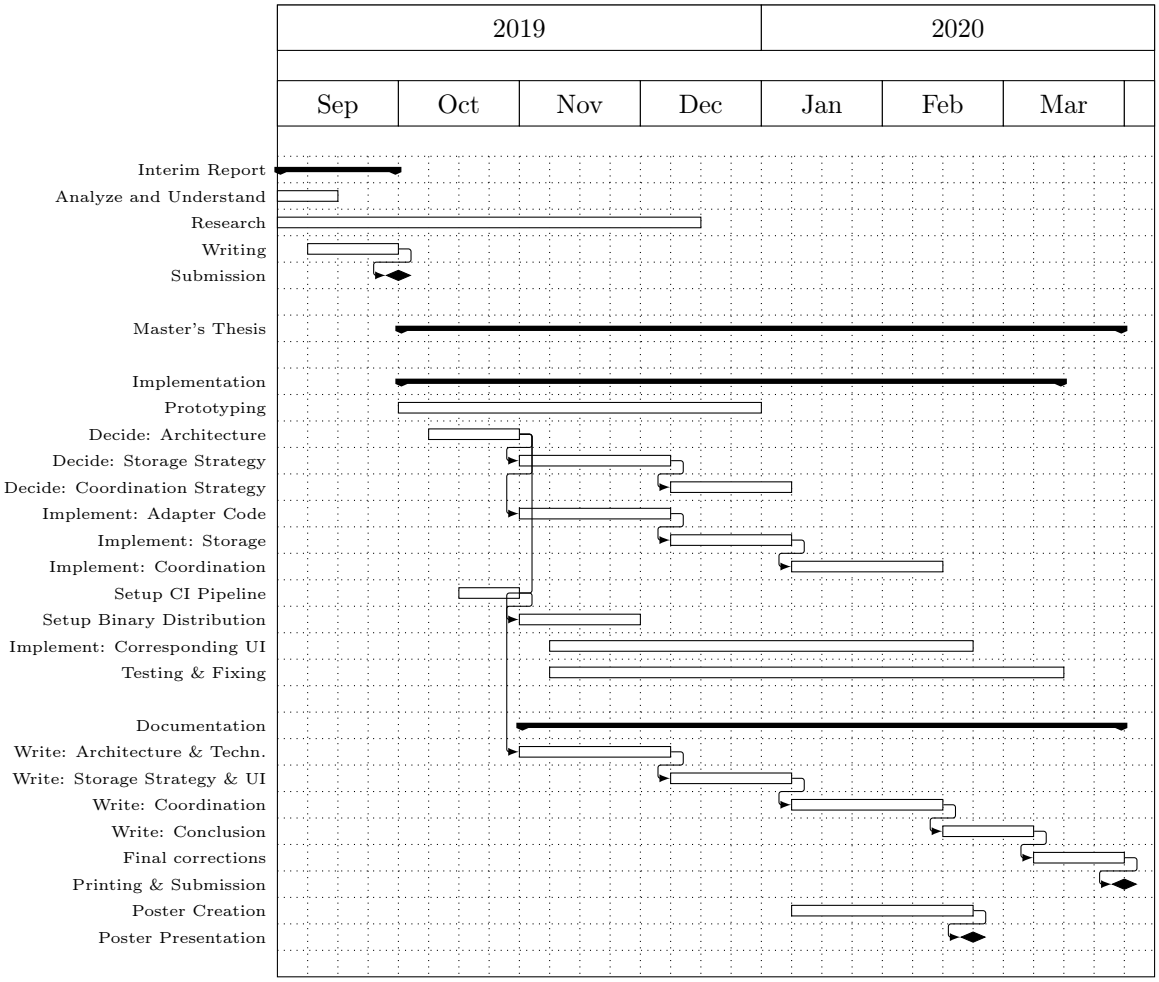


Figure 4.1: Time schedule

Chapter 5

Introducing Winslow

The program that shall implement the requested features listed by **??** is being called Winslow. This name refers to Frederick Winslow Taylor who was an American mechanical engineer and one of the first management consultants in the 19th century[.] Both strive to make people's work more efficient.

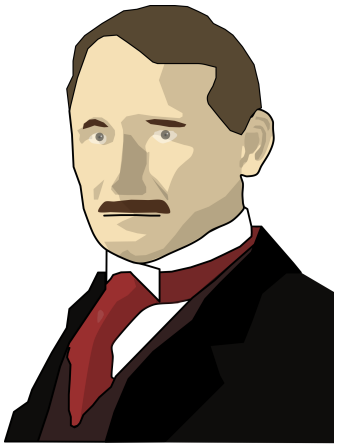


Figure 5.1: Winslow Logo

5.1 Common Terminology

This chapter explains the meaning of words and terminology used later on. It is crucial that every reader has the same understanding for the words used so that there are no wrong assumptions, expectations or surprises.

The root of a work item is called project. A project will usually refer to one video footage that shall be processed. Each project has its exclusive directory for input, output and intermediate files, called workspace. Furthermore a project as an author and possibly participants, that can help in steering and monitoring the processing. To do so, there is always a pipeline assigned to a project.

A pipeline consists of at least one but usually multiple stages that can at least be processed linearly - one after the other. Furthermore, a pipeline can define environment variables that valid for all stages within the stage.

A stage is the smallest work unit that can be executed. A docker imagesection 6.1 is specified as execution environment as well as further stage specific environment variables, command line arguments and hardware requirements (such as CPU cores, GPUs and minimum available RAM). This enables a stage to use a common image as well as to specify very precisely how to process the data.

In addition, stages and pipelines can be distinguished in a active, running or completed stage or pipeline and a definition. A stage definition specifies the above mentioned presets - allowing the user to adjust details just before submission - to create multiple stages from, while a running or complete stage has all information of what is or was executed and allows no further alteration. For pipeline definitions this is similar, a pipeline definition can be used to specify a common order of execution. Once assigned to a project, the project can freely adjust and change its instance of the pipeline.

TODO: execution node

TODO: winslow instance

Chapter 6

Fundamentals

This chapter will explain and discuss fundamental knowledge and technologies used in this thesis. It is required to understand their basic principles for the following chapters.

6.1 Docker

Docker is the name of a software that combines isolation technologies (subsection 6.1.1) and a rich ecosystem (subsection 6.1.2) to provide and execute third party software in virtual environments. Docker aims are to increase security and to simplify installation as well as maintenance of applications. Docker Swarm[29] and Kubernetes[30] use these fundamental features to scale applications, services and microservices in the cloud.

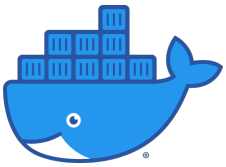


Figure 6.1: Official Docker “Moby” Logo[28]

6.1.1 Technology

Docker uses so called images to package and transport binaries with all their required libraries and configuration files as a read-only archive to the destination

system. TODO: layers? base image, less download, diffs. Instead of spawning a new process for a binary in the host environment, the image is used to create a new virtual environment - the so called container. Because the image includes all required libraries (in the correct version!), it is possible to isolate the virtual environment which makes the host system invisible to any process inside. Changes to files within containers are also stored in separately[1], which allows an image to be used by multiple containers at once. Privileges, resource limitations and storage configurations can also be specified when creating a new container. Processes inside containers are unable to see other processes or files that are not part of or assigned to the their container[2].

