Master's thesis in the course of studies
Computer Science and Media

Implementation and evaluation of a hybrid microservice infrastructure

submitted by

PATRICK KLEINDIENST
Matr-Nr. 31924

at Stuttgart Media University
on October 24, 2017

in partial fulfillment of the requirements
for the degree of Master of Science

Supervisor:
PROF. DR. DIRK HEUZEROTH,
Stuttgart Media University

Co-Advisor:
DIPL.-ING. (FH) THOMAS POHL,
IBM Deutschland R&D GmbH

Ehrenwörtliche Erklärung (Declaration of honour)

Hiermit versichere ich, Patrick Kleindienst, ehrenwörtlich, dass ich die vorliegende Masterarbeit mit dem Titel: *Implementation and evaluation of a hybrid microservice infrastructure* selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der ehrenwörtlichen Versicherung und die prüfungsrechtlichen Folgen (§ 23 Abs. 2 Master-SPO (3 Semester)) einer unrichtigen oder unvollständigen ehrenwörtlichen Versicherung zur Kenntnis genommen.

Ort/Datum	Unterschrift

Acknowledgement

Foremost, my sincere thanks goes to my first advisor Prof. Dr. Dirk Heuzeroth for his outstanding support of my master's thesis.

I would also like to express my sincere gratitude to my second advisor
Dipl-Ing. (FH) Thomas Pohl for offering me the opportunity to spend a great and instructive time at the labs of IBM Deutschland Research and Development GmbH in Böblingen while working on this thesis. His patient and enthusiastic guidance helped me in all the time of research and writing.

In this sense, I thank all my colleagues of the IBM System z firmware department for their warm welcome, the pleasant working atmosphere and their constant support over the last months.

Besides, I would like to thank M.Sc. Stephan Soller for his encouragement, his insightful comments and the sound discussions.

Last but not least, I would like to express my sincere thanks to my family and girlfriend for their emotional and spiritual support during my studies.

Abstract

Large-scale computing platforms, like the IBM System z mainframe, are often administrated in an out-of-band manner, with a large portion of the systems management software running on dedicated servers which cause extra hardware costs. Splitting up systems management applications into smaller services and spreading them over the platform itself likewise is an approach that potentially helps with increasing the utilization of platform-internal resources, while at the same time lowering the need for external server hardware, which would reduce the extra costs significantly. However, with regard to IBM System z, this raises the general question how a great number of critical services can be run and managed reliably on a heterogeneous computing landscape, as out-of-band servers and internal processor modules do not share the same processor architecture.

In this thesis, we introduce our prototypical design of a microservice infrastructure for multi-architecture environments, which we completely built upon preexisting open source projects and features they already bring along. We present how scheduling of services according to application-specific requirements and particularities can be achieved in a way that offers maximum transparency and comfort for platform operators and users.

Kurzfassung

Die Adminstration von Großrechnerplattformen, wie beispielsweise des IBM System z Mainframes, erfolgt oftmals anhand einer Out-of-Band Herangehenweise, bei der ein Großteil der Systems Management Software auf dedizierten Servern betrieben wird, welche zusätzliche Hardwarekosten verursachen. Indem Systems Management Applikationen in kleinere Services aufgetrennt und gleichermaßen über die Plattform selbst verteilt werden, kann möglicherweise die Auslastung der Plattform-internen Ressourcen erhöht und gleichzeitig der Bedarf nach externer Serverhardware gesenkt werden, was die zusätzlichen Kosten deutlich reduzieren würde. Im Fall von IBM System z wirft dies allerdings die allgemeine Frage auf, wie eine große Anzahl an kritischen Services auf einer heterogenen Rechnerlandschaft zuverlässig ausgeführt und verwaltet werden kann, da Out-of-Band Server und interne Prozessormodule auf unterschiedlichen Prozessorarchitekturen basieren.

In dieser Masterarbeit stellen wir unseren prototypischen Entwurf einer Microservice-Infrasturktur für Multiarchitekturumgebungen vor, der vollständig auf bereits existierende Open-Source-Projekte und darin verfügbare Features aufgebaut ist. Wir demonstrieren, wie Scheduling von Services entsprechend anwendungsspezifischer Anforderungen und Besonderheiten in einer Weise umgesetzt werden kann, die sowohl Plattformbetreibern als auch -nutzern maximale Transparenz sowie größtmöglichen Komfort bietet.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

APT	Advanced Package Tool
API	Application Programming Interface
ARM	Advanced RISC Machines
ASF	The Apache Software Foundation
AUFS	Advanced Multi-Layered Unification Filesystem
AWS	Amazon Web Services
CD	continuous delivery
CLI	command-line interface
CNCF	Cloud Native Computing Foundation
CoW	copy-on-write
CPU	central processing unit
DCIM	Data Center Infrastructure Management
DNS	Domain Name System
Docker CE	Docker Community Edition
Docker EE	Docker Enterprise Edition
DoS	Denial of Service
DSL	Domain Specific Language
ESB	Enterprise Service Bus
FIFO	first-in-first-out
GPG	GNU Privacy Guard
HTML	Hypertext Markup Language

HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
JDK	Java Development Kit
JNI	Java Native Interface
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LXC	Linux Containers
LPAR	logical partition
OS	operating system
RAM	random-access memory
RDBMS	relational database management system
REST	Representational State Transfer
RHEL	Red Hat Enterprise Linux
RPC	remote procedure call
SDK	Software Development Kit
SE	Support Element
SLES	SUSE Linux Enterprise Server
SQL	Structured Query Language
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UI	User Interface
VM	virtual machine
WSC	Warehouse-Scale Computer
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Motivation

Nowadays, a wide range of computing platforms exists, each one with a slightly different focus in terms of who uses them and in which way. While, in the meantime, almost everyone has a smartphone to run apps, companies like Google or Amazon operate large-scale data centers in order to supply private or commercial customers with cloud storage as well as an universal platform for running their IT services or web applications. Critical businesses, like financial institutions, have their own mainframes in order to process financial transactions in a reliable manner.

Especially large computing platforms like mainframes or data centers usually require some sort of "high-level firmware" which, as part of a platform's systems management infrastructure, provides the basic functionality to execute end user workload, for instance a MySQL database. This firmware, which we also refer to as *system management software*, performs tasks like initializing peripheral hardware, installing software and supervising power consumption [44, 11]. The exact functional scope of firmware depends on the concrete type of platform as well as its characteristics, since different computing landscapes usually have different needs related to systems management. Nevertheless, the overall goal of each kind of firmware is bringing a system into operational state and keep it there as long as possible.

It is crucial to not confuse firmware as we understand it in the context of systems management with software which responsibilities are limited to enabling interaction of an operating system (OS) with a single specialized hardware component like a graphic card [37]. Even though the latter variant is probably closer to the general interpretation of firmware, what we mean by it is administrative software that condenses a large number of central processing units (CPUs), memory modules etc. into a single reliable computing platform.

Depending on which type of computing platform is considered, there are different ways in which firmware and hardware can be organized. Taking multi-machine

platforms like data centers as an example, we can imagine firmware services which are not tightly bound to a certain hardware component, for example applications that periodically create snapshots of virtual machines (VMs) or watch the global health state of all servers. One option is to put such services on extra servers residing outside the actual computing platform, as shown by figure 1.1. This leads to a strict isolation between the platform itself, which almost exclusively runs customer workload, and some external hardware, which solely serves the purpose of running systems management services. Such a design is quite common for mainframes like IBM System z*, where a large portion of the firmware runs on dedicated Support Elements (SEs) [44]. The idea behind this model is to achieve a high level of fault-tolerance for the platform by means of redundancy as well as a strict hardware boundary between firmware and customer workload.

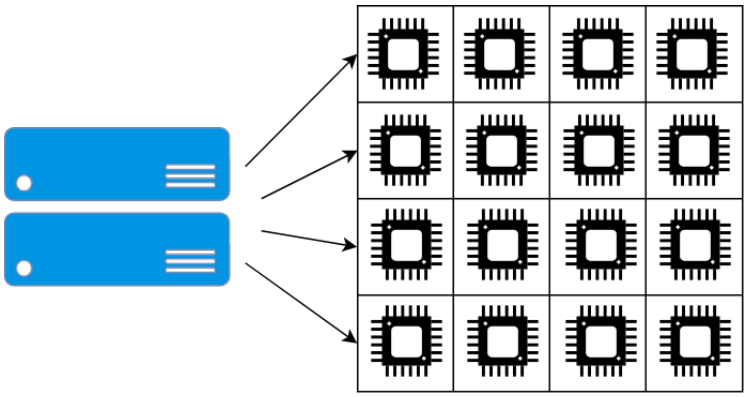


Figure 1.1: Platforms may be managed in an out-of-band manner.

While this makes perfectly sense from a risk management perspective, it must be considered that extra hardware still causes additional costs related to acquisition and power consumption. Besides, supplying hardware components redundantly usually causes their overall utilization to be low. Consequently, from an economical perspective, the question is raised if the same level of reliability and stability of the platform can also be reached with less extra hardware. A thinkable approach to reach this goal is the moving of great parts of the firmware from external components to the platform itself. On condition that the mainframe's availability guarantees can be sustained, we expect the additional out-of-band servers, at least partially, to become obsolete and the overall utilization with respect to remaining external components and the platform-internal resources to grow. Figure 1.2 illustrates this idea.

A direct consequence of the lower demand for extra hardware are lower costs.

* Trademarks of IBM in USA and/or other countries.

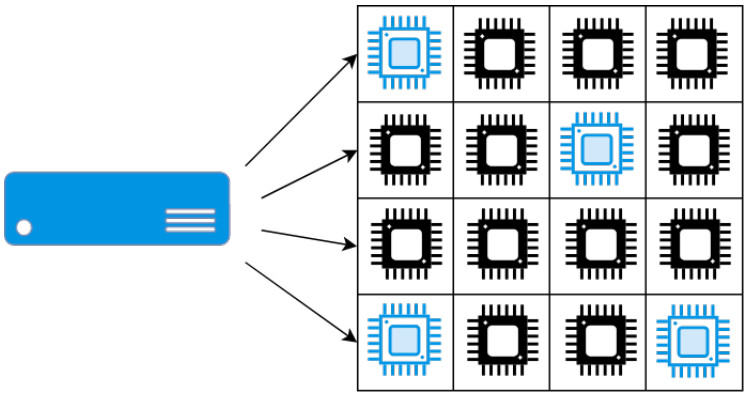


Figure 1.2: Partially shifting firmware onto the platform can save additional hardware.

However, there are even more possible benefits that might not be so obvious at first sight: Shifting firmware onto a platform which is logically and/or physically distributed by design would enforce the firmware applications themselves to follow a modular and decentralized software architecture. Today, even though many parts of a mainframe's systems management software already execute in their own processes, the software design as a whole is still highly monolithic. If we, however, embrace the inherently distributed nature of large-scale computing platforms, then we can possibly evolve firmware towards a microservice-based architecture, which can benefit from clear interfaces and responsibilities, and, as a side effect, offers the opportunity to help with improving code quality.

Unfortunately, operating software as microservices entails increased operational costs, as a large number of independent and distributed processes must be controlled and managed. Because the complexity that comes along with the administration of microservices grows exponentially with the number of applications deployed, using an automation and management infrastructure for microservice orchestration becomes inevitable at a certain point. While cluster management in data centers is not a novelty any more today, this is not true for less common platforms like mainframes. This is because, especially in the case of IBM System z, expanding the scope of firmware services towards the platform introduces the need for managing applications in a system comprising more than just one kind of processor architecture. Consequently, we end up with a so-called *hybrid* platform, consisting of amd64-based SEs as well as s390x* processor modules [44].

* Trademarks of IBM in USA and/or other countries

1.2 Thesis objectives and scope

In this thesis, we introduce our prototypical implementation of a hybrid microservice orchestration infrastructure, using the IBM System z mainframe as a reference target platform. Our work was mainly driven by achieving the goal of a reliable and flexible microservice management system which is able to meet possibly specific characteristics and individual demands of applications if necessary. Although this project has its origins in the context of systems management software, we want to clarify that, for the scope of our prototype development, the actual sort of workload is irrelevant and that we have not limited ourselves to firmware in any sense. Consequently, while infrastructure characteristics like application-aware scheduling make sense in the context of firmware, others probably do not.

We also put emphasis on reusing existing open source projects and tools that already exist rather than developing a custom solution from scratch in order to see how far we can get by doing it this way. In order to fulfill the special requirements of our mainframe use case, we decided upon designing our prototype around Apache Mesos [78], a highly-available and scalable resource abstraction layer that condenses an arbitrary number of machines into a single pool of computing resources. Mesos offers the opportunity of running multiple frameworks on a single cluster, a fact we took advantage of in order to realize application-aware scheduling, which considers the divergent nature of firmware applications, by maximizing the overall utilization of the available hardware resources simultaneously.

It should be noted that the scope of this thesis is limited to describe our idea of how hybrid microservice management could work, without demanding to demonstrate a complete and production-ready setup which covers all the details of operating distributed systems. We therefore mainly focused on the issues of platform heterogeneity and high-availability of infrastructure parts and deployments, leaving equally important aspects like monitoring and security aside for further advancement of the solution we present as the outcome of our work.