In contrary to a hypervisor, docker is archiving this without running additional virtual machines for each container. Instead of running on virtualized operating systems, container processes share the host kernel. In order to do so, the host operating system needs to support additional isolation mechanisms. At the time of writing this, only the Linux Kernel is capable to separate processes, network interfaces, interprocess communications, filesystem mounts and the timesharing systems by namespaces. By configuring these namespaces, Docker is capable to isolate containers into virtual environments. Furthermore, control groups can be used to limit and constraint access to hardware resources. [31]

These approaches allow containers to run with very little overhead in comparison to running the application directly on the host. Containerization increases security by limiting what an application sees and is able to interact with, decreases maintenance overhead because of no additional operation systems to maintain and allows to run multiple instances of the same application besides each other with independent configurations and environments.

[1] TODO: overlayfs, similar to differentials in backups

[2] This is the default behavior. It is possible to manually lift or modify many boundaries Docker enforces for containers on default.

6.1.2 Architecture and Ecosystem

TODO: .??

6.1.3 Self hosted registry

6.1.4 Deployment

Dockerffile: creating a docker image

6.1.5 Something something ref cloud

docker is so popular that even microsoft is trying to support it, although most images require a Linux kernel - therefore microsoft introduced (WSL)

Paravirtualisation?

https://www.monkeyvault.net/docker-vs-virtualization/

Instead it uses built-in Linux Kernel containment features like CGroups, Namespaces, UnionFS, chroot (more on these later) to run applications in virtual environments. Those virtual environments - called Docker containers, have separate user lists, file systems or network devices.

6.2 Network File System

When distributing workload onto different machines, accessing input and output files becomes another concern which is not present when computation and storage reside on the same physical machine. In theory, one could use portable mediums such as USB-Sticks, CD-ROMs or external HDDs, but in reality, this becomes to tidies very fast. More advanced users might be able to take advantage of a common network connection between the computation and storage nodes to copy files and directories from and to the required places. This strategy - still being tidies - surfaces another issue: having multiple copies requires careful version management and additional storage space. One certainly would not want to

continue computation on outdated files or have too many copies of the same files when the required next file no longer fits on the computation node.

The solution is to use what is called network file system. To programs they seem to be just another local directory hierarchy, but in reality, the files might be located on another or multiple other machines.

6.2.1 Basics: file system?

To understand how a network file system works, it is necessary to first understand the concept of a file system. The following paragraph will focus on EXT(v4), which an open source file system used by many Linux servers, but the main concepts also apply to proprietary file systems such as NTFS from Microsoft.

The EXT (extended file system) family is based upon the concept of interlinked Inodes and TODO: data blocks. Data blocks describe large addressable binary objects that are linked by Inodes when allocated. Inodes carry metadata to regulate access privileges and to represent a directory hierarchy. To do so, the Inode has a type attribute that can declare it as a directory. The data associated to the Inode is then interpreted as table of names and Inode references to entries of the directory the Inode represents. In file mode, the Inode contains the content of the file. For every Inode there is a certain amount of storage assigned inline allowing the Inode to store small files or directories in itself without referencing others and improves access times.

TODO: show (simplified) Inode+fields

Furthermore, EXTv4 is a so called journaling file system. TODO: interlink with DB ACID event system

TODO: cite: link online man page ext4

6.2.2 POSIX Streams API

`https://www.gnu.org/software/libc/manual/html_node/I_002f0-on-Streams.html`

The POSIX TODO: (Portable Operating System Interface) defines a basic set of functions to work with files. It is a standard that was standardized by IEE TODO: number to define common functions across operating systems. Simplified, and without caring about functions that create directories and links, move files or change attributes, the very basic functions to open and close streams are specified as the following C functions:

- `FILE* fopen(const char *filename, const char *opentype)` to open a stream to the file which the path `filename` refers to. The `opentype` specifies whether the file shall be created, overwritten, opened in read-only, write-only or append mode.
- `int fclose(FILE *stream)` to close the given `stream`
- `size_t fread(void *data, size_t size, size_t count, FILE *stream)` to read `count`-number of objects of size `size` from `stream` and to the `data` memory location.
- `size_t fwrite(const void* data, size_t size, size_t count, FILE *stream)` to write `count`-number of objects of size `size` read from the `data` memory location to the `stream`
- `int fflush(FILE* stream)` to write buffered and not yet committed data to the underlying destination (filesystem).

The operating system usually delegate these calls to the corresponding file system driver for a given path. The Linux kernel requires the filesystem to implement th following (simplified and incomplete) function listing: `https://www.tldp.org/LDP/khg/HyperNews/get/fs/vfstour.html`

- `int open(struct inode*, struct file*)` TODO: .
- `int read(struct inode*, struct file*, char*, int)` TODO: .

- `int write(struct inode*, struct file*, const char*, int)` TODO: .
- `void release(struct inode*, struct file*)` TODO: .
- `int fsync(struct inode*, struct file*)` TODO: .

A network file system needs to implement this interface and instead of writing and reading to and from a local device, it needs to send the requests to a remote storage server or to the storage cluster.

6.2.3 Centralized Network Storage

NFS, SMB

6.2.4 Decentralized

ceph, glusterfs, hadoop

TODO: basic words on UDP and TCP?

6.3 HTTP and REST

6.4 TypeScript and Angular

6.5 yaml?

6.6 Agile development

Chapter 7

TODO: Aims and Objectives

This chapter works out the desired capabilities of the software and then lists the resulting requirements. Requirements help to keep track of whether the software covers all customer needs and wishes. They also help during development to keep track of the progress and estimate the time required to implement the remaining requirements.

TODO: not complete, not anti-agile: it specifies the basic needs and expectations at the beginning

7.1 Top Level Requirements

7.2 Requirements

7.2.1 Managing Pipelines and Projects

- **Requirement 1110: Define a new Pipeline**

  The user must be able to create a new pipeline definition. Only valid definitions must be accepted. A valid pipeline definition has a name and must contain at least one stage definition.

- **Requirement 1120: Update an existing Pipeline**

The user must be able to see and modify an existing pipeline definition.

- **Requirement 1130: Delete an existing Pipeline**
  The user must be able to remove an existing pipeline definition. Deleting a pipeline definition must not break and therefore must not prevent associated projects from further execution.

- **Requirement 1210: Create a new Project**
  The user must be able to create a new project. Only valid projects must be accepted. A valid project has a name and must be using an existing pipeline definition.

- **Requirement 1220: Update the Pipeline of a Project**
  The user must be able to change the pipeline definition an existing project is based on.

- **Requirement 1230: Update the Name of a Project**
  The user must be able to update the name of a project.

- **Requirement 1240: Updating Tags of a Project**
  The user must be able to add and remove tags to and from an existing project.

- **Requirement 1250: Delete an existing Project**
  The user must be able to delete a Project. Deleting a project must delete all associated files, directories and configuration files.

7.2.2 Managing Resources and Workspaces

TODO: what about res/workspace/init directories

- **Requirement 1310: Upload Files**
  The user must be able to upload files into the scope of a project, so that further stage execution is able to retrieve said file.

- **Requirement 1320: Download Files**

  The user must be able to download files that are within the scope of a project. Said files can be files that were previously uploaded by the user or are results of executed stages.

- **Requirement 1340: List Files**

  The user must be able to retrieve a list of files associated with a project.

7.2.3 Managing and Monitoring Executions

- **Requirement 1410: Start a Stage**

  The user must be able to start a new Stage for a project. Any Stage defined in the associated Pipeline Definition is considered a valid choice. The user shall be able to choose whether the following Stages shall be executed automatically or the pipeline shall be paused upon completion.

- **Requirement 1420: Pause a Pipeline**

  The user must be able to mark a running Pipeline of a Project to be paused before executing the next Stage.

- **Requirement 1430: Resume a Pipeline**

  The user must be able to resume paused Pipelines.

- **Requirement 1440: Abort a running Stage**

  The user must be able to commit an abort request for a running Stage. An aborted Stage shall be considered failed and further Stage execution shall be paused.