Terminology. Note that, even though we explicitly refer to the more up to date 64-bit variants of the processor architectures in question (amd64 and s390x), we implicitly include their 32-bit (x86 or rather s390 [44]) predecessors. Furthermore, we use the terms *service*, *application* and *job* synonymously. A *task* denotes a single instance of a job, which can consist of n tasks with $n \geq 1$.

1.3 Outline

We begin with providing some background in chapter 2, giving a more precise definition of systems management and the problems it solves. Furthermore, we explain the advantages and challenges of transforming a monolithic application design into

a microservice architecture and cover the most essential issues in terms of orchestrating microservices, with a special focus on multi-architecture platforms. Chapter 3 describes the basic ideas and concepts behind Apache Mesos and its peripherals, while chapter 4 shows how this knowledge can be applied to establish a microservice infrastructure in a common amd64 environment. In chapter 5, we expand the basic setup to span not only across amd64 VMs but also a s390x firmware partition to end up with a prototype that simulates a mainframe scenario. We evaluate our results in chapter 6, and chapter 7 closes.

Chapter 2

Background

First, this chapter takes a closer look at systems management and the responsibilities of systems management software in particular. In the next step, the focus is placed on systems management software design and how changing its monolithic structure towards a highly modular microservices architecture can contribute to both a platform's stability and cost-efficiency. Lastly, this chapter covers microservice management infrastructure, its general necessity and the added value it provides with respect to orchestrating a large number of applications.

2.1 About systems management

The concrete challenges related to the administration of computing platforms strongly depend on characteristics like structure, size (in terms of number of machines and resources like random-access memory (RAM) and storage), involved processor architectures and purpose. Despite their similarities, platforms like data centers and mainframes obviously differ in these aspects, showing a diversity that makes finding a universal definition for *systems management* very hard. As a consequence, there is no uniform concept of systems management in computer science that describes a generally valid set of disciplines and techniques which is suitable for all computing landscapes. Therefore, we considered it reasonable to take a look at this topic from different angles. Before that, though, we want to give a concise overview of different kinds of platforms and their characteristics.

2.1.1 Large-scale computing platforms

Computing platforms, or - more simply - *systems*, can take various forms, from small end-user devices like smartphones to huge data centers being made up of thousands of servers [5]. Because or focus is on large-scale computing platforms and their inherent administration complexity, we will mainly focus on data centers and especially mainframes.

Barroso et al. [5] describe data centers as buildings that, in a traditional sense, host a great number of software applications that run on their dedicated hardware infrastructure each. While these infrastructures, which might even belong to different companies, probably differ from each other, each one is homogeneous within itself. The homogeneity property is even more present regarding Warehouse-Scale Computers (WSCs), which, according to Barroso et al. [5], are a more modern kind of data center. WSCs are usually operated by a single large company like Google or Amazon and thus comprise highly homogeneous hardware and software [5].

Mainframes, for instance IBM System z, pose another type of computing platform or system. They can be viewed as large-scale servers that can run multiple host OSes in their distinct logical partitions (LPARs) simultaneously. In contrast to a data center unit or a WSC, the IBM System z mainframe is a strongly heterogeneous platform, comprising amd64-based SEs for out-of-band system control as well as a s390x-based processor complex which is responsible for executing the actual workload [44]. While data centers are used for rolling out large deployments of web applications particularly, mainframes play an important role for critical businesses like finance and health care [44].

2.1.2 What is systems management?

In the field of data centers, systems management, which is also referred to as *Data Center Infrastructure Management (DCIM)* in this context, is defined as a set of tools that "monitor, measure, manage and/or control [..] use and energy consumption of all IT-related equipment", for instance servers and storage [11]. As its core disciplines, DCIM includes, inter alia, power, network and alert management. It must be noted, though, that these aspects are not solely implemented in software, but also require hardware components like sensors for temperature measurement [11].

Drawing our attention towards mainframes, we discovered that Jones et al. [44] define systems management in a similar way, describing as "a collection of disciplines that monitor and control a system's behavior.". While they discuss this topic with a special focus on IBM System z, they still stress its inherent context sensitivity. As their main aspects of systems management, they list business management, configuration management, performance management, operations management, change management and problem management. As with DCIM, it is important to understand that systems management as it is described here is not limited to firmware, but also "designed into System z hardware" [44]. As for tasks which are exclusively related to systems management software, Jones et al. [44] allude, amongst others, to hardware testing, loading of operating systems and failure recovery.

2.2 Rethinking IBM System z firmware

Of course, the stability and reliability of a computing platform do not only depend on systems management firmware, but also on the hardware components a system is built upon as well as their quality and durability. Nevertheless, firmware can make its contribution to a robust system by following mature practices for good software development and architecture. As for IBM System z, the current, mostly monolithic systems management software design is strongly dominated by the strict separation between firmware and workload infrastructure, with the firmware being located - to a large extent - on two integrated SEs between which the system can switch in case of failure. The SEs serve the purpose of operating and monitoring the System z mainframe [44].

The actual workload, for example one or more databases, is executed on host systems booted in dedicated LPARs. Unlike the SEs, which basically are common amd64 servers, the processor modules running the LPARs are based on IBM's custom s390x processor architecture [44]. As explained in the introduction, it makes perfectly sense to think about how free resources on the processor modules (and also on the alternate SE) can be used more efficiently, for example by moving some of the system management services from the SEs onto the platform itself.

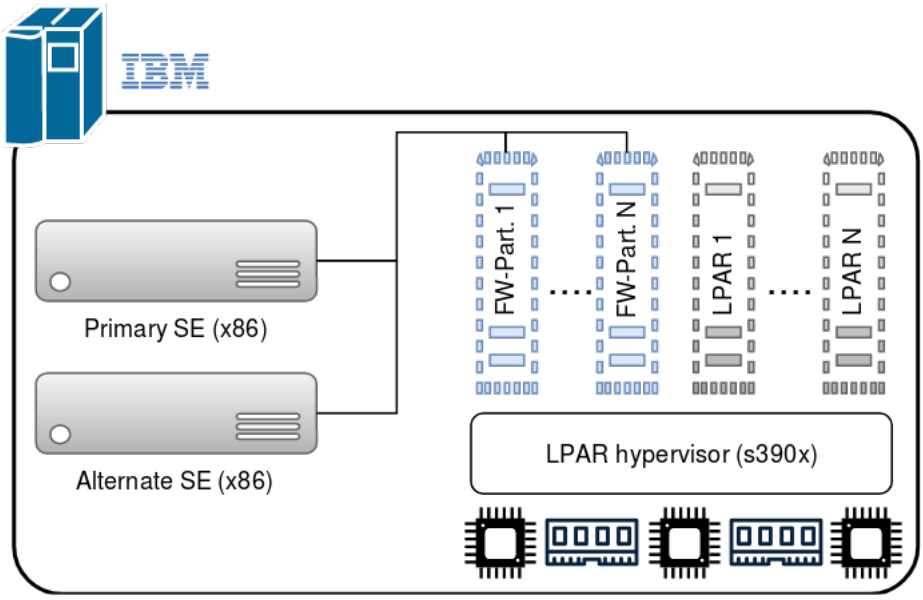


Figure 2.1: Shifting systems management workload from SEs to firmware partitions.

Practically, this can be done by creating one or more so-called *firmware partitons* on top of the LPAR hypervisor (see figure 2.1). Consequently, the degree of distribution of firmware applications automatically increases. We can embrace this fact and exploit it as a motivation to rethink the overall design of the IBM System z firmware.

Considering modern software architecture, the microservice architectural style is a lasting trend over the last years, promising to be the key to highly available, scalable and flexible application deployments. Today, these topics are mainly discussed in the context of web applications backed by cloud providers like Amazon Web Services (AWS) or the Google Cloud Platform, which are backed by numerous large scale data centers spread across the world. The microservice style is, however, by no means limited to end-user applications running on cloud provider infrastructure. Instead, regarding our mainframe scenario illustrated in figure 2.1, we can also exploit it to rework the System z firmware design towards a microservice-based structure that takes advantage of the physical distribution between the SEs and the firmware partitions. The resulting microservice architecture can help with improving the stability of not only the systems management software, but of the mainframe platform as a whole. Pursuing this idea even further, moving more and more parts of the firmware onto the processor modules could, at least, render the alternate SE obsolete since the platform itself could be used for redundancy purposes. In this way, the utilization of existent hardware could be increased while the costs for additional components would partially disappear.

The next section gives an accurate introduction to both monolithic and microservice-based architecture and works out in which way software services in general, but maybe also systems management applications, can benefit from moving along the path towards a more modular design.

2.3 Monoliths vs. microservices

2.3.1 The monolithic architectural style

According to Fowler et al. [36], the monolithic style can usually be observed in classical enterprise web applications, which, in essence, are made up of three major parts: A client-side User Interface (UI), consisting of Hypertext Markup Language (HTML) documents enriched with JavaScript code, a relational database management system (RDBMS) comprising one or more databases which in turn can include several tables, and lastly a server-side backend application. The server-side application listens for Hypertext Transfer Protocol (HTTP) requests and handles these by executing business logic and reading from or writing to the underlying database. At least, it selects a certain HTML view which is then populated with the required data and finally delivered to the client's web browser [36].

It must be considered that a server-side application does not necessarily need to return HTML documents as HTTP response payload. Alternatively, it might accept and return raw data in JavaScript Object Notation (JSON) or Extensible Markup Language (XML) format, which is normally true for applications adhering to the Representational

State Transfer (REST) paradigm. Such a RESTful design allows for rich client applications, which either reside on distinct web servers or directly on end-user devices and communicate with server backends by exchanging data representations. Following the RESTful paradigm entirely frees the server backends from all UI concerns. Instead, they can offer a web-based HTTP and/or remote procedure call (RPC) Application Programming Interface (API) that can be consumed by a wide range of client applications [59].

Since server application, database and UI are executed in separate processes and are usually located on different hosts or VMs, enterprise applications can already be considered distributed systems in some way. However, this statement is merely valid to a limited extent. Despite potential physical boundaries between the processes, it is still the application server process which owns the entire business logic [36]. Such a design, which is commonly referred to as a *monolithic* architecture [36], shows certain significant characteristics:

- The entire business logic is consolidated into one executable is owned by a single process [36].
- Shipping updates requires compiling, packaging and deploying a new version of the application artifact [36].
- Depending on the programming language in use, a monolith might be modularized in terms of classes, functions, packages or namespaces [36].
- In case of capacity bottlenecks, scaling is done horizontally by putting several instances of the same server application behind a load balancer [36].

2.3.2 Drawbacks of monolithic applications

There are lots of use cases where a monolithic design is perfectly suitable. However, the software industry has experienced substantial changes for a couple of years, and is still doing so. Because software thrills a constantly increasing amount of business segments, numerous IT companies have already started moving their applications to the cloud, striving for shorter release cycles and fast adaption to customer needs in order to create even more business value. As a consequence, some essential properties of the monolithic approach have evolved into serious handicaps:

- Establishing short release cycles with legacy applications is hard to achieve, since applying changes related to a single business logic aspect implicitly requires recompilation and redeployment of the application as a whole. Consequently, all developers working at different application parts must coordinate closely and should agree on synchronous release cycles [36].

- Keeping a clean modular structure in a single software project is an enormous challenge. As bypassing module boundaries in order to implement features the quick and convenient way is always a temptation, it might introduce undesirable dependencies that span across many parts of the system [35].
- Horizontal scaling comes along with a huge demand for computing resources, because the monolith itself poses the smallest scalable unit. This demand causes substantial extra costs, regardless of whether the application is hosted in cloud environments or in-house [36].

2.3.3 The microservice architectural style

The microservice architectural style as described by Fowler et al. [36] tends to overcome these issues by disposing business logic over a suite of services instead of keeping it within a single deployment unit, as figure 2.2 illustrates. Each one of these so-called "microservices" is built around a certain business capability, providing functionality tied to a discrete domain. Communication between services is done via lightweight mechanisms like HTTP web requests or RPCs protocols. In addition, microservices are independently deployable and may even be written in different programming languages [36].

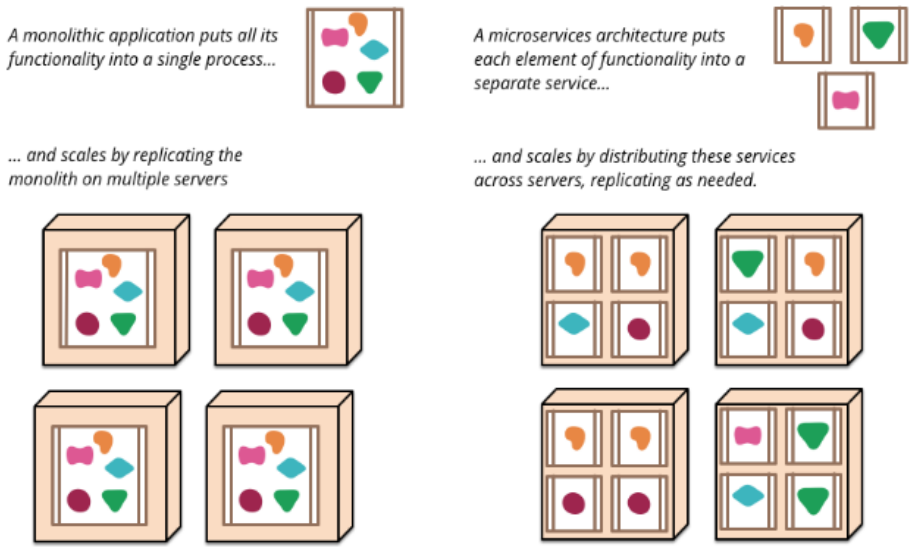


Figure 2.2: High-level comparison of monoliths and microservices [36].

Wootton [102] chooses a similar explanation, describing microservices as an approach that relies on a set of services with a certain set of properties:

- Simple and focused on doing one thing well.

- Possibly built upon different technologies, picking "the best and most appropriate tool for the job".
- Designed in a way yields an implicitly decoupled system.
- Can independently be evolved and delivered by separate development teams.
- Promote continuous delivery (CD) by enabling frequent releases whilst the system itself always remains stable and available.

Fowler et al. [36] also emphasize that the differences between currently existing systems based on the microservice style complicate drawing up a more formal and generally valid definition. Instead, they provide a list of characteristics that, according to their practical experience, most paratical implementations have in common.

Componentization via Services

Splitting up larger systems into pluggable components is a field-tested technique in software development. A brief definition for a component is given by Fowler et al. [36], who describe it as a "unit of software that is independently replaceable and upgradeable". Whereas most monolithic architectures make use of libraries, classes or packages for componentization, microservices go one step further by putting components into their own processes. In this way, each component's life cycle becomes independently manageable.

Organization around Business Capabilities

Forming teams with a focus on technology leads to the emerge of UI teams, server-side teams as well as database teams. According to Conway [13], this eventually results in an equally layered software design. Conway [13] generalized this theorem, which finally became famous as *Conway's Law* [12] and today is an important driver for the microservice architecture.

> *"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."*
>
> Melvin Conway, 1968 [12]

The microservice style utilizes the causal link between software design and organization structure by forming cross-functional teams which possess all the skills required in order to build an application around business capabilities. That increases the chance to end up with a suite of highly decoupled services with a strong focus on

their business domain [36]. Newman [59] correlates the narrow focus of microservices to cohesiveness of source code, which reveals to what extent code that is grouped together (in a class or package) is actually related. He translates Robert C. Martin's definition of the *Single Responsibility Principle* to microservices, claiming that it should be "obvious where code lives for a given piece of functionality" [59].

Products not Projects

Instead of a project-oriented model, which solely pursues the goal of giving birth to a piece of software that is handed over to another maintenance and operations team after development, the microservice style favors the product model. This mentality widens a team's accountability to not only developing, but also maintaining and running software in production [36]. Amazon CTO Werner Vogels [61] paraphrases this notion as "You build it, you run it", arguing that bringing developers in day-to-day contact with operating their own applications and also with the customers using them has a positive impact on the quality of the software they produce.

Smart endpoints and dumb pipes

Another important property Fowler et al. [36] mention is that "microservices aim to be as decoupled and as cohesive as possible". They make a comparison between the microservice style and classical Unix* command line tools, which take some input, apply their logic and produce output that can be further processed. In the context of microservices, a single service acts as a filter in a similar sense, receiving requests via lightweight protocols like REST over HTTP or a simple message bus, applying domain logic and producing a response that returns the result. This idea differentiates the microservice architecture from concepts like the Enterprise Service Bus (ESB), which tends to put lots of business logic, like message routing, into the communication channel [36].

Decentralized Governance

Decentralized governance is about relieving teams, each of which is responsible for another service, from exhaustive internal standards and instead enables them to solve their problems in a way that best fits their individual needs. This relates to technologies like the programming languages and frameworks in use, but should also encourage teams to build and share tools in the sense of an open source model, so that others can use them for similar issues. Just like the product-oriented way of thinking, decen-

* UNIX is a registered trademark of The Open Group in the United States and other countries.

tralized governance can be considered another manifestation of the "You build it, you run it" principle [36, 61].

Decentralized Data Management

Monoliths are usually backed by a single Structured Query Language (SQL) database. Updates to SQL databases are processed in a sequential and transactional manner, with the RDBMS ensuring transitions between consistent states only. The strong consistency guarantees of SQL databases are at the expense of higher performance. The microservice style favors the idea of letting each service manage its own database and accordingly allow them to make use of different storage technologies where it makes sense. Fowler et al. [36] call this "Polyglot Persistence". Since distributed transactions are hard to manage, microservices relax their consistency guarantees towards *eventual consistency*, meaning that the propagation of updates across the whole system might take a while, leaving it in a inconsistent state for a short period of time [35].

Infrastructure automation

In order to increase confidence in the correctness and quality of software, Fowler et al. [36] recommend to automate repetitive tasks as much as possible, for instance test execution and deployments. In their opinion, CD should be applied as a means to make the deployment process literally "boring" [36]. In addition, Newman [59] brings up virtualization platforms as a crucial factor for infrastructure automation, since they allow to mostly automate provisioning and scaling of VMs for development or production.

Design for failure

In an environment consisting of many interacting components, every service must constantly be prepared for the case of another service it depends on being unavailable. This might be the consequence of an application or an entire machine being crashed. Such error conditions must be detected quickly and ideally be fixed in an automated fashion. This requires extensive logging and monitoring of a microservice landscape in order to deal with the increased operational complexity compared to monolithic applications [36].

Evolutionary design

Fowler et al. [36] regard service composition as an instrument helping developers with gaining control over changes in a strictly bounded environment. They consider it an important factor which prevents the overall application from suffering from a loss of evolutional speed. They recommend that component boundaries should be chosen with a strong focus on independent replaceability and upgradeability. As an example,

features that are only needed temporarily should be put into self-contained units, so that they can simply be thrown away once they are no longer required [36].

2.3.4 Downsides and challenges of microservices

Of course, the microservice style does not only entail benefits. Embracing this kind of software architecture also introduces some serious drawbacks that must be handled in some way:

- Moving towards microservices means getting involved with distributed systems and their inherent complexity. For example, remote calls hardly achieve the same performance as in-memory calls due to network latencies. Moreover, remote communication is constantly exposed to network link failures and crashes of processes or entire machines. In order to mitigate performance difficulties, distributed services utilize asynchronous communication protocols, which are much more difficult to handle than synchronous protocols. Facing these complexities requires software developers to adapt a new mindset and think in distributed way [35].

- Eventual consistency yields the risk of business logic operating on inconsistent information, which might lead to serious trouble. Software developers must be exceptionally careful to implement their services with regard to inconsistency issues [35].

- Defining component boundaries is a non-trivial task, as they are subject to constant change. Thus, Fowler et al. [36] recommend adhering to the evolutionary design principles and shaping them in a manner that makes refactoring them as simple as possible.

- The refactoring of service boundaries is also far from being simple. Wootton [102] denotes component boundaries as "implicit interfaces" between dependent services which demand all changes applied to a service's public API to be propagated to all consumers and hence induce undesirable coupling. Fowler et al. [36] suggest backwards compatibility strategies as an effective countermeasure, but Wootton [102] answers that this does not relax coupling between services from the business logic's perspective.

- The oftentimes large scale of microservice-based systems, as well as aspects like asynchronism and nondeterministic network behavior massively hampers testing. For that reason, such systems prefer another approach, which comprises extensive logging and monitoring of production services, as well as quickly rolling back faulty deployments if necessary. This can indeed speed up delivery, but

also poses a risk that can impossibly be taken by highly critical business segments like finance or health care [102].

- A large amount of independent services generates significant operations overhead, because additional infrastructure is needed to automate tasks like compiling, testing, deployment and monitoring that can barely be managed manually [102]. This raises the need for high-quality microservice management tooling, which however requires software developers to not only adopt a new set of skills, but also embrace the *DevOps* [35] culture which suggests close interaction between development and operations.

The main focus of this thesis lies on the last point and is about demonstrating how to construct and deploy a hybrid microservice infrastructure by means of plumbing of available open source tools. Since aspects like compiling, testing and packaging are usually covered by a CD pipeline, microservice management tools particularly concentrate on assisting development teams in concerns related to operations, including deployment, scaling and failover. The following section takes a closer look at the overall requirements that should be met by a suitable microservice infrastructure solution.

2.4 Common requirements to microservice orchestration tools

Before introducing a possible approach of building a microservice or cluster management platform, we want to give an overview of the purposes such tools actually serve as well as the most essential requirements they are expected to comply with. Over the last years, various open source projects emerged, e.g. *Kubernetes* [99], *Apache Mesos* [78] or *Docker Swarm* [27], which promise to be capable of orchestrating a large number of services in a reliable and highly available manner, with a particular focus on long-running web applications oftentimes. Despite their differences, certain similarities can still be detected regarding the problems these tools intend to solve. These commonalities identify a set of fundamental aspects that outline what microservice management essentially is about.

Resource abstraction layer

One of the main goals of microservice orchestration is to provide an abstraction layer across a set of physical machines and make them appear as a single pool of hardware resources (CPU, RAM, storage etc.) [78]. Thereby, application developers and operators shall not need to know about the details of the underlying platform since the microservice platform makes the decisions concerning the exact physical location of workload within a cluster.

On demand provisioning

The traditional infrastructure provisioning model involves developers sending predefined specifications for VMs to system administrators and then waiting some time until they can continue with their work. Cluster management tools break up this time-consuming process by encouraging the concept of developers autonomously and almost instantly allocating the computational resources they need for their purposes [59].

Increased hardware utilization

Clustering tools are based on a scheduling policy, which can be considered a set of mandatory conditions to be optimized in terms of workload placement and usually promotes hardware utilization in the most cost-effective manner. To achieve this, a variety of scheduling algorithms ranging from simple to very complex is conceivable. Additional factors determining if a server gets allocated by the scheduler are, for example, data locality, application dependencies, the server's health state and explicitly defined constraints by application developers [39].

Deployment automation

A microservice infrastructure performs automated deployments and rolling upgrades, i.e. avoiding down time while bringing new versions of applications in production. In case of a faulty deployment, it induces a interference-free rollback to the old software version and keeps the system available under all circumstances [59].

Service availability and scaling

Another primary purpose of cluster management is keeping all active services and the system as a whole available. It must be ensured services are running anywhere in the cluster while their exact physical residence is actually irrelevant. Since the different parts of a microservice-based application frequently experience different amounts of requests, another challenge is dynamically scaling up individual services which face a load peak. For this, additional resources must be allocated and used to start more instances of a service in order to sustain the system's stability. Once the load starts decreasing after a while, these instances must again be stopped and their resources must be revoked in order to be assignable afresh [59].

Service Discovery

As the number as well as the exact position of a service's instances within a cluster of machines can dynamically change over time, the question is how applications can keep track of Internet Protocol (IP) address and ports of services they depend on,

which is crucial for communication via the network [59]. Hard-coding network addresses in client code is a far too fragile approach which besides corrupts the idea of lose coupling between services. Accordingly, other solutions must be found, like passing information about network locations as environment variables [100] or integrating a Domain Name System (DNS) capabilities [27] which make services addressable by means of an unique and consistent identifier.

Isolation

In situations where a variety of applications and tasks share a common physical environment isolation mechanisms are absolutely vital to prevent processes from interfering in any way that might constitute an operational or security-related risk. Most clustering tools achieve this by leveraging container technology [59], either bringing their own APIs and containerization mechanisms or including support for third party vendors like Docker [17]. In chapter 3, we cover containers and Docker in more detail.

Authentication and authorization

Production-ready tools which are in charge of the stability and availability of a system should be accessible only by authorized employees. Furthermore, as most users solely need entrance to a subset of the entirety of functions to do their job, such tools should also follow the principle of least privilege [67], limiting a user's permissions to the bare minimum they actually need. Another security aspect focuses on different attack vectors related to cluster membership [27]: Every attempt to join the cluster should require authentication, for example by means of Transport Layer Security (TLS) certificates. A similar mechanism might be applied to prevent a compromised worker node from taking over leadership of the cluster and possibly occupying all available computing resources.

Logging and Monitoring

Section 2.3.4 already stressed the importance of employing a mature monitoring strategy in order to get a clear view of a distributed system's state. An appropriate cluster management solution should either integrate functionality to monitor infrastructure as well as application health state or provide a user-friendly API by means of which different kinds of metrics can be exported and aggregated by third party tools [59]. In the ideal case, it also offers the opportunity to gather the logging output of services and management components at a central point where it can be accessed quickly [59].

Shared and exclusive persistent storage

When using containers as a mechanism to establish a certain level of isolation between physically co-located tasks, it must be kept in mind that container state is ephemeral, which means that all changes applied to a container are irreversibly gone once it is destroyed [14]. This is fine for stateless applications, but nevertheless containerized applications sometimes need to create or modify data which is desired to survive a container crash, which is true for stateful services [83]. Another aspect is that having multiple service instances on different cluster hosts working on the same data normally requires replicating potentially large sets of files across several machines and keep the copies in sync [39]. An ideal microservice infrastructure remedies this by either shipping with shared persistent storage capabilities [85] or providing APIs for integration with appropriate third party implementations.

2.5 Challenges of hybrid microservice orchestration

We already explained that, despite its advantages, the microservice architectural style is still a trade-off. Dealing with increasing operational and organizational complexity introduces the need for orchestration infrastructure in order to make things manageable. In terms of multi-architecture environments, some special demands emerge that make microservice management on heterogeneous computing landscapes significantly different from the traditional homogeneous data center scenario, especially if a high degree of workload flexibility shall be achieved, too.