- **Requirement 1450: Inspect logs of a Stage**

  The user must be able to see log messages produced by a selected Stage as well as to that stage associated system events.

- **Requirement 1460: Inspect state a Stage**

  The user must be able to retrieve the state ('RUNNING', 'PAUSED', 'SUCCEEDED', 'FAILED') for all stages of a project.

7.2.4 Monitoring Nodes

- **Requirement 1510: Monitor CPU usage**

  The user must be able to retrieve the CPU utilization of all known nodes.

- **Requirement 1520: Monitor RAM usage**

  The user must be able to retrieve the RAM utilization of all known nodes.

- **Requirement 1520: Monitor Network IO**

  The user must be able to retrieve the Network IO utilization of all known nodes.

7.2.5 Derived Requirements

Requirements that are derived by looking at other requirements.

TODO: functional vs nonfunctional

Die hier gelisteten funktionalen Anforderungen beschreiben das gewünschte Verhalten des Systems [32, p. 155].

Nichtfunktionale Anforderungen zeigen im Gegensatz zu funktionalen Anforderungen Rah- menbedingungen bei der Umsetzung des Systems auf [32, p. 155].

Chapter 8

System analysis

This chapter is about analyzing the to be implemented system more detailed, separating concernsTODO: cite? and starting to evaluate solutions in an architectural viewpoint. It is important to find an architecture, that is able to fulfill all requirements and is able to be evolved on for upcoming needs. In some areas where this is foreseeable, preparations can be implemented, whereas in other regions this is not possible. When working on a architecture, the SOLIDTODO: . and YAGNITODO: . principles can help to evaluate the stability and expandabilityTODO: can it?.

8.1 System Architecture

In Figure 8.1 a high level overview of the architecture is shown. It shows various services and regions with internal and external communication channels. The architecture follows the "Onion Architecture Pattern"TODO: cite, which is also indicated by the layer numbers and coloring (inner layers are darker).

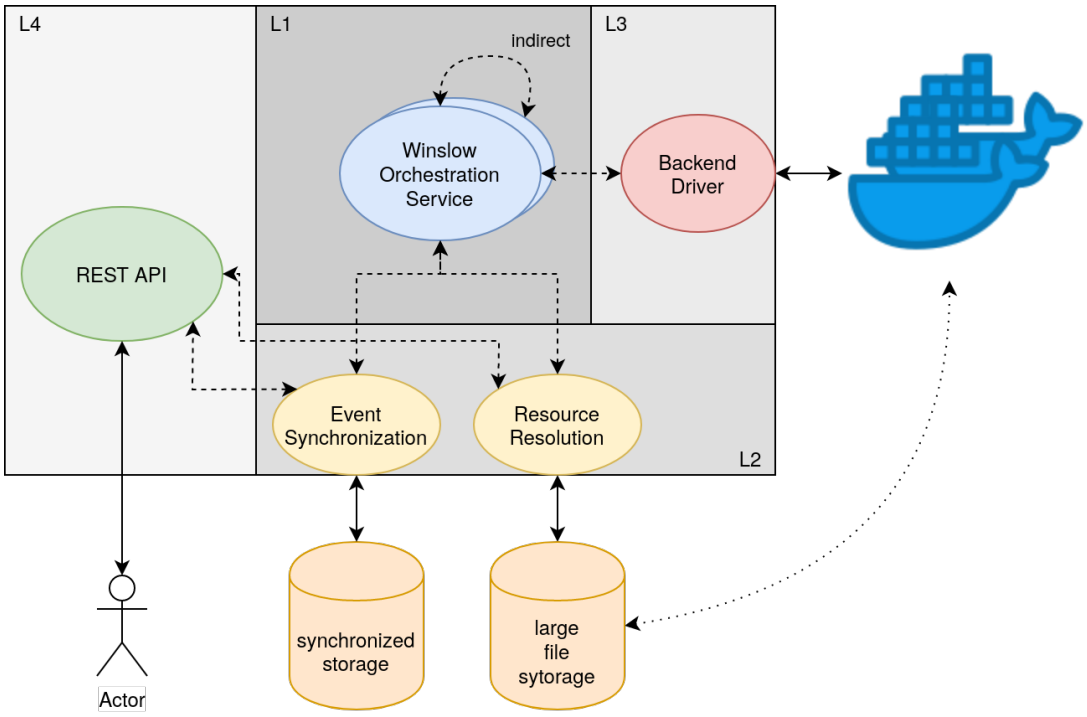


Figure 8.1: Architecture overview of a Winslow instance

The meaning of the layers and their services will be explained in the following sub-sections.

8.1.1 Layer 1: Orchestration Service

This layer is all about the fundamental business logic: when to schedule, start and fail which stage of what project, while watching the hardware utilization. This level is also responsible to coordinate these actions between all Winslow instances, so that there is no duplicate, missing or undetected node failure.

The following requirements are therefore implemented here: TODO: .?

8.1.2 Layer 2: Events and Resources

For the first layer to communicate with the environment, such as reading configurations, project files or cooperation commands, this layer is essential. It provides two important services: a synchronized event bus and a large file resource management. The first is used to communicate with the other Winslow instances, while the latter is required for a stage execution. Although list here as two separate services, they do not necessarily need to be backed up by two storages. To provide a synchronized stream of events, it is required that the underlying storage provides some sort of synchronization primitives (see TODO: ref stream api / files?). The large file storage has no such requirement, as this is ensured by Winslow's stage synchronization and thus a eventual consistentTODO: cite storage could be used here.

While knowing the different requirements, because of the time constraint of this thesis, a common NFS storage will be usedTODO: autoref.

8.1.3 Layer 3: Backend Driver

This layer is responsible to bind a remote or external service to the reach of Winslow. Once the orchestration service decided to run a stage, collected all environment variables and prepared the workspace, the backend driver is instructed to start a certain image with said prepared environment. By separating this task to its own layer - and service in this case - it will be easier to change from Docker to another platform if necessary. A reason for this could be the change of needs, the appearance of a platform that is fulfilling the needs better or the disappearance of the currently used platform. The latter sounds not that common at first, but the recent partial acquisition of Docker Inc by Mirantis[33] proves that even very popular third party software is not going to be around for all eternity. By moving the driver implementation into its own layer, but leaving the interface

definition in layer 1, this also complies to the “Dependency Inversion Principle”. TODO: .

While the backend driver does not access any storage itself, the stages need to read and write to the large file storage. In the case of the Docker implementation with an NFS share, the access from within the executed stage can be guarded to limit read and write access to only the required working directories. This is discussed more in-depth in TODO: autoref.

8.1.4 Layer 4: Client Communication

The final layer in this architecture is the client communication layer. This service is not crucial for the actual execution and may be disabled on Winslow instances that shall not response to a user input themselves. The REST APITODO: autoref rest provides sites that can be called from the static AngularTODO: autref angular content to upload or download files, to control stage execution and to monitor usage and logs. Removing or disabling this layer is not stopping the Winslow

8.2 Communication / message analysis ?

For the communication described in subsection 8.1.2 messages need to be defined. Because this is a multi instance system which can suffer partial failure, all operations that are non-instantaneous require to have a timeout, so that a failure is detectable. Without a failure detection, the end of an operation might not be signaled, which could potentially block further operations for ever.

To account for transmission delay and clock offsets an additional time padding is granted. TODO: .

TODO: all messages are broadcast because of bus

TODO: all messages have an id

TODO: to detailed here? move detailed description to system design?