Heterogeneous platform

In section 2.1.1, we already mentioned that while data centers usually comprise a homogeneous amd64-based server landscape [5], mainframes pose a more heterogeneous type of platform. For example, IBM System z includes uses amd64 servers for as out-of-band control infrastructure, whereas the actual processor modules which are responsible for running workload are based on IBM's* s390x processor architecture [44].

Since data centers increasingly apply Advanced RISC Machines (ARM) processors next to amd64 CPUs due to their high energy efficiency [5], multi-architecture computing environments become indeed more and more common. Nevertheless, administrating hybrid microservices introduces basically two platform-related challenges: First, a microservice infrastructure must be found which is capable of orchestrating jobs across a cluster based on different processor architectures. Second, a possible solution

* Trademarks of IBM in USA and/or other countries.

must of course itself be available on all CPU architectures involved. While a growing number of open source projects integrates ARM support, this is a much greater problem in terms of mainframes, since for example s390x processors are only used in System z mainframes rather than being part of mainstream computing, so there is usually no native s390x support by most open source projects.

Diversity of workload

On hybrid computing landscapes, we basically differentiate between two categories of jobs. Generally, we assume that a job j (also called *service*) consists of n stand-alone *tasks* (also called *instances*). T denotes the set of all tasks that belong to a job j.

1. **Homogeneous jobs:** We refer to a job j as homogeneous, if the set of all tasks T is obliged to run on a homogeneous group of cluster components. Two cluster components, or machines, are considered homogeneous if they are based on the same kind of CPU architecture. Consequently, all tasks of a homogeneous job either run on amd64 or s390x systems exclusively (see figure 2.3).

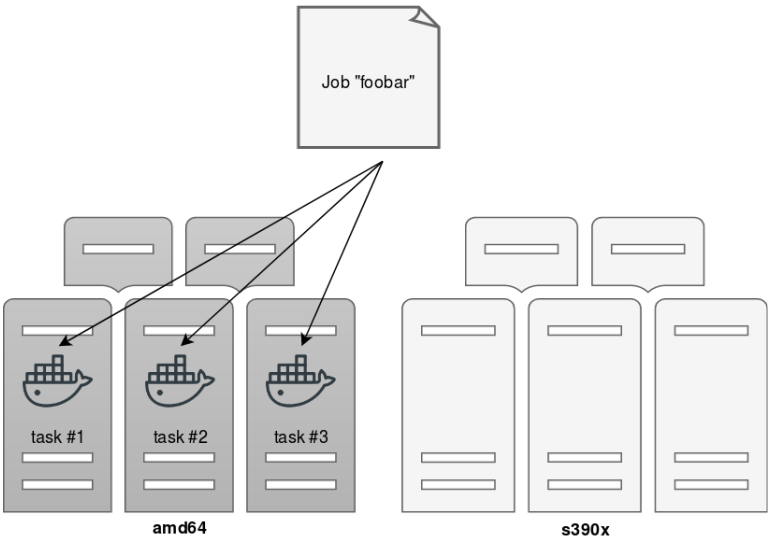


Figure 2.3: A homogeneous job's tasks are distributed across machines of the same type.

   Within a group of homogeneous cluster components, it shall be assumed that no explicit placement constraints are established, meaning that the exact location of tasks within a homogeneous group is entirely the responsibility of the scheduling algorithm in use.

2. **Heterogeneous jobs:** A job j shall be called heterogeneous, if two tasks $t, u \in T$ are principally allowed to placed on machines which differ in their underlying CPU architecture. Emphasis is placed on *principally*, because having all tasks of

a heterogeneous job being scheduled to a homogeneous group of machines is perfectly valid, whereas the opposite is not allowed.

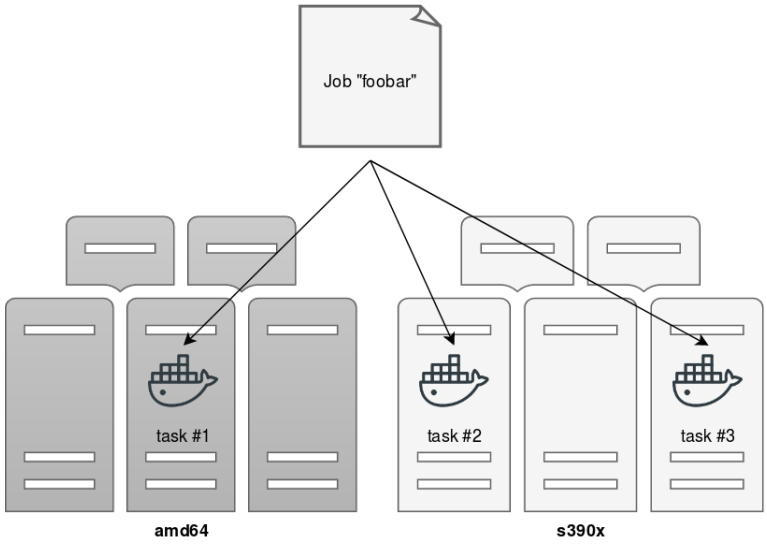


Figure 2.4: A heterogeneous job's tasks can be spread across different kinds of machines.

One of our goals is to establish a microservice infrastructure that supports a wide range of of jobs, for example cron jobs, ad-hoc jobs, batch jobs or common long-lived services. Furthermore, we want to respect the case that two jobs j and k might be interdependent, which means that j has to be finished before k can be launched or vice versa. This is useful when, for example, k has to operate on the output of j for some reason. Generally, we want scheduling to take individual application characteristics and requirements (e.g data locality, dependencies or placement constraints) into account as far as possible, which might sometimes even be a contradiction to the goal of maximum hardware utilization.

Operation at moderate scale

In terms of scale, the conditions on an IBM System z mainframe can barely be compared to classical data centers as described by Barroso et al. [5]. Although System z includes two SEs and a variable amount of processor modules and memory which can run up to 60 LPARs (status as of February 2010) [44] hosting a dedicated OS each, this is nothing compared to the enormous capacity of today's huge large-scale centers with their thousands of servers [5].

Accordingly, the space of operation on a mainframe is much more limited, which at first sight helps with reducing complexity. On the other hand, however, this also means that a suitable microservice infrastructure must be able to operate its jobs as well as itself at a much smaller scale than highly elastic data center deployments. Moreover, while the amount of resources than can be allocated is virtually unrestricted,

there is a predefined limit on mainframes depending on a predefined technical configuration.

Minimal footprint

With respect to the fixed amount of resources available on mainframes, we consider it very important that a microservice infrastructure has a footprint (regarding CPU, memory and storage) which is as low as possible, in order to leave as much resources for service deployments as possible. This might be also desirable in the context of data center deployments, but we think this point deserves special attention with regard to a mainframe's bounded amount of resources.

2.5.1 Summary and prioritization of requirements

After having explained the most important differences between traditional data centers and mainframes in terms of microservice orchestration, we will now give a brief overview of the requirements of a microservice infrastructure as they will be treated throughout this thesis, ordered in two different categories. Thereby, the first category summarizes the requirements we considered critical for the development of our prototype, so they should finally be met as far as possible. The second category contains demands related to aspects like security and monitoring. We want to emphasize we do not consider them second-class requirements as these topics are crucial for operating microservices. However, we found it reasonable to limit the scope of our work by delaying our work on these aspects until we achieved building a basically functional prototype. In this way, we were able to focus on the main issues caused by hybrid clustering at first.

Mandatory requirements

- Representation of an abstraction layer for available hardware resources (RAM, CPU, disk space, network etc.).
- Autonomous and transparent on demand resource allocation for developers and/or operators for their deployments.
- Scheduling of tasks with respect to optimal hardware utilization and optional user-defined invariants, for example placement constraints.
- Automation of deployments, upgrades and rollbacks.
- Assurance of service availability and dynamic scaling in accordance with experienced workload.

- Dynamic service discovery and provisioning of a level of indirection for inter-microservice communication.
- Isolation of service instances from host systems by means of OS-native mechanisms, for example containers (see section 3.5).

In addition, the hybrid mainframe ecosystem use case leads to some special demands that must also be considered:

- Aggregation of machines based on divergent CPU architectures into a transparent platform for heterogeneous tasks, i.e. tasks which can be placed at an arbitrary location within a multi-architecture computing environment.
- Optional deactivation of platform transparency for homogeneous tasks which are, for some reason, bound to a certain type of machine, like services that can only run on a LPAR within a System z mainframe.
- Integrated support for long-lived services, batch jobs, ad-hoc jobs which may or may not be interdependent, and scheduling according to application-specific characteristics and requirements.
- Reliable operation at moderate scale induced by a mainframe's much smaller size and scope compared to modern large-scale data centers.
- Low footprint to leave as much resources as possible to service deployments.

Optional requirements

- Authentication and authorization for administrators and individual computing nodes joining the microservice management cluster, following the principle of least privilege.
- Comprehensive monitoring, either built-in or by attaching third party tools via APIs, for infrastructure components as well as deployments.
- Support for stateful applications and storage sharing between service instances by means of both exclusive and shared persistent volumes.

Chapter 3

A microservice infrastructure based on Apache Mesos

In this section, we provide an in-depth overview of the components our microservice infrastructure prototype is based upon. We start with a short comparison between several popular orchestration tools, justifying our decision to stick with Mesos for our hybrid target environment. Then, we step through Mesos as well as its peripherals and dependencies and explain the roles they play from the overall system's point of view.

3.1 State-of-the-art microservice orchestration

Currently, the probably most dominating tools for microservice management are Docker Swarm [27], Kubernetes [99] and Apache Mesos [78]. Their most frequently mentioned area of use is related to highly available deployments in homogeneous data center environments. Thus, we throw a short glance at these tools and evaluate their basic eligibility for hybrid computing platforms.

3.1.1 Relevant tools for microservice management

Docker Swarm

Since the 1.12 release, the Docker engine comes with built-in cluster management support and facilitates orchestrating services across a set of nodes without the need for any other third party dependencies. Docker in high availability mode is usually referred to as *Docker Swarm*. Having started as a simple container runtime, Docker has continuously evolved into a fully-integrated ecosystem for distributed applications that includes many features like service discovery, load balancing and networking that would otherwise require the use of additional tooling [27]. It should be noted, however, that Docker can only handle *containerized* applications (i.e. services running

within a Docker container) and is not able to manage or execute other formats, for example raw binaries. Although Docker and thus Docker Swarm is completely open source and under great influence of its community, only some parts of the product are actually held by an independent foundation. The strategic direction with respect to the project's evolution is mostly set by Docker Inc., the company behind it.

Kubernetes

Kubernetes considers itself "an open-source system for automating deployment, scaling, and management of containerized applications" [99] which is, in a sense, very similar to Docker Swarm. The most significant difference between them probably is that Kubernetes by itself does not bring its own container runtime, but instead leaves this to third party providers like Docker. Currently, it is the only cluster management tool implementing the concept of *pods*, which denotes a group of physically co-located containers forming a logical unit [100]. The Kubernetes project has originally been initiated at Google and is now hosted by the Cloud Native Computing Foundation (CNCF), a sub-organization of The Linux* Foundation.

Apache Mesos

According to its documentation, Mesos is as a "distributed systems kernel" [78] that provides resource management APIs for applications in a similar manner as the Linux kernel does, but at another level of abstraction. Instead of only managing a single machine, Mesos can be employed to perform application orchestration and scheduling across entire computing platforms. However, in contrast to Kubernetes or Docker, Mesos is not restricted to container deployment. Moreover, it does not integrate a wide range of functionality like service discovery but, as an alternative, offers APIs and leaves concrete implementation of lots of behavior to custom components. In this way, Mesos achieves to provide a maximum of flexibility while keeping itself lightweight. Mesos has been introduced in 2011 by Hindman et al. [39] at the *8th USENIX Symposium on Networked Systems Design and Implementation* and meanwhile is hosted by The Apache Software Foundation (ASF), which demands the project's further development to be neutral towards commercial, company-specific interests.

3.1.2 Eligibility for clustering in hybrid computing environments

If an application orchestration solution is suitable for managing hybrid platforms primarily depends on if the tool itself can be run on each of the different types of ma-

* Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

chines in question. As for the IBM System z s390x architecture, support for the tools mentioned above is - to anticipate it - at most work in progress and surely additional efforts are required to get things operational. Nowadays, the probably most progressive technology in terms of multi-architecture support, especially when it comes to System z, is Docker (see section 5.2).

Another important aspect is that hybrid cluster management must be able to condense a diverse set of machines into a single system, that can execute architecture-agnostic jobs without the platform's inherent heterogeneity shining through. At the same time, it must still be aware of the architectural differences between machines in order to add placement constraints to jobs where the sort of machine they should execute on does actually matter.

Docker Swarm, Kubernetes as well as Mesos integrate the concept of user-defined node labels, that their corresponding schedulers can use as optional selectors to choose appropriate target machines for launching tasks. Even though this sounds relatively simple, labels pose a powerful mechanism which is - we will see - exactly what we need to split up a multi-architecture cluster into n pools of machines of the same kind. In theory, this functionality is already sufficient for dealing with both homogeneous and heterogeneous jobs, as it allows the definition of placement constraints. Section 5.7 shows that there are, however, additional challenges that must be considered in practice.

So far, it can be seen that the most common mircoservice management platforms already supply basic functionality which makes them a candidate for hybrid environments, while constantly expanding their focus towards multi-architecture support anyway. Coming from IBM System z and s390x, going with Docker Swarm seems to be the best way to get started. The next part, though, provides a justification why our choice has still fallen on Mesos.