- **LOCK**: Locking a named resource or operation. A lock is exclusive to the issuer and can only be granted if no other lock of the subject or parts of it exist.
- **EXTEND**: Extending an ongoing operation timeout. This can be used to signal, that a lock is required although its timeout is closing by, in which case the point in time the timeout is reached is pushed back. The issuer must be the same node as the one that issued the **LOCK** previously.
- **RELEASE**: An operation or resource is no longer required. The issuer must be the same node as the one that issued the **LOCK** previously.
- **KILL**: Non-gracefully stop an operation or destroy a lock. This signals that an operation on its own or associated to a lock shall be stopped immediately and that the lock shall be released. At the moment, this is only used whenever a user wants cancel a stage execution early. More details in TODO: autoref.
- **ELECTION_START**: Signals that the need to execute a stage was detected. The project, pipeline and stage execution id this message is about is included.
- **ELECTION_PARTICIPATE**: Signals that the issuer node is capable of executing the stage. It also includes a scoring for the affinity and aversion of executing the stageTODO: cite on the issuer node. This message also includes the id to the **ELECTION_START** it refers to.
- **ELECTION_STOP**: Signals that the election has ended. The participator with the best affinity and aversion scoring is now allowed to start to execute the stage. More detail in TODO: auref. The issuer must be the same node as the one that issued the **ELECTION_START** previously.

Chapter 9

System design

TODO: divide and conquer?

Designing a system requires deep understanding of the TODO: problem space/use cases/task definition to then create a solution. The better the knowledge of existing system is, the more easier it is to take advantage of established solutions and evade known pitfalls. This chapter will concentrate on the big TODO: headaches by analyzing and comparing them to existing or similar solutions and develop a solution with the help of the gathered knowledge.

9.1 Execution Management

9.1.1 Remote Execution

In this approach, the job is executed on a remote machine and not on the same node which has the responsibility in managing the execution. Continuous Integration (CI) platform Jenkins[4] as well as GitLab[3] do offer this approach. The so called slave node (Jenkins) or runner (GitLab) is accessed by the CI through a common interface (SSH in these cases) to start the job execution. Jenkins and GitLab even copy a custom binary onto the slave node, that is then managing the execution on the remote machine locally. This is due to the complexity in executing the job for a given configuration as well as being able to continue the

execution on disconnects or maintenance reboots of the master machine.

In this scenario, the CI instance requires and stores login credentials for every remote machine to be able to login whenever needed. The system administrator therefore has to create a new user account on the remote machine, install required programs and environments (bot Jenkins and GitLab require an JRE[1]). In case of an security breach on the CI instance, the attacker is also able to login on all remote nodes and execute arbitrary programs as well TODO: there is probably a great article about an issue like this!?.

TODO: WRONG, on GitLab YOU need to intall/setup the runner

GitLab follows another approach, the system administrator has to manually install the GitLab Runner[2] on a slave node and is then able to add this runner to a Project. The security risk is somewhat similar to Jenkins: once an attacker has access to the CI instance, arbitrary code can be uploaded and executed on the remote machines. TODO: but no ssh login .... until first binary uploaded

TODO: winslow is more secure because of docker and therefore limited access to the system, no network..?

9.1.2 Local Execution

Executing a job locally means running the job on the same machine as the program that is watching and managing the execution. This has the advantage of having all libraries, tools and resources already present. But in the case of Jenkins and GitLab, this is means, each CI instance is separate from the others. There is no integration between those instance in this configuration. What is missing Jenkins and GitLab here is the capability to decentralize their core task of managing projects, resources and executions.

[1] Java Runtime Environment
[2] TODO: cite

9.1.3 Decision for Winslow

9.2 Communication and Synchronization

For the instances of Winslow, communication and especially synchronization is very important. Without proper coordination, race conditions could cause stages to be started multiple times simultaneously, corrupt workspaces, configuration or project files. While starting too many stage executions are only wasting resources, data corruptions can lead to unrecoverable damage.

Before synchronization can be achieved, communication must be possible. There are multiple ways to exchange data for systems that do not share the same process or machine. The most flexible implementation can be achieved by implementing a custom protocol on a raw TCP socket which allows to exchange blobs[3] between exactly two nodes. For a blob or message to reach all nodes, there needs to be connections to all other nodes TODO: great mafs N*N, N*(N-1) or whatever, a centralized broker (introducing single point of failure), or another connection organization such as a tree structure. Having many connections can stress the hosting system while complex structures require additional development and maintenance overhead in non-problem related domains. To not have to worry about the structure, broadcasting or multicasting blobs to all nodes could be an alternative. But UDP messages are not guaranteed to be delivered and introduce further overhead to detect the need to retransmit a blob. Because of these concerns, the idea of a custom implementation on top of a raw TCP Socket was abandoned, instead alternatives were investigated.

https://docs.microsoft.com/de-de/azure/architecture/microservices/design/interservice-communication

raw TCP (how to connect, centralized? star? tree?) centralized works towards -> centralized master node broadcast? microservice REST event bus? kafka messaging queue mqtt NFS provides filesystem, filesystems are kind of standardized

[3] TODO: Binary Large Objects

https://kafka.apache.org/protocol

Apache Kafka TODO: CITE comes very close in fulfilling all needs for a communication system. It is open source, distributed and is focused on data and event streams. But it introduces a dependency on the project. It also provides functionality that is not needed, like a transaction log and persistence and replayability of all transmitted dataTODO: rly?. Furthermore, because it is an independent service, it also requires its own TCP connection. It is required, that this connection can be established between seemingly random nodes. This adds another constraint to the system: the firewalls in-between the nodes must allow these connections.

Because the communication between the Winslow nodes are primarily for synchronization and not for exchange of large binary data, the previous mentioned scenarios are over fulfilling the need. The most anticipated communication and synchronization mechanic should use and already established communication channel. The only defined requirement yet is a common filesystem, can this be utilized to coordinate stage execution and prevent data corruption by providing basic synchronization between the compute nodes? It can!

TODO: ACID in fundamentals? ACID, which stands for Atomicity, Consistency, Isolation and Durability, describes the needs a database must fulfill so that it does not suffer from side effects or data corruption when executing queries. A series of changes might not be allowed to be intercepted (atomicity), must see and result in consistent data, is not allowed to interfere with simultaneously executed queries and guarantees that once completed successfully, is in fact persistent and will not be lost (durability).

TODO: filesystem is close

TODO: DB principles: ACID

TODO: ...

TODO: requires common a way to exchange messages

TODO: all messages - called events - are published to all nodes

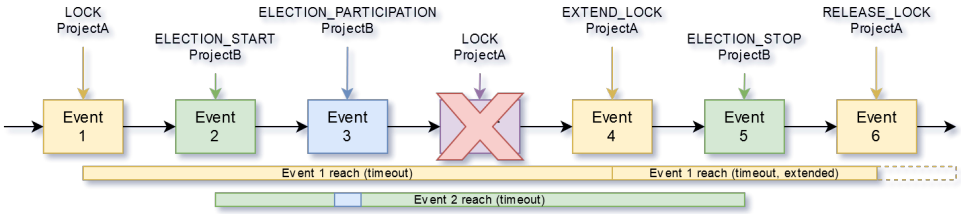


Figure 9.1: Sample event and lock series with lifetimes

TODO: using file system as bus

TODO: clear and globally same order of events

TODO: events consist of an id, command, time, duration, subject and issuer

TODO: events might have a duration

TODO: unspoken requirement: all nodes share the same clock - or one with very little drift

TODO: usage as synchronization primitive

9.3 Storage

9.3.1 Stage storage

Thinking about the storage organisation for the pipeline and its stages, a few expectations and concerns arise. First of all, to redo a stage, one needs to be able to access the files that were the result of one, two or multiple stages before the current. Sometimes a stage wants to access intermediate data produces by multiple previous stages. Next, the input video footage needs to be accessed by multiple stages throughout the pipeline execution. Finally, some stage results are not intermediate but final results. TODO: examples

The first and second concern can be solved by providing a workspace directory for each stage, that is copied from the logically previous stage. Once the computation of a stage is completed, the workspace is considered immutable and only used to source new workspaces from. This works fine for small intermediate results, but it does not work very well for large files - like the video footage. Start-

ing a new stage will take as long as the copy of multiple gigabytes on a spinning devices take (TODO: sample GB and time), require unnecessary storage due to multiple copies and provides no benefits in an archival and version control sense, because the video footage is not altered. So there needs to be another storage pool for input data, that is globally accessible and never changed: the global input storage pool. Providing one further storage pool for final results (global output pool), concern number three and four are also solved.