3.1.3 The rationale behind using Apache Mesos

There are essentially three arguments why we preferred Apache Mesos over Docker Swarm and Kubernetes. In the first instance, we already had a rough idea of what Mesos is about before we started our work on this thesis. Admittedly, this is also applies to Docker, so Kubernetes was the first option to be excluded due to missing familiarity.

Another important point is that Mesos was known to be a popular and field-tested tool that has successfully been used by companies and organizations like Twitter, Uber, Airbnb or Cisco [81] over a long period. In contrast, Docker only provides information about customers using their fee-based products [17].

This directly leads to the next point pleading for Mesos. As a project hosted by the ASF, it is ensured that, as already mentioned above, the future development is

not dominated by the interests of a single company but rather serves the good of the community. Unlike Mesos, the evolution of the Docker ecosystem is mainly subject to a single company, namely Docker Inc. This should be a major consideration when searching for an option that is ideally able to serve as the foundation of a sustainable prototype.

Altogether, the most essential advantage of Mesos is that, unlike Docker and Kubernetes, it does not raise a claim to be full solution to all cluster management issues thinkable. Instead of integrating a large number of features for all kinds of problems, the philosophy behind Mesos is to bundle a minimum of functionality that can be extended as needed. What that means is Mesos prefers delegation over integration, leaving the implementation of possibly varying behavior to third party tools which leverage the Mesos APIs. In this way, custom features can simply be added if necessary while the Mesos core itself can stay lightweight [39]. On the other side, comprehensive tools like Docker Swarm carry the risk of shipping with a vast number of features most of which may not actually be required for the majority of use cases. Indeed, Docker Swarm and also Kubernetes expose some plugin APIs [25, 98] which allow some parts of the predefined behavior to get customized (for example, adding custom volume drivers to Docker engine), but they are still far off Mesos' flexibility.

The most relevant example for this is that Mesos is based on the philosophy of decentralized scheduling [39], giving its users the opportunity to exactly implement the scheduling behavior which fits best for a certain scenario. One of our goals is to end up with an infrastructure that is suitable for a wide range of application types (long-lived services, cron jobs, batch jobs etc.) which possibly require to optimize scheduling and task distribution towards different criteria, like data-intensive workload that tries to achieve high data locality. Therefore, we think that the Mesos approach fits best to our needs as it enables application-aware scheduling.

Docker Swarm, on the other hand, does not implicitly implement the notion of various kinds of jobs, but instead is designed with a special focus on long-lived services and the goal of keeping them available as long as possible. This does not necessarily mean that cron jobs or interdependent jobs cannot be run on top of Docker Swarm. However, that would require an additional custom management layer on top of Docker Swarm which takes care of this. In our opinion, this is a clear sign that Docker Swarm does not really meet our requirements.

Further evidence of Mesos' flexibility is that, in contrast to Docker and Kubernetes, it is not limited to containerized deployments or Docker containers in particular. Instead, the deployment format of applications can be arbitrary and is entirely up to third party scheduling components.

3.2 Mesos fundamentals and overview

The fundamental ideas and technical concepts behind Apache Mesos have been described in detail by Hindman et al. [39] in their paper released in 2011. This section summarizes the central characteristics of Mesos as one of the key components our hybrid cluster management system we present in this thesis is built upon.

3.2.1 Basic idea

Mesos has been designed around the observation that today, clusters are increasingly shared between different "frameworks", a designation Hindman et al. [39] use to describe "software system[s] that manage[.] and execute[.] one or more jobs on a cluster". Each framework is tailored to a certain kind of application, for example large-scale web applications or data-intensive scientific applications. Concrete examples are, amongst others, Hadoop and MapReduce. Hindman et al. [39] argue that, since no framework is optimal for all use cases, the need to run multiple frameworks on the same infrastructure ("cluster multiplexing" [39]) is a logical consequence. In their opinion, high utilization and efficient data sharing can, however, only be achieved if resources are shared across framework boundaries, which is not the case if clusters are logically or even physically split up between frameworks. Thus, the paper [39] justifies the necessity for Mesos as "a thin resource sharing layer that enables fine-grained sharing [...], by giving frameworks a common interface for accessing cluster resources".

The most significant technical innovation that came with Mesos was the absence of a single central scheduler. Considering Docker Swarm [27] as a counterexample, it relies on a global scheduler that is responsible for managing all tasks of a job. A global scheduler makes scheduling decisions based on a centralized view of a cluster, including the entirety of all framework requirements, available resources and user-defined policies. Although this system-wide knowledge can be applied to develop an optimal scheduling policy that converges the overall cluster utilization towards a global maximum, it may not be optimal from a single framework's point of view which, for example, suffers from poor data locality. This can be the case if a framework wants to run multiple jobs on the same dataset, but has to make several copies of it as the respective tasks are scheduled to different machines. Moreover, it must be taken into account that new kinds of frameworks with their own scheduling demands will emerge in the future, forcing centralized implementations into constantly keeping up with the latest developments [39].

Thus, the Mesos philosophy is to push accountability for scheduling to the frameworks, with each framework managing a set of jobs with similar scheduling needs. This is achieved through a concept called *resource offers* [39]. A resource offer is a bundle of free computing resources (inter alia CPU and RAM) which Mesos passes on to

frameworks. A framework that receives such an offer can use free resources in order to run tasks. Hindman et al. [39] formulate this as follows: "Mesos decides *how many* resources to offer each framework, [...], while frameworks decide *which* resources to accept and which tasks to run on them".

3.2.2 Architecture

In essence, Mesos is made up of three components: A Mesos *master* that manages Mesos *agents* running on each node in the cluster, and one or more Mesos *frameworks* that launch tasks on these agents (see figure 3.1).

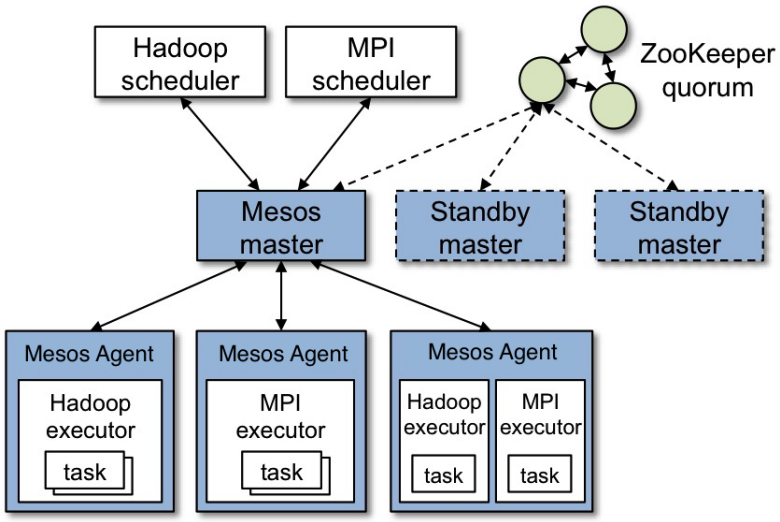


Figure 3.1: Mesos architecture diagram [81].

The Mesos master's responsibility is to bundle free resources, about which it gets constantly notified by the Mesos agents, into resource offers, which are basically lists of free resources on agent nodes, and to pass them on to registered frameworks. It thereby follows a certain policy in the form of an interchangeable *allocation module*, that determines how unused resources shall be divided up between frameworks. Figure 3.1 further shows that each Mesos framework consists of a *scheduler* as well as an *executor* process. The scheduler's purpose is to register with the Mesos master in order to receive resource offers, while an executor can be regarded a control process that runs and observes a framework's tasks [39].

The basic principal behind resource offers can be seen in figure 3.2. In step (1), agent 1 informs the Mesos master about free resources, whereupon the master invokes the allocation module which determines that the resources shall be offered framework 1. Subsequently, the master creates a resource offer in step (2) and sends it to framework 1, which, as next step (3), answers with a list of task descriptions it would like to run on these resources. At least, the master passes on the task list to the agent node,

which then allocates the resources and launches the framework's executor process that in turn starts the actual tasks [39].

For the purpose of isolation, Mesos supports running both executors and tasks in containers, using a mechanism which Mesos calls *containerizers*. Currently, the Docker containerizer (requires the Docker engine to be installed on the agent) as well as the Mesos containerizer (Mesos native container technology without external dependencies) are at disposal. Wrapping up workload in containers comes with several advantages, like isolation, ease of resource usage limitations and reusability [81]. Containers are discussed more detailed in section 3.5.

As a complement, it is important to mention that a framework scheduler is not obligated to accept resource offers. Given that a certain offer does not match a framework's demands, for example because it only wants to launch tasks on a particular agent node to achieve data locality, it is free to reject it and wait for another one to be sent by the Mesos master [39].

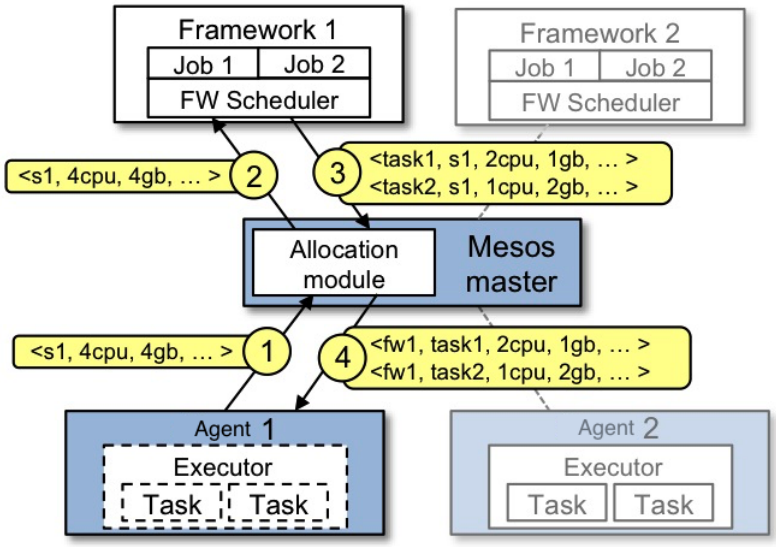


Figure 3.2: The resource offer mechanism in Mesos [81].

In order to short-circuit this process, Mesos allows frameworks to set *filters* on the master. A filter is a simple Boolean predicate that the master can evaluate to check if a certain framework would reject an offer anyway. A sample filter might look like "only offer resources from nodes in List L to framework f" [39].

3.2.3 Fault tolerance

It is absolutely vital for a microservice infrastructure to be extremely fault-tolerant. As for Mesos, there is a number of critical points where potential failures could endanger the stability of the entire cluster management platform and also of its deployments.

Fortunately, Mesos implements sophisticated failover strategies to ensure availability and reliability.

Mesos master failover

The master is at the heart of Mesos, since all frameworks rely on for obtaining computing resources. In order to be able to intercept the loss of the master process, Mesos allows for running multiple master instances in what Hindman et al. [39] call a "hot-standby configuration" (see figure 3.1), with a single node acting as the leader. If the leading master fails, a new leader is elected to which the frameworks and agent nodes can reconnect afterwards. It is recommended to run at least three Mesos master instances as an ensemble for the purpose of fault tolerance. The Mesos masters employ ZooKeeper [92] for leader election (see section 3.4).

The leading master stores cluster information as *soft state*, meaning that it is kept in volatile RAM only. Losing it in succession of a crash failure or a power outage is not critical, though, as the new leader can simply restore its state by means of the information stored at the agents as well as the schedulers. The information ephemerally held by the leader includes a list of active agent nodes, a list of active frameworks and another list capturing currently active tasks [39].

Nevertheless, there are still some edge cases that might lead to inconsistencies, for example if an agent fails while a master leader election is currently in progress and no leader is available for a short period of time. As soon as the remaining agents reconnect to the new leader, the failed agent node would literally be forgotten by the leading master, whereas the framework would still assume the crashed agent's tasks to be running. Thus, the ensemble of master nodes applies a persistent registry to keep track of active agent nodes, so that agent failures can reliably be recognized at any point in time. This registry can be stored in memory (which is not recommended for production setups) or by using the *Mesos replicated log* library [91], which facilitates the creation of fault-tolerant append-only logs that store data in a safe, replicated and especially durable fashion across multiple master nodes. In order to be able to guarantee a strong consistency model, the replicated log implements the *Multi-Paxos* consensus algorithm, which requires a majority of Mesos master nodes to be live [81, 43].

Mesos agent failover

If a Mesos agent process exists, for instance due to a bug or because it shall be upgraded to a new version, it loses contact to all executors it has been managing. The leading master grants it a defined period of time to recover (75 seconds by default) before it marks the agent node as unreachable. The leader then notifies all affected frameworks of the loss, which are free to determine their own error handling strate-

gies (for example scheduling replacement tasks). From the moment when the agent crashes, the executors also wait for a certain amount of time (15 minutes by default) and self-terminate if they cannot reconnect within this period [80, 43].

Assuming that a crashed agent successfully recovers and reregisters with the leading master, its standard behavior is to kill all running executors and let the frameworks reschedule them. To avoid the deletion of existing executors when the agent is restored again the frameworks can activate *checkpointing*, which impels the agent to periodically write executor state to disk. On a subsequent restart, the agent can reload its state from the previously checkpointed information and reconnect to the executors which are still running [80].

While the official documentation points out that wiping out all executors on agent restart is the default strategy for the latest Mesos versions [80], Ignazio [43] describes caching and recovery of executor state as the standard approach for earlier releases.

Framework executor failover

Mesos supports native task checking by leveraging the executors running on agent nodes to perform health checking of tasks. If a task fails, the leading master will be notified by the corresponding executor and immediately passes on the status updates to the affected framework schedulers. In the event of an executor crash, the leading Mesos master will also be advised and informs the scheduler right away. In both cases, each framework must define its own response to the loss of tasks, for example the scheduling of replacement tasks [90].

Framework scheduler failover

The third component that could possibly fail is the framework scheduler. Therefore, frameworks are free to register multiple scheduler instances with the master ensemble, with one scheduler being active and the other ones running in standby mode. If the Mesos master finds the active scheduler to be unavailable, it notifies one of the standby instances which immediately takes its role. It should be noted that Mesos does not dictate the frameworks how their scheduler state shall be persisted and shared across multiple instances. Once again, this concern is left to the frameworks and can be implemented in different ways, as the following comparison of *Marathon* and *Aurora* will show [39].

3.3 High-level comparison between Marathon and Aurora

This section takes a glance at two different Mesos frameworks, namely *Marathon* [50] and *Apache Aurora* [72], which we consider suitable candidate frameworks for flexible and reliable microservice orchestration. Because both frameworks are developed as

open source projects with extensive documentation [48, 74] that covers many details, the focus will be on the most significant similarities and differences. Instead of sticking with a single framework, we decided to take a broader approach by examining how different implementations compare to each other. This comparison is especially interesting because on the one hand, Marathon is backed by Mesosphere, Inc., which offers it in a standalone version or as part of an integral cluster management stack called *DC/OS* [49]. Aurora, on the other hand, belongs to the Apache Software Foundation and is entirely community-driven project [72].

Marathon and Aurora will also be evaluated with respect to the demands on a microservice infrastructure we established in section 2.5.1. Some of them, like providing an abstraction layer across distributed physical hardware, are already fulfilled by Mesos itself.

3.3.1 Use case

Both Marathon and Aurora imply the idea of *jobs* as an abstraction for a collection of identical tasks, which is important because Mesos only operates at the notion of tasks [73]. They both characterize themselves as platforms especially for long-running services in the first place. Besides, Aurora ships with support for cron jobs (i.e. tasks that shall be executed periodically) [74], while Marathon integrates functionality targeting job dependencies [51], which is currently limited to keep a group of jobs in sync in terms of state and size. So neither Marathon nor Aurora cover the whole spectrum of job types we originally defined. Though, that issue can theoretically be bypassed by running multiple frameworks side by side on top of a single Mesos deployment and combining their capabilities.

3.3.2 Architecture

Marathon or rather the Marathon scheduler can be deployed as a single artifact which, as a Scala project, depends on a Java* Runtime Environment (JRE) to be available. It does not bring its own executor but instead relies on the pre-defined *Mesos command executor* [51] for launching tasks on agents.

Aurora goes a different way here and is split up into multiple components that have to be deployed independently. Apart from the scheduler, Aurora requires a custom executor implementation to be present on each agent node, and the Aurora command-line interface (CLI) is another separate deployment unit. Along with the fact that the Aurora components are partially implemented in different programming lan-

* Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

guages (scheduler in Java, client and executor in Python) this makes setting up Aurora very cumbersome [73].

3.3.3 Resilience and reliability

Mesos enables frameworks to operate in high availability mode by letting frameworks register multiple scheduler instances with the leading master, one of which is elected as the active scheduler, while the remaining ones stay in standby mode, waiting for the leading scheduler to fail and possibly taking its position [81].

Mesos leaves the concern of persistently saving scheduler state in order to allow it to survive a crash of the leading scheduler to frameworks [39]. Marathon and Aurora choose different strategies in oder to guarantee that in case of a scheduler failover, the new master can seamlessly continue from the point where the old one failed. Marathon, for example, creates dedicated ZooKeeper nodes (see section 3.4) which keep track of scheduler state. This is a reasonable approach since Mesos requires ZooKeeper as a dependency anyway [53].

On the other hand, Aurora waives relying on ZooKeeper for persisting its state and rather refers to the Mesos replicated log (see section 3.2.3) for that purpose [75]. What speaks in favor of this approach is that, compared to how Marathon deals with persisting state, an additional dependency between the framework and a third party tool like ZooKeeper is avoided.

3.3.4 Application deployment and scaling

Marathon as well as Aurora both promise deployment and scaling of applications to be simple. They come with various interfaces for creating, updating and watching jobs. Marathon, for example, offers a sophisticated UI where all activities related to job administration can be performed [50]. Aurora also ships with an UI which is designed to be read-only, though. That means its purpose is limited to observe the state of jobs and their tasks. Deployments and updates can only be done via the Aurora CLI and the Aurora Domain Specific Language (DSL) [73]. Marathon does not have a CLI but instead provides a RESTful HTTP API [50] which allows the definition of jobs in JSON format.

In terms of functionality, both frameworks supply a very similar set of basic features which let users define jobs, transparently scale them up/down and and finally delete them again. Since Aurora requires a dedicated DSL to be learned, interacting with Marathon through its UI feels a little bit more user friendly.

With regard to supported deployment formats, both Marathon and Aurora are able to launch tasks by either directly executing binaries on the agent hosts or starting tasks as Docker or Mesos containers (see section 3.5) [89].

3.3.5 Service discovery

Keeping different services loosely coupled means giving them the opportunity to refer to each other by a unique identifier instead of hard-coded network addresses. This identifier states a permanent reference by which services can reach each other even if their exact locations within the network changes over time. Ideally, given that more than one instance for a certain service is deployed, requests issued to that service are transparently and evenly load balanced across all instances.

Service discovery can be implemented in different ways. Marathon, for example, delegates the translation of an identifier into a network address and port to a standalone component named *Mesos-DNS* [55]. Mesos-DNS constantly fetches the list of active tasks from the Mesos master using the Mesos APIs, and creates appropriate DNS records which make Mesos deployments addressable based on the naming schema `<app-name>.<framework-name>.domain`. While the name of a framework can be defined at startup time, `domain` defaults to *mesos* if not specified otherwise [55].

For service discovery via Mesos-DNS to work, all client host's networking settings need to be changed to point to Mesos-DNS as their primary nameserver, which causes significant configuration overhead [55].

According to its documentation, Aurora has also started to integrate Mesos-DNS support for service discovery [77]. Previously, ZooKeeper was (and still can be) used to perform service discovery. Under the hood, this works by having the framework executors publishing tasks and their network addresses in what Aurora calls *ServerSets*, which are based on the ZooKeeper-native *group membership pattern* [77]. As this approach seems deprecated, the details shall be skipped.

3.3.6 Persistent storage

Marathon leverages the Mesos persistent volumes capabilities in order to allow tasks to permanently store their state on disk beyond their lifetime [48]. Persistent volumes are especially practical for stateful applications [54], which can restore their state even if they crash, but also for sequential batch jobs where one job processes data produced by a previous one. By default, persistent volumes can only be accessed by tasks managed by the the same executor [83].

Besides, Mesos provides support for shared persistent volumes [85] in order to share data sets between tasks running on executors. However, it is still required that tasks sharing a volume reside on the same agent node. From a framework perspective, the Marathon documentation states that persistent volumes are not yet fully supported at the moment [54]. As for Aurora, the documentation currently lacks any information about the integration of persistent volumes. The only hint related to this

topic is a ticket (AURORA-1713[1]) in the JIRA issue tracker that suggests the integration of the corresponding functionality.

3.3.7 Scheduling constraints

Both Aurora and Marathon allow for job definitions being supplemented with scheduling constraints. These constraints can be matched against custom labels that must be added to Mesos agents at startup time. The concept of node labels has already been introduced in section 3.1.2. Scheduling constraints make it possible to split up the entirety of agents into subsets which, for examples, run on machines sharing the same CPU architecture. In this way, placement constraints for homogeneous jobs that are restricted to only execute at particular places within a divergent platform can easily be defined. Both frameworks offer similar ways of scheduling constraint definition [52, 76], yet the Marathon approach based on various operators (see section 5.7) appears to be more powerful, while Aurora's constraint capabilities seem rudimentary.

3.4 Coordinating Mesos with Apache ZooKeeper

3.4.1 Why Mesos needs coordination

As a critical component of a microservice infrastructure, Mesos is implicitly designed to be reliable and highly available. This is particularly important for the Mesos masters, since all agents and frameworks depend on a operational leading master in order to get their job done. Mesos allows n master nodes to form an ensemble consisting of exactly one *leader* and, given that every node is healthy, $n - 1$ *standby masters* [43]. Because it is unknown a priori which master node will be the elected leader when Mesos is initially started and because the distribution of roles between nodes might change over time, it is obvious that the configuration state of Mesos is highly dynamic. That brings on some problems that must somehow be tackled:

- How can the Mesos masters can come together and form an ensemble?
- How do the Mesos masters agree on a single leader?
- How can Mesos agents and framework schedulers learn which one is the leading master and where it lives in the network (IP address and port)?
- How can Mesos agents and framework schedulers determine if the leading master has failed? How can they reconnect to a new leader?

[1] https://issues.apache.org/jira/browse/AURORA-1713

Of course, Mesos could bring its own library for dealing with distributed consensus and synchronization, but, due to high availability reasons, would also have to handle configuration data replication and consistency. The increasing complexity resulting from this approach would, however, be in contradiction with the design principle of keeping the Mesos core lean and lightweight. Thus, Mesos uses ZooKeeper to implement custom coordination primitives while delegating replication and persistence of configuration information [43].

In the following, we will explain the purpose and some technical details of ZooKeeper and then walk through the questions above and see how ZooKeeper helps with answering them.

3.4.2 Intent and purpose of ZooKeeper

Hunt et al. [41] describe ZooKeeper as "a service for coordinating processes of distributed applications". However, ZooKeeper itself does not integrate any ready to use coordination mechanisms, but instead is designed around the goal to provide an API that enables clients to implement arbitrary complex primitives for different kinds of coordination needs at the client-side. Common coordination needs include [41]:

- **Configuration:** Sharing lists of static and/or dynamic parameters between multiple processes.
- **Group membership:** Informing a process belonging to a certain group of the other members and which one of them are currently alive.
- **Leader election:** Making multiple processes agree on role allocation, i.e. which one is the leader.

Instead of being a service with a special focus on only one of these points, ZooKeeper aims for enabling "multiple forms of coordination adapted to the requirements of applications" [41]. Additionally, it ensures safe and reliable replication of configuration data across multiple ZooKeeper servers [41]. ZooKeeper frees different applications from dealing with these configuration concerns on their own and facilitates sharing of a single service instead of re-implementing similar behavior over and over again.

While coordination primitives are also offered by alternative services like the Chubby lock service [8], ZooKeeper prefers wait-free data objects over blocking lock primitives in order to achieve high performance and throughput under read-dominated workloads [41].

3.4.3 ZooKeeper internals

Hunt et al. [41] stress that, even though "ZooKeeper seems to be Chubby without the lock methods" at first sight, it considerably differentiates from Chubby [8], as it is

built upon wait-free data objects instead of blocking primitives such as locks. These data objects, which are also called *znodes* in the context of ZooKeeper, are arranged in a hierarchical manner similar to the structure of filesystems. Figure 3.3 illustrates a ZooKeeper data tree, which is made up of two subtrees, one that belongs to *app1* and another one for *app2*. Clients can issue calls via the ZooKeeper client API in order to create, read, edit or delete znodes within a data tree [41].

In the first place, znodes pose an abstraction for the implementation of coordination primitives and are not designed for general data storage. Nevertheless, they can also be used to store small amounts of data (max. 1 MB by default), for example meta-information like dynamic configuration parameters. ZooKeeper makes a distinction between two types of znodes [41]:

- **Regular:** Regular znodes are explicitly created and deleted by clients.
- **Ephemeral:** Ephemeral znodes are created by clients, which are free to also delete them explicitly or leave that to the system. They are removed automatically as soon as a client session ends.

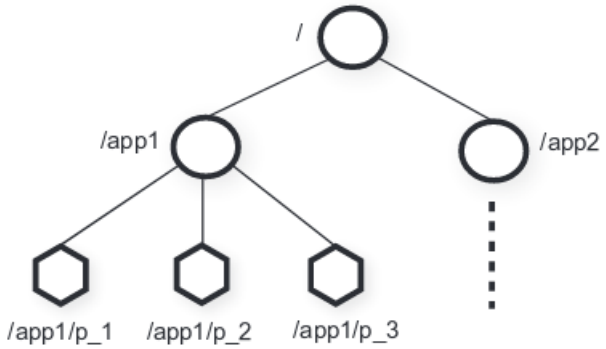


Figure 3.3: ZooKeeper organizes znodes hierarchically [41].

On znode creation, additional flags can be set which define a znode's properties. The *SEQUENTIAL* flag causes a monotonic increasing sequence number to be appended to a new znode's name. This is shown in figure 3.3, where *app1* created three znodes suffixed with the value of an increasing counter. The *WATCH* flag, on the other hand, registers a one-time trigger for a certain znode and associates it with a client session. In case the corresponding znode is deleted or modified, each client watching that znode gets notified by ZooKeeper. In this way, clients can be kept informed about ZooKeeper updates without polling [41].