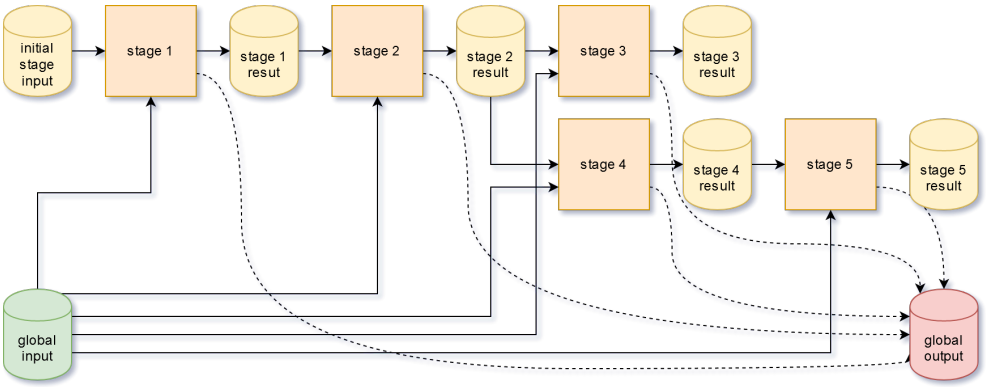
Because the very first stage has no workspace to source its files from, on creation of the pipeline a workspace directory for the "zeroths" stage is created. The user can then provide the very first stage with a predefined and non-empty workspace if necessary.

Because in the implementation of this project NFS was used, this also solves one further problem that was experienced in the prototyping phase: copying files on NFS (and many other network filesystems like Samba) is a client operation. This means, the client reads the input file and writes to the output file. Because the common network used (gigabit in this case) is slower than spinning disk sequential read/writes (network: 112 Megabytes/s in theory, divided by two because read+write on the same connection: 56 Megabytes/s, spinning enterprise disk: about 170 Megabytes/s TODO: test on johnny5), this operation does not only take unnecessary long but also renders the network connection close to unusable for any other participant in the network that needs to use parts of the same physical route.

To ensure that the global input and the previous intermediate results are not altered, the Docker daemon is instructed to mounted them as read-only filesystems. This makes it impossible[4] for the stage to accidently delete or alter the wrong files[5].

[4]When there is no bug in the implementation of mount or Docker

[5]Like in this scenario, where an unexpected space in a script caued people to loose their home directory: TODO: steam rm -rf incident



In summary, the storage pool solution looks like this:

TODO: check paths

- `/input` is mounted read-only from `nfs-server:/winslow/workspaces/<pipeline-id>/input`
- `/output` is mounted with write permissions from `nfs-server:/winslow/workspaces/<pipelin`
- `/workspace` is mounted with write permissions from `nfs-server:/winslow/workspaces/<pipe`
  and was created by the copying the workspace directory of the logically previous stage

TODO: define logically previous stage -> previous stage on linear execution and ... when jumping around

initial input

stage execution does not need to depend on the result of the exact previous

9.4 Decentralized Execution - how?

TODO: refer to kyberd presentations?

9.4.1 Every node has a connection to every other node

9.4.2 Centralized broker

9.4.3 Tree hierarchy

9.4.4 outcome

9.5 Communication and Node Management

The system that shall be developed, is supposed to spread jobs onto execution nodes as available. There are two main approaches in nod management and job assignment.

9.5.1 Centralized Management, Remote Execution

In the central management approach, there is always exactly one leader at any given moment in time. It is the responsibility of this leader to decide what to execute and where to execute it.

9.5.2 Decentralized Management, Local Execution

9.5.3 Combinations worth noting

stupid: - centralized management, local execution - decentralized management, remote execution

combination, decentralized + some remote slaves

9.6 Architecture

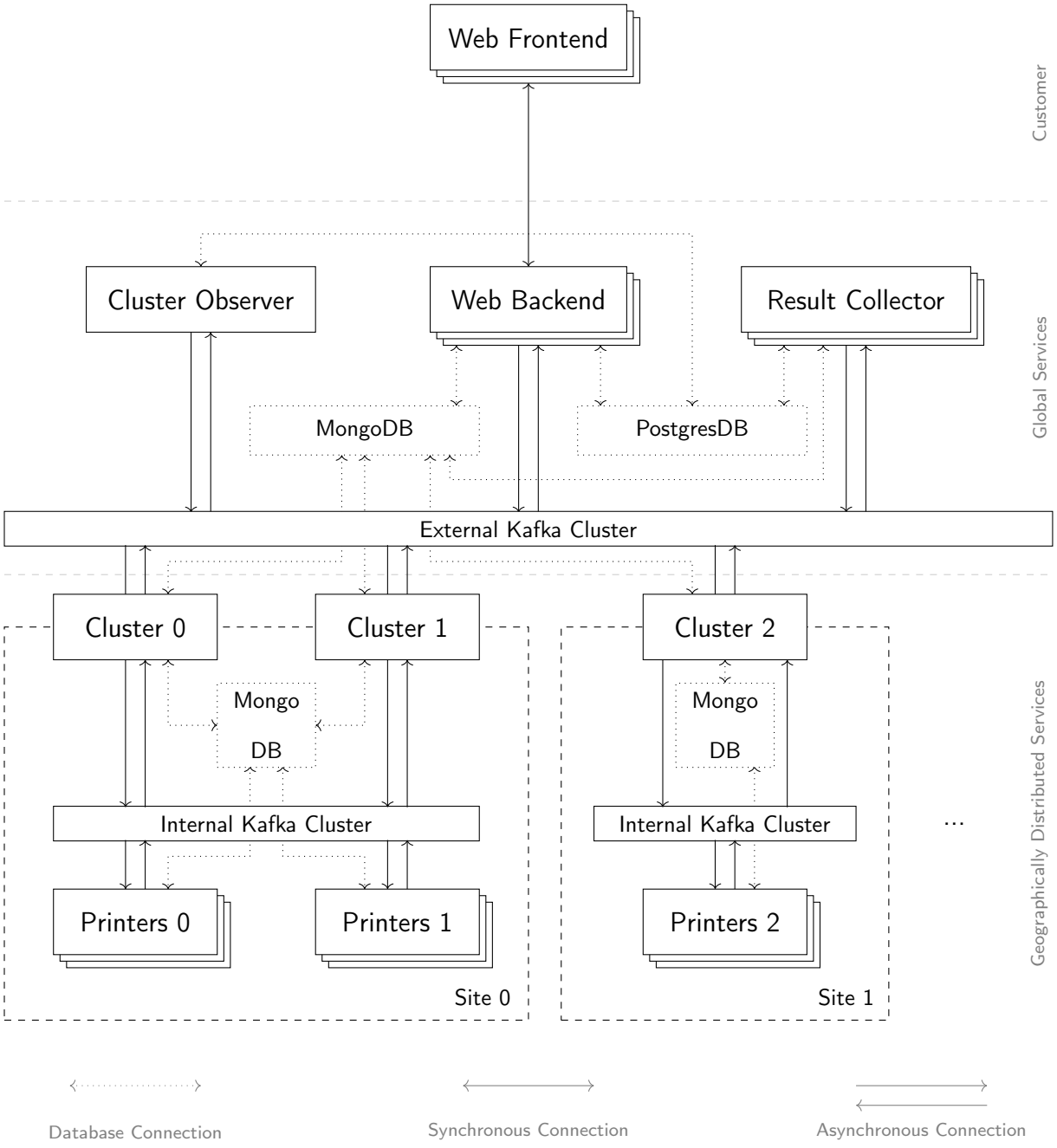


Figure 9.2: System Concept Overview

event based

common file system for communication because minimum requirements stick to unix principles(list): simple, human readable intermediate format voting/election by capabilities and 'will' of a node to run a stage