As part of critical infrastructure, ZooKeeper puts emphasize on high availability, which is achieved by replication of the in-memory data tree across an ensemble of

servers. For the purpose of consistency between replicas, the ZooKeeper server group implements a leader-based atomic broadcast protocol called Zab. An extensive discussion of the Zab protocol is beyond the scope of this thesis, but it shall be noted that it helps with providing two important request ordering guarantees [41]:

- **Linearizable writes:** All write requests (i.e. requests that modify ZooKeeper state) from all clients, which may or may not run concurrently, are sequentially ordered in a manner that respects real-time precedence. Herlihy [38] gives a in-depth explanation of linearizability as well as its importance as a correctness property for concurrent systems. Informally, its impact on ZooKeeper is that all write requests must be forwarded to the Zab leader, where they are totally ordered. The Zab protocol ensures that all writes are applied to a majority of replica servers in exactly the same order [41]. Given that the number of ZooKeeper servers is n, the minimum number of servers q which must acknowledge a write operation can be calculated as follows: $q = floor(n/2) + 1$.
- **FIFO client order:** All requests (reads and writes) from a certain client are executed in a first-in-first-out (FIFO) fashion, meaning they are processed in exactly the same order as they were sent. In this context, a request being *processed* means that, in case of a write operation, it actually takes effect on the internal state of ZooKeeper rather than being solely acknowledged.

ZooKeeper utilizes various sophisticated techniques to maximize performance. It is inherently optimized for read-dominant workloads and thus uses Zab to totally order only write operations. Read operations, on the other hand, do not undergo an atomic broadcast, but instead are processed locally by the replica a client is connected to. While this restriction bears the risk of clients receiving stale data from a replica which has not seen the latest updates, it enables highly performant reads, though. Additionally, a client-side cache is used to further improve read performance. ZooKeeper does not explicitly manage these caches, meaning they are, for example, not invalidated by ZooKeeper itself in case of updates. This is left to the clients, which can make use of the watch mechanism to get informed about changes they are interested in [41].

As the previous explanations are completely sufficient to describe how Mesos can benefit from ZooKeeper, we refer to the paper written by Hunt et al. [41] which covers ZooKeeper in much more detail.

3.4.4 How Mesos uses ZooKeeper

Mesos relies on ZooKeeper for different aspects of coordination between the Mesos master group on the one side and the agents as well as the frameworks on the other,

but also between the servers of the master ensemble themselves. This section gives a short introduction to the implementations of the corresponding primitives.

Election of leading Mesos master

When Mesos is operated in high availability mode, all servers participating in the group of Mesos masters must consistently and reliably agree on a single leader. At any point in time, there must be at most a single master node which acts as the leading master. We say *at most* deliberately, because there might be short periods between the failure of a leader and the takeover of a new one during which the leader election process takes place and no leading master is available.

The leader election algorithm is not explicitly described by the Mesos maintainers. Instead, they refer to a collection of ZooKeeper "recipes" [94] that are effectively reference implementations of coordination primitives. From these blueprints, the leader election algorithm can be derived to principally work as follows:

1. On startup, each Mesos master m creates an ephemeral znode z with the SEQUENTIAL flag set under the `/mesos` path in the data tree. In this way, each master is associated with a uniquely identifiable znode `/mesos/m_i`, where i is a monotonic increasing sequence number.

2. Each master node m queries all children C of `/mesos` and checks if its znode z has the lowest sequence number in C. If so, then m is the elected leader l. Otherwise, it is a standby master s and starts watching the next znode `/mesos/m_j` in C where j is the largest sequence number such that $j < i$.

3. If any standby master s receives a ZooKeeper notification because an observed znode has been deleted, it again queries C and repeats the sequence number check from step 2. If the znode `/mesos/m_i` of the leader l has vanished, standby master s with the next znode sequence number $j > i$ learns it is the new leader.

Figure 3.4 illustrates the `/mesos` ZooKeeper subtree with three active Mesos master servers forming a quorum. If the current leader *master 1* dies, *master 2* will be notified as soon as znode `/mesos/m_1` is gone (will be removed after a timeout has expired) and takes over leadership instantaneously [94].

As for the framework schedulers, which also perform leader election, the documentations of Marathon and Aurora merely refer to the same ZooKeeper recipes [94] as the Mesos design documents do. The basic algorithm can therefore be assumed to work similarly to the procedure presented above.

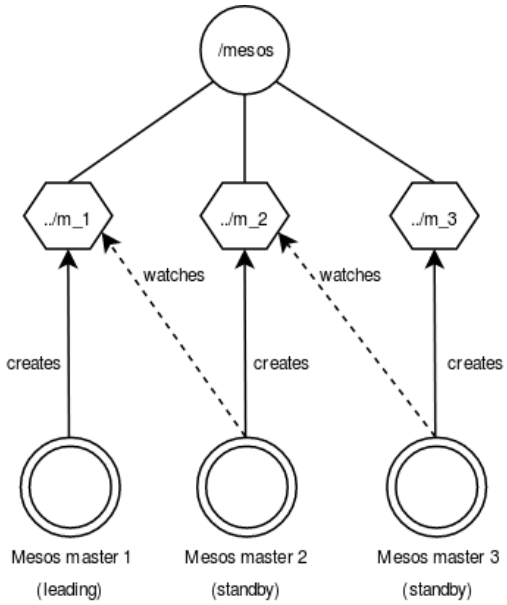


Figure 3.4: The Mesos masters use ZooKeeper for leader election.

Discovery of the leading Mesos master

The second substantial problem is about how the Mesos agents and frameworks can determine which master is the current leader and by which network address the leader can be reached. Again, the documentations of Mesos, Marathon and Aurora point to the ZooKeeper recipes for technical details with regards to leader detection. With the knowledge of how the Mesos master group manages to agree on a leader, we can, however, deduce how their `/mesos` ZooKeeper subtree can help with keeping peripheral components informed of the ensemble's state.

Since, as a convention, it is known that the Mesos master which holds the znode with the lowest sequence number must be the current leader, frameworks and agents can use that knowledge to determine the leading master by simply retrieving all the children C of `/mesos` and look for the znode with the lowest sequence number in C. Because a leader's network location (IP and port) is unknown a priori and might even change over time, Mesos applies the concept of a *rendezvous znode* [94] in order to allow Mesos agents and framework schedulers to lookup the leader's network address dynamically. A rendezvous znode is just a usual znode at which the leading Mesos master stores its current network meta data, which can then be retrieved by clients. For simplicity, the Mesos masters use their leader election znodes `/mesos/m_i` for that purpose. This concept enables client applications to stay loosely coupled to Mesos.

3.5 Workload isolation with Linux containers and Docker

Section 3.2.2 already outlined the different ways in which Mesos utilizes containers briefly, concretely mentioning containerization of not only tasks but also executors. Up to now, no accurate definition of containers and their advantages has been given, and is has neither been explained how Docker relates to containers. In the following, we aim to provide a short but clear overview of the idea behind containers and point out the correlation with the Docker project.

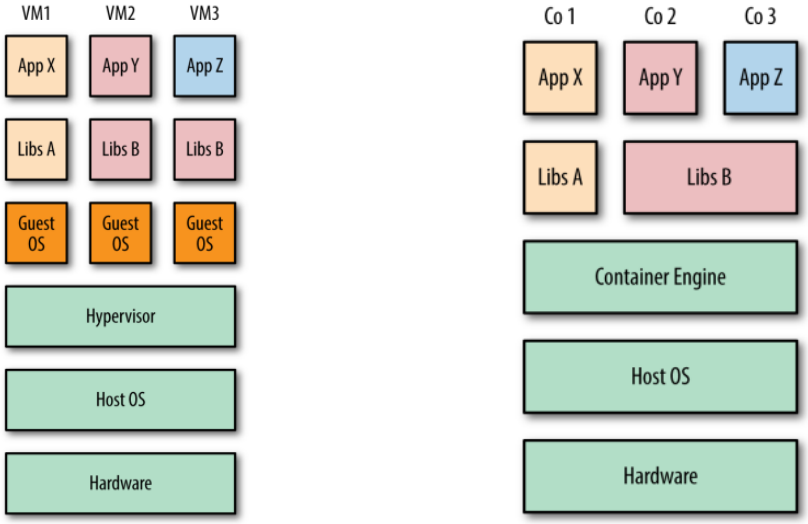
3.5.1 What is a container?

First of all, a distinction must be made between Docker and the concept of containers in general. A container is a virtualization technique based on features of the Linux kernel, whereas Docker is a tool which harnesses these features and wraps them with a user-friendly API. Containers differ from regular VMs by virtualizing at the OS level, running processes in their own isolated environments on top of the Linux kernel of a single host system. VMs, however, are managed by a hypervisor that operates at the hardware level and assigns hardware resources to them, and each VM uses them to bring up a full-fledged OS with its own kernel. While this sort of virtualization is based on resource sharing by creating well-defined slices of available hardware, containers enable OS sharing by means of Linux kernel features like *namespaces* and *control groups (cgroups)* [46].

The Linux kernel offers a wide range of different namespaces (process namespace, user namespace and network namespace, among others) which serve the purpose of isolating a container from the underlying host system. Thus, all processes running within a container only have a limited view of the host OS with their own process hierarchy, virtual network interfaces and user databases [46]. That allows a variety of containers to coexist on the same OS without even knowing they are actually sharing resources with each other. Another important kernel feature are cgroups, which allow for putting resource constraints upon a collection of one or more processes. By that means, a container's resource consumption concerning memory, CPU share and disk I/O bandwith can be limited and fair resource sharing between containers and other host OS processes can be achieved [46].

Compared to hardware-level virtualization (see figure 3.5a), resource utilization with containers is much more efficient (see figure 3.5b) because they avoid the overhead of booting multiple OS kernels which requires a significant amount of resource capacity.

Moreover, booting a complete OS from scratch is time-consuming, whereas starting and stopping processes can usually be done in fractions of seconds. Consequently, bringing up a container can be done orders of magnitudes faster than setting up a VM. Containers also solve the problem of what Merkel [46] calls "dependency hell".



(a) Hardware-level virtualization **(b)** OS-level virtualization

Figure 3.5: OS-level virtualization and hardware virtualization compared. [57]

Installing several applications on the same system might lead to conflicts between their dependencies, for example two applications demanding different versions of the same shared library. The side by side installation of possibly interfering versions of the same library threatens the stability of deployments and renders periodic updates almost impossible. By packing up each application along with its, and only its dependencies in a sandboxed process environment which is isolated from the surrounding host filesystem (e.g. via chroot [57]), these problems no longer exist [46].

3.5.2 What is Docker?

Docker is an open source project written in Go that, at its core, acts as a wrapper around the set of Linux kernel features described above and offers a convenient interface to launch containers from user space. To achieve this, Docker is split up into a command line client as well as a server component running as a daemon. While the Docker client offers a convenient interface for container management, the Docker daemon is in charge of the heavy lifting by interacting with the kernel. The fact that the Docker daemon executes with root permissions in order to be capable of accessing all required OS resources like devices is, however, often considered a major security risk [57].

The Docker daemon, which is sometimes also referred to as *Docker engine* [57], has undergone a major transformation in the past few years, moving from a monolithic structure towards a modular design comprising various standalone open source projects. One of these projects is the *runC* container runtime [42], which is build upon the *libcontainer* execution driver that has been implemented to talk to the kernel APIs directly. In its early stages, Docker did not interact with the kernel on its own but

instead integrated several drivers that delegated container administration to existing container technologies like Linux Containers (LXC) [57].

One of Docker's key features is its strong focus on reusability. In order to avoid repeatedly packaging of popular software components for usage within a container, Docker ships with the notion of *images*, which pose isolated filesystems that can be populated with arbitrary software. What is special about Docker images is that they are effectively read-only. Every change that is made to an image goes into a seperate *layer* that can again be committed to a new immutable image. Docker images usually start with an empty root or a minimal distribution-flavored filesystem which can be enriched with additional packages, configuration files and other modifications that get stored in their own layers. As a consequence, a Docker image can more precisely be defined as a read-only filesystem consisting of up to n layers, with $n \geq 1$. Docker launches containers from images, meaning that all n read-only image layers plus an additional writable *container layer* get mounted at a common mount point [14], which is called a *union mount* [57]. To generate the impression of a single consistent filesystem, the layers are stacked on top of each other in the order of their creation, so the initial root filesystem must be the bottom layer. The container layer, which is empty at the beginning, stores all changes so that they *shadow* the underlying image [14]. This is a widely spread technique known as copy-on-write (CoW) [57]. The changes kept in the writable layer be made permanent by committing them to a new image, from which in turn another container can be launched. If a container (i.e. the writable layer) gets deleted instead, the modifications it holds are also gone permanently [14].

Docker images are created from *Dockerfiles* [57], which are basically text files listing sets of build steps. Each build step includes a single instruction, like running a command or defining an environment variable, which results in a new layer being added to the final image. Mouat [57] gives a detailed explanation of how Dockerfiles work.

The layered design of Docker images has a strong impact on reusability and allows for efficient sharing not only on a per-image but even at a per-layer basis. Firstly, creating a public place where prepackaged images for commonly used software can be downloaded seems logical, which is exactly what Docker offers with its public image *registry* called *Docker Hub*. For scenarios where placing images on third party servers is not an option, self-hosting a private instance of the Docker Registry is another alternative. A registry contains n image *repositories* which usually store different versions of the same image. These versions can be differed by alphanumeric identifiers - or *tags* - that can be attached to images in the same repository and default to "latest". Consequently, Docker images adhere to the following nomenclature, which must be followed when pulling an image from or rather pushing it to a registry [57]:

`<registry-server>/<repository>:<tag>`

3.5.3 Mesos and containers

In section 3.2.2, we already mentioned that Mesos is capable of using different container technologies as *containerizers* [89] for tasks as well as executors. That is, frameworks define tasks that run an arbitrary executable like `/bin/echo` and Mesos takes care of launching the command within a container. If desired, it is also possible to instruct Mesos to put executors, which act as supervisor processes for tasks, in containers. That makes rolling out new custom executors and switching between existing ones much more convenient. Mesos is able to leverage third party container technologies like Docker to implement containerizers, but meanwhile also brings its own container tools, which are built upon the same set of kernel features like Docker or LXC, however with smaller set of features. From a Mesos perspective, integrating a custom solution for containerization is a reasonable goal, as this approach supersedes the necessity of external dependencies. If for example Docker is used as a container provider, it must be ensured that each host running a Mesos agent has the Docker engine installed. Apart from leaving containerization of tasks to Mesos, frameworks may, as an alternative, launch containers explicitly by themselves. For instance, Marathon supplies a JSON-based abstraction for the definition of tasks as Docker containers. Behind the surface, these tasks definitions are translated into plain shell commands using the Docker CLI which are then passed to Mesos and launched by the default command executor.

3.5.4 Docker, containers and how they help with building a hybrid microservice infrastructure

The favorable characteristics of containers and the Docker ecosystem can be exploited to tackle the demands on a reliable microservices infrastructure defined earlier. Firstly, employing containers instead of VMs for application isolation significantly lowers memory and disk space footprints and leaves more resources to application deployments. Although VMs are still needed for the OSes serving as the container hosts, much less of them are required to accomplish the same degree of logical cluster partitioning.

Besides, since we assume a heterogeneous amd64/s390x platform, there are many packages and software tools that are available only on a subset of the Linux distributions for special processor architectures like s390x. As Docker images may be based on various Linux flavors independent of the underlying host distribution, we can always choose the base image that best fits our needs. In this thesis, we even try to go one step further by putting not only arbitrary applications, but also the microservice infrastructure components itself, including Mesos and ZooKeeper, into containers. While this seems like radical approach, it reduces the complexity of running our infrastructure on different s390x Linux distributions to only having to provide appropriate Docker

packages for the respective Linux VMs. As for our infrastructure and possible dependencies, all we have to do is creating one image per component which is derived from a single, arbitrary s390x base image instead of tweaking the installation processes according to the particularities of several Linux distributions.

Chapter 4

Practical implementation of a microservice infrastructure

The last chapter discussed many technical details of the components that we have chosen to form our infrastructure for system management services. It has been shown that, in theory, Mesos comes with an inherent flexibility that makes it a suitable candidate even for very special environments like IBM System z. However, this flexible design comes also at cost: In contrast to integrated ready-for-use solutions like Kubernetes or Docker Swarm, setting up Mesos in practice is much more costly, as additional dependencies for coordination, scheduling, networking etc. must also be installed. Furthermore, regarding a hybrid target platform, it became clear after a while that moving Mesos to the IBM System z s390x architecture is a non-trivial task.

Thus, we chose a two-step approach to keep complexity manageable: First of all, a fully functioning Mesos setup shall be established on several amd64 machines in order to get familiarized with its characteristics as well as potential pitfalls. In a second step, the existing homogeneous infrastructure shall be extended to also integrate s390x-based VMs. This chapter covers the first part of this journey. Thereby, the focus will be on the most relevant impediments we run into when installing Docker, ZooKeeper, Mesos, Marathon and Aurora.

4.1 The interim goal

The homogeneous setup we decided to strive for as an intermediate milestone (see figure 4.1) comprised three VMs running CentOS 7.3 (Kernel version 3.10.x) [95] as their host OS. It has further been determined that a software stack including Docker, a ZooKeeper server, a Mesos master, a Mesos agent, a Marathon instance as well as an Aurora instance should be launched on each of these machines. From a resilience perspective, spreading the ZooKeeper, Mesos and framework components across different sets of hosts might be a reasonable approach for large-scale deployments in or-

der to create multiple independent failure domains. However, we decided to adhere to a smaller scale in order to simulate a scenario that resembles the situation on mainframes as far as possible. There, we can currently assume to have two SEs (primary and alternate) as well as a single LPAR at our disposal.

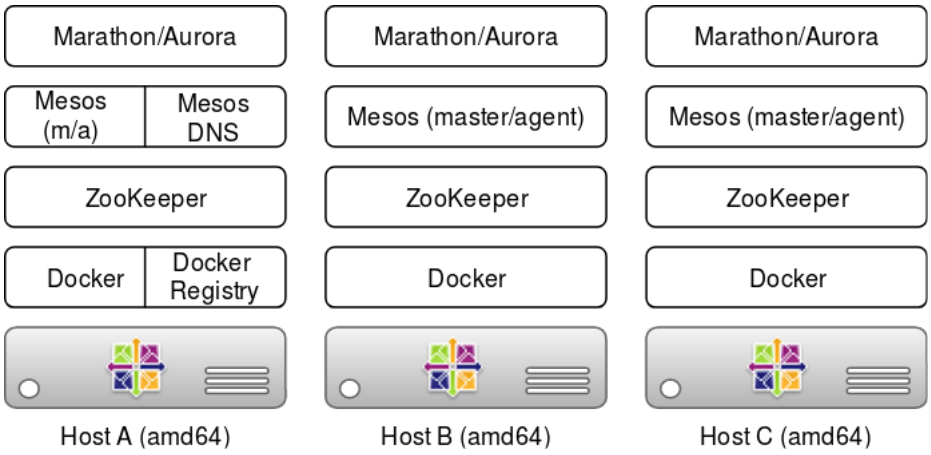


Figure 4.1: An ensemble of three amd64 hosts running a fully functional Mesos setup.

Besides, in order to have a private Docker image store available, one of the machines hosted a Docker registry, where all containerized infrastructure components as well as demo services could be saved and from where they could easily be pulled to the other servers. For service discovery, we placed a single Mesos-DNS instance on one of the hosts, which was used as the primary DNS server by all VMs.

Due to restrictions enforced by the underlying virtualization platform, our setup only had a minimal network configuration without DNS and with no direct access to external package repositories. Thus, we could only use an internal mirror server to install packages and resolve dependencies, so that software from third party repositories had to be installed by downloading the CentOS RPM packages to our local workstation and copying them to the VMs manually.

The following list gives an overview of the versions of the software tools that have been in use for the development of our amd64 prototype:

- **Docker CE:** 17.03.1 (released March 27, 2017)
- **Docker Registry:** 2.6.0 (released January 18, 2017)
- **ZooKeeper:** 3.4.10 (released March 30, 2017)
- **Mesos:** 1.1.1 (released February 25, 2017)
- **Mesos-DNS:** 0.6.0 (released September 15, 2016)
- **Marathon:** 1.4.2 (released March 27, 2017)

- **Apache Aurora:** 0.17.0 (released February 6, 2017)

4.2 Manual installation of Docker on CentOS

In March 2017, Docker Inc. announced that their core product, the Docker engine, would be available in two different flavors from now on: The Docker Community Edition (Docker CE) that comprises all free Docker products, and the fee-based Docker Enterprise Edition (Docker EE), which is recommended for business-critical deployments and comes with additional features like vulnerability scanning and monitoring [68]. For our prototype, we decided upon the free Docker CE version in order to avoid any license costs during development.

4.2.1 Bringing Docker to our CentOS servers

Basically, Docker officially provides the latest version of Docker CE for amd64 CentOS via their corresponding RPM repository [19]. As already mentioned, though, we could not use online repositories as we were unable to access them over the network. Luckily, Docker also publishes their RMPs on one of their websites[1], so we were able to download the Docker packages to our local workstation, from where we could transfer them to the CentOS hosts. At that time, the newest Docker version available was the 17.03.1 release (as of March 2017).

The dependency resolution process based on the internal mirror server worked well and the installation process finally finished without an error.

4.2.2 Fixing the Docker storage driver settings

Although the installation procedure itself worked on first try, we discovered that the Docker daemon was not running afterwards. Trying to launch the daemon by hand caused the following error message:

```
$ sudo systemctl restart docker.service
Error starting daemon: error initializing graphdriver: driver not supported
```

Listing 4.1: Docker daemon failure due to missing AUFS support in CentOS.

At first, this error message was not really helpful as there was no hint on what Docker actually means by a "graphdriver". Some research on this topic revealed that, as Estes [34] explains, the term *graphdriver* derives from the "image graph" formed

[1] https://download.docker.com/linux/centos/7/x86_64/stable/Packages/

by the layers of a Docker image, which represents the relationships between them. A *graphdriver* or *storage driver* follows these relationships when mounting image layers into a consistent root filesystem for a container. A storage driver in Docker can be defined as a plugin or backend the daemon integrates to interact with the kernel modules enabling different kinds of filesystems on a host OS [46, 34]. In section 3.5, it has been explained that each Docker container lives within its own dedicated filesystem where it can manage its software dependencies in isolation from other containers and host processes. In order to create these filesystems, Docker ships with several storage drivers that either add union filesystem and CoW capabilities on top of existing filesystem implementations that do not include these functionalities or, if available, use filesystem-native features for that [34].

Since distributions are built upon different, mostly modified versions of the Linux kernel, not every type of filesystem can be used on every distribution out of the box. A complete list of filesystems supported by a particular Linux flavor can be found under `/proc/filesystems` [60]. Supplementary, the Docker documentation [26] offers a list of available storage drivers which can be used without further ado depending on the Linux distribution in use.

The standard storage driver used by Docker, given that no other is explicitly configured, is *aufs*, which sits on top of the Advanced Multi-Layered Unification Filesystem (AUFS) [30]. According to the documentation [30], Docker prefers AUFS over alternative filesystems because it allows containers to start quickly and manages storage as well as memory efficiently [46]. Estes [34] describes efficient memory handling more precisely, saying that, for example, memory pages of shared libraries can be used by multiple containers simultaneously.

Unfortunately, AUFS is not part of the mainline kernel and can therefore not assumed to be present on every Linux distribution [30], which is in fact what causes the Docker daemon to fail in CentOS (see listing 4.1). Circumventing that issue can be done by either installing the necessary kernel modules by hand or switching the Docker daemon to another storage driver. We decided upon selecting an alternative driver, as enabling AUFS in CentOS requires not only to build the kernel module but also to patch the Linux kernel itself [62]. The second option is much simpler and less error-prone. However, the question remains which storage driver shall be chosen.

Alternative no. 1: The overlay/overlay2 storage driver

Since version 3.18, the Linux kernel integrates *OverlayFS*, a union mount filesystem [6] which differs from AUFS only in some details, as Petazzoni [63] explains. The corresponding Docker overlay driver should therefore be considered the preferred alternative for Linux systems based on kernel 3.18 or later. Concerning this matter, the problem with CentOS is that, as it is based on Red Hat Enterprise Linux (RHEL), even the

latest releases are built upon kernel 3.10.x [66]. Nevertheless, the RHEL 7.3 documentation states that this version comes with at least experimental OverlayFS support, particularly emphasizing that Docker interoperability is fully supported. Potential users are still warned against the occurrence of bugs and random errors, though [65].

An alternative option that bypasses this restriction is to manually upgrade to a more recent kernel version. In CentOS, which runs on a RHEL kernel, this can only be done by means of a community-maintained third party repository [96] that lacks any official vendor support by Red Hat. In case this variant is chosen, it is recommended to use the *overlay2* storage driver that first came with the Docker 1.12 release, as this rewrite of the legacy *overlay* driver includes several major bug fixes Estes [34] discusses in detail.

Alternative no. 2: The devicemapper storage driver

In situations where manually updating or patching the CentOS kernel is not eligible for some reason, working with the *devicemapper* storage driver is another option. This storage driver is built upon the kernel-based *Device Mapper* framework, which is able to directly interact with storage devices on the block level rather than using an abstraction in the form of a filesystem [32].

The most important advantage of the devicemapper storage driver is that the corresponding Device Mapper subsystem [7] is also available in slightly older kernels (since kernel 2.6). On the other hand, this storage driver might introduce serious performance issues when relying on the Docker standard configuration. By default, devicemapper runs in `loop-lvm` mode [32] and thus puts all container data and metadata on a loopback device backed by a sparse file. Such a sparse file comes with the favorable characteristic of only occupying disk space for blocks that have actually been written. The remaining blocks are assumed to be zero and are never allocated on disk. If a running container writes to a new block, the devicemapper storage driver dynamically allocates a fresh 64 KB block from a dedicated *thin pool* [63].

The problem with this approach is that it causes significant performance overhead as allocating blocks from the pool is a costly operation, especially on loopback devices. At worst, write operations are simply stalled given that there are no more blocks available in the pool. They can only continue if the pool gets increased [63].

While declines in performance may be acceptable for testing, they are definitely unacceptable for production environments. Petazzoni [63] therefore strongly recommends to place all Docker