9.7 Communication/Event architecture?

9.8 Targeting capabilities

9.8.1 General thoughts

9.9 Planned

9.10 Implementation details

9.11 Synchronization, Locking, publishing events

9.12 REST for UI

9.12.1 Atomicy of (Unix) Filesystems

9.12.2 Atomicy and behavior of NFS in particular

9.12.3 Using as lock backend

9.12.4 Using as election backend

9.13 Election System

Utilizing Event Bus for timely limited elections and to ensure that there are no concurrent election for a single project.

TODO: what about concurrent elections on multiple projects

Chapter 10

Implementation notes

how thin shall it be

10.1 Agile development

10.2 Available infrastructure

10.3 Execution environment

10.4 Continuous Deployment

Chapter 11

Metrics?

idle times that would be without automatically executing tasks

maybe with error scenario: being able to dynamically execute a stage of another project instead of stalling

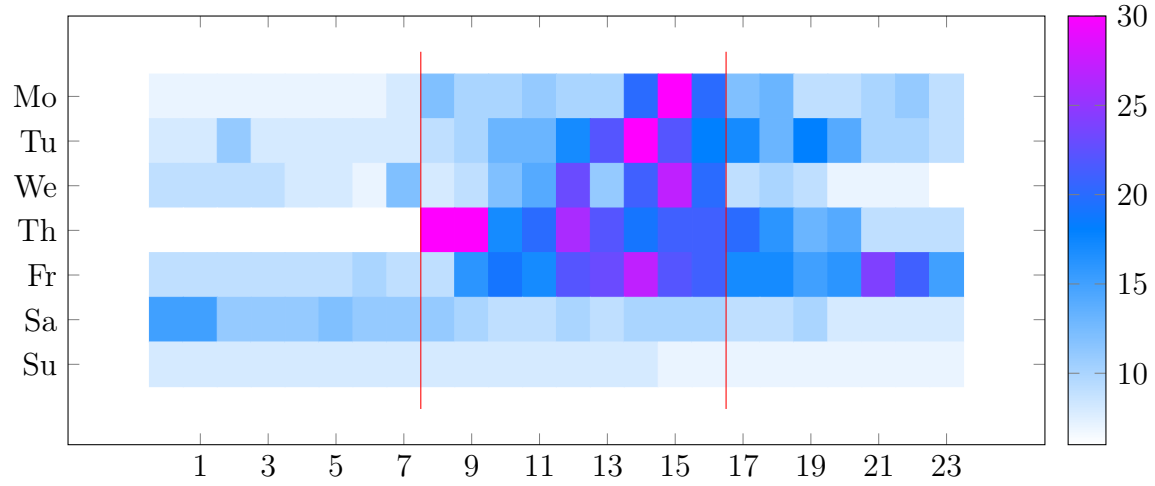


Figure 11.1: Actively running stages at a given day of the week and time of day

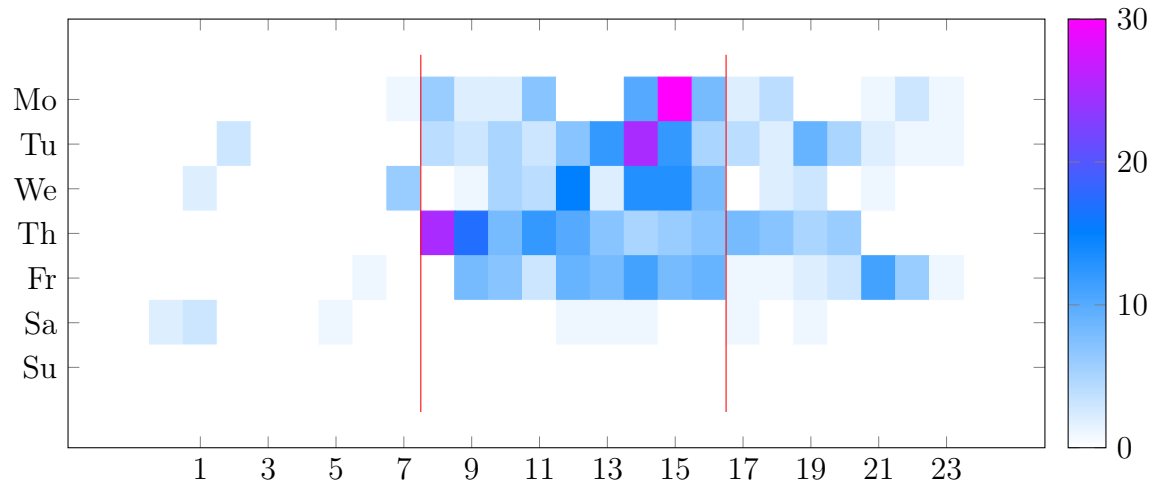


Figure 11.2: Stages starting at a given day of the week and time of day

TODO: same graphs but where the stage was not manually started between 8h and (+9h) 17h

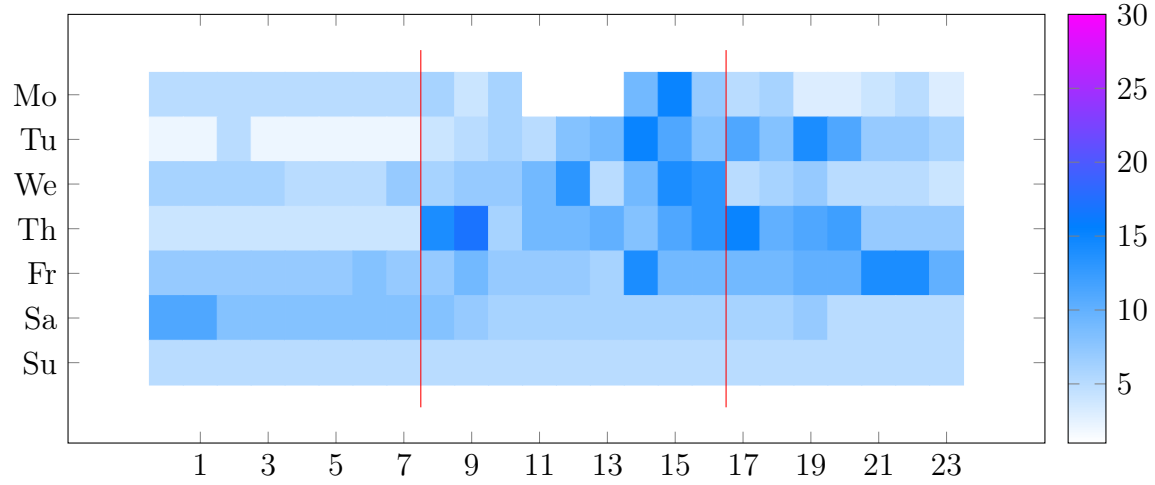


Figure 11.3: Actively running stages at a given day of the week and time of day which were started automatically

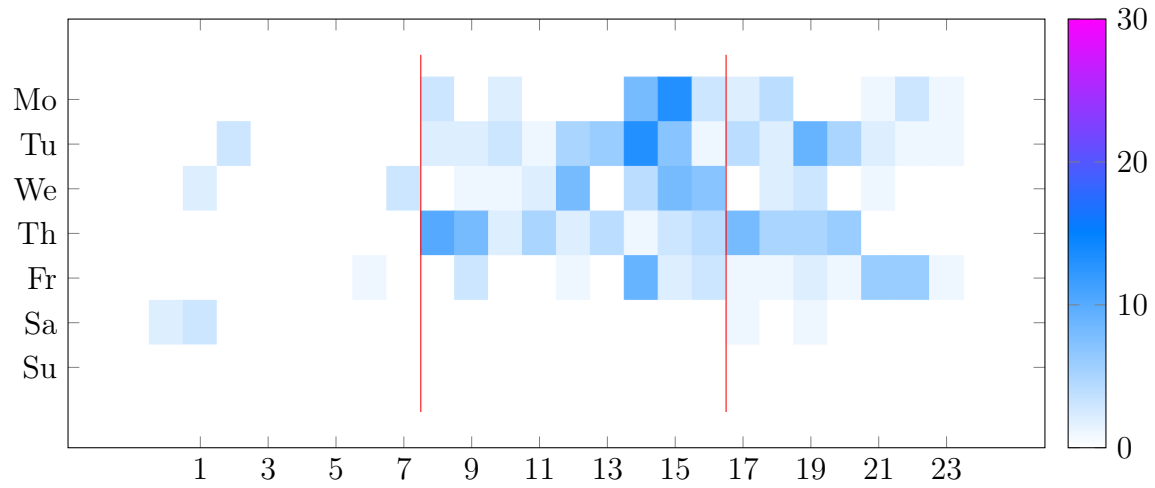


Figure 11.4: Stages automatically starting at a given day of the week and time of day

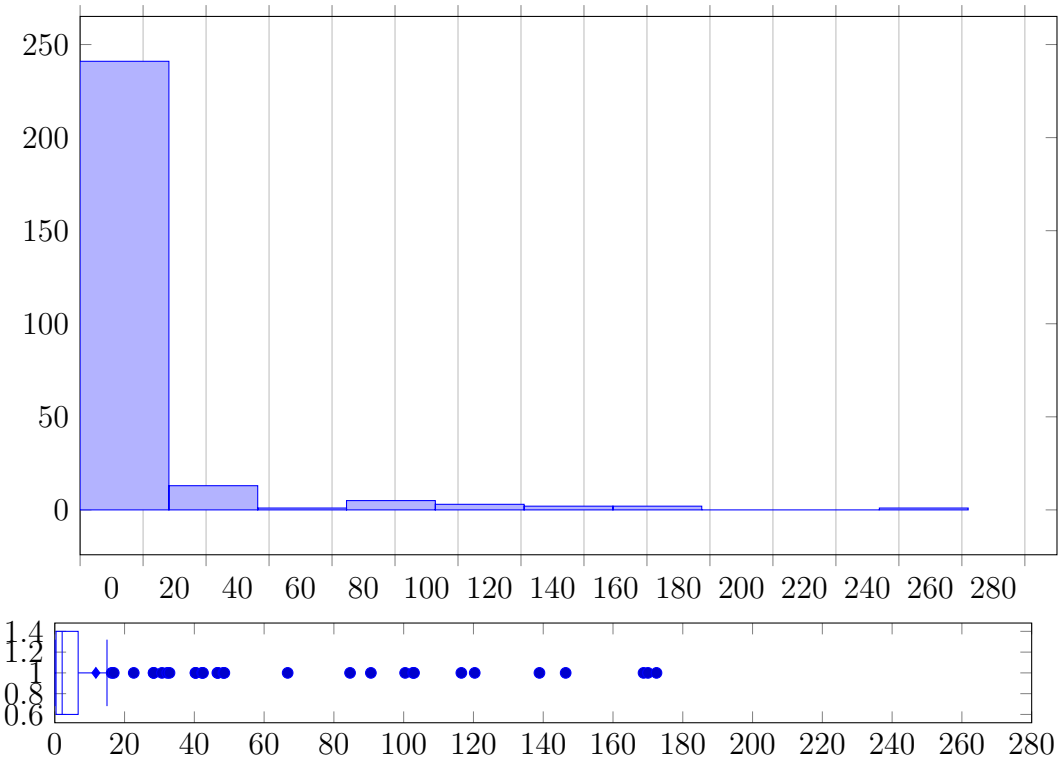


Figure 11.5: Stages automatically starting at a given day of the week and time of day

Chapter 12

Outcome and further work

In this chapter the main concerns are listed. For each concern the current progress is described as well as further work that needs to be done.

12.1 Storage

One of the central concerns is the storage management. The program needs to make input files available on each execution node and collect the results once the computation is complete. There are a few main architectural strategies to approach this. Simplified, either at a centralized location which is accessed by all execution nodes, a copy of the input files to the execution nodes or decentralized and distributed between all execution nodes and replication. The advantages and disadvantages can depend on the specific implementation and is therefore discussed in combination of such (see chapter 3).

Further testing is required to decide whether a more complex storage system is required, or the simplicity of a centralized solution outweighs the setup and maintenance overhead.

12.2 Coordination

Another important concern is the coordination of the nodes. A central coordinator with external server nodes, such as GitLab and Jenkins have, might not be sufficient for more complex and longer lasting pipelines. The probability that the master would need to be offline while there is a stage executed, is in the scenario of the desired workflow higher than for GitLab or Jenkins, because the stage is being execution for hours or days. Coupling stage execution plans on node availability ahead of time, as well as recovering from a sudden master failure implies additional implementation complexity. A decentralized coordination needs to be able to do this as well, but also allows the usage of the system while a node failed or is unreachable due to maintenance. With further prototyping and research a reasonable solution shall be found.

12.3 Binary distribution

In a time where containers are common and have proven to be usable, the installation of the binaries directly on the operating system they are executed on shall be avoided. There shall be no manual, nor automatic but custom file copies of the binaries or images from one server to the other. Experience shows, that without a proper management, this can easily become a mess, in which it is no longer clear, which files or images belong to which version. At the same time, making all binaries publicly available through the Docker Hub[34] is no option either. Whether a self hosted Docker Registry[35] could be the solution to this will be determined in further testing.

12.4 User Interface

Providing a useful user interface might not be important to the functionality of the system itself but for the user experience. A bad user experience will cause

a system not to be used. It became common practice for a rich user experience to be web based and interactive with JavaScript. For a potentially decentralized system, it is also advantageous to be able to access a disconnected node in the same manner as the remaining system, which further encourages a web based solution. Web based solutions such as React and Angular shall therefore be investigated for being used as user interface.

12.5 Requirements

TODO: more than first defined - as expected

TODO: lots more UI shortcuts found necessary while starting to use winslow

12.6 Architecture

TODO: loosely coupled backend driverFigure 8.1

12.7 High Level System Overview

TODO: intro words

TODO: explain what is going on in Figure 12.1

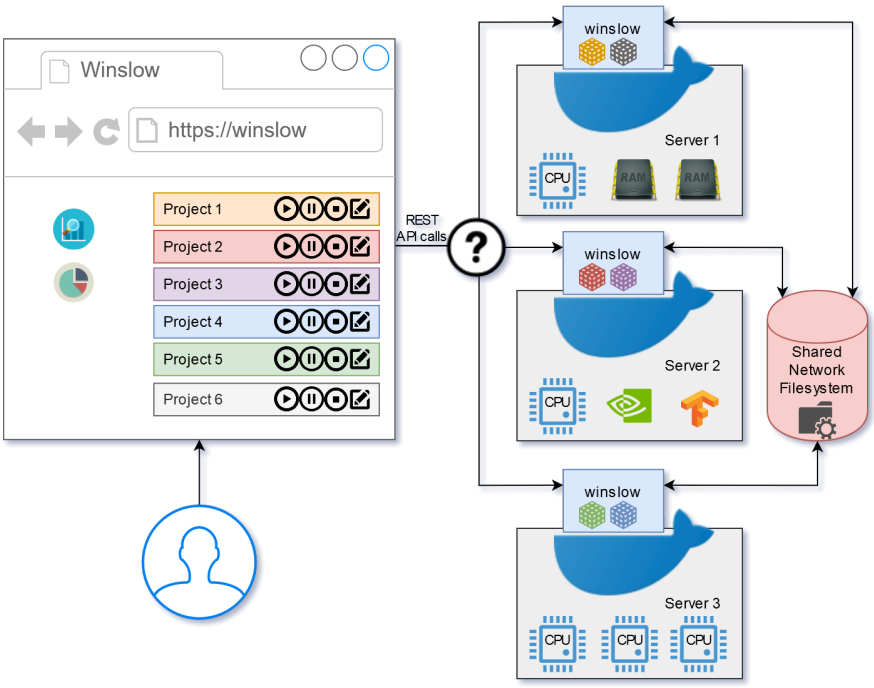


Figure 12.1: High level view on the system structure

Chapter 13

Further work

??

websockets (more event driven, less polling)

ceph/glusterfs?

Chapter 14

Conclusion

Bibliography

[1] MEC-View Dr. Rüdiger W. Henn. *Mobile Edge Computing basierte Objekterkennung für hoch- und vollautomatisiertes Fahren.* URL: `http://mec-view.de/` (visited on 09/29/2019).

[2] The Apache Software Foundation. *Apache Hadoop.* URL: `https://hadoop.apache.org/` (visited on 09/19/2019).

[3] Inc. GitLab. *The first single application for the entire DevOps lifecycle.* URL: `https://about.gitlab.com/` (visited on 09/21/2019).

[4] jenkins.io. *Jenkins. Build great things at any scale.* URL: `https://jenkins.io/` (visited on 09/19/2019).

[5] Inc. GitLab. *GitLab CI/CD. Pipeline Configuration Reference.* URL: `https://docs.gitlab.com/ee/ci/yaml/` (visited on 09/19/2019).

[6] jenkins.io. *Using a Jenkinsfile.* URL: `https://jenkins.io/doc/book/pipeline/jenkinsfile/` (visited on 09/19/2019).

[7] Camunda Services GmbH. *Workflow and Decision Automation Platform.* URL: `https://camunda.com/` (visited on 09/19/2019).

[8] Camunda Services GmbH. *Process Engine API.* URL: `https://docs.camunda.org/manual/7.6/user-guide/process-engine/process-engine-api` (visited on 09/19/2019).

[9] Camunda Services GmbH. *Rest Api Reference.* URL: `https://docs.camunda.org/manual/7.8/reference/rest` (visited on 09/19/2019).

[10] HashiCorp. *Nomad.* URL: https://www.nomadproject.io/ (visited on 09/19/2019).

[11] HashiCorp. *Nomad. Nomad vs. Kubernetes.* URL: https://www.nomadproject.io/intro/vs/kubernetes.html (visited on 09/19/2019).

[12] Bc. Pavel Peroutka. *Web interface for the deployment andmonitoring of Nomad jobs. Master's thesis.* URL: https://dspace.cvut.cz/bitstream/handle/10467/80106/F8-DP-2019-Peroutka-Pavel-thesis.pdf (visited on 09/22/2019).

[13] Tigran Mkrtchyan Patrick Fuhrmann. *dCache. Scope of the project.* URL: https://www.dcache.org (visited on 09/19/2019).

[14] Patrick Fuhrmann. *dCache, the Overview.* URL: https://www.dcache.org/manuals/dcache-whitepaper-light.pdf (visited on 09/19/2019).

[15] Inc. Terracotta. *QuartzJob Scheduler.* URL: http://www.quartz-scheduler.org/ (visited on 09/19/2019).

[16] Data Revenue. *Distributed Python Machine Learning Pipelines.* URL: https://www.datarevenue.com/en/blog/how-to-scale-your-machine-learning-pipeline (visited on 09/19/2019).

[17] Ask Solem. *Celery: Distributed Task Queue.* URL: http://www.celeryproject.org (visited on 09/19/2019).

[18] IBM Knowledge Center. *IBM InfoSphere. DataStage.* URL: https://www.ibm.com/support/knowledgecenter/en/SSZJPZ_9.1.0/com.ibm.swg.im.iis.ds.design.doc/topics/c_ddesref_Server_Job_Stages_.html (visited on 09/19/2019).

[19] Wikipedia contributors. *Qsub. Wikipedia, The Free Encyclopedia.* URL: https://en.wikipedia.org/w/index.php?title=Qsub&oldid=745279355 (visited on 09/22/2019).

[20] The University Of Iowa. *Basic Job Submission. HPC Documentation Home / Cluster Systems Documentation.* URL: https://wiki.uiowa.edu/display/hpcdocs/Basic+Job+Submission (visited on 09/22/2019).

[21] Swiss National Supercomputing Centre. *High Throughput Scheduler.* URL: https://user.cscs.ch/tools/high_throughput/ (visited on 09/19/2019).

[22] Colin Phipps. *zsync. Overview.* URL: http://zsync.moria.org.uk/ (visited on 09/19/2019).

[23] OpenIO. *High Performance Object Storage for Big Data and AI.* URL: https://www.openio.io/ (visited on 09/22/2019).

[24] Chris Lu. *Simple and highly scalable distributed file system.* URL: https://github.com/chrislusf/seaweedfs (visited on 09/22/2019).

[25] Colin Phipps. *Alluxio. Data Orchestration for the Cloud.* URL: https://www.alluxio.io (visited on 09/19/2019).

[26] Inc Red Hat. *Gluster. Free and open source scalable network filesystem.* URL: https://www.gluster.org (visited on 09/19/2019).

[27] Inc. Docker. *Enterprise Container Platform.* URL: https://www.docker.com/ (visited on 09/22/2019).

[28] Inc. Docker. *Docker. Docker Logos and Photos.* URL: https://www.docker.com/company/newsroom/media-resources (visited on 01/16/2020).

[29] Inc. Docker. *Docker Documentation. Swarm mode key concepts.* URL: https://docs.docker.com/engine/swarm/key-concepts/ (visited on 03/17/2020).

[30] The Kubernetes Authors. *Kubernetes. What is Kubernetes.* URL: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (visited on 03/17/2020).

[31] Inc. Docker. *Docker Documentation. Docker overview.* URL: https://docs.docker.com/engine/docker-overview/ (visited on 01/16/2020).

[32] J. Goll. *Methoden des Software Engineering: Funktions-, daten-, objekt- und aspektorientiert entwickeln*. Springer Fachmedien Wiesbaden, 2012.

[33] Inc. Docker. *Press Release. Docker Restructures and Secures $35 Million to Advance Developer Workflows for Modern Applications*. URL: `https://www.docker.com/press-release/docker-new-direction` (visited on 03/23/2020).

[34] Inc. Docker. *Docker Hub. Build and Ship any Application Anywhere*. URL: `https://hub.docker.com/` (visited on 09/22/2019).

[35] Inc. Docker. *Docker Documentation. Docker Registry*. URL: `https://docs.docker.com/registry/` (visited on 09/22/2019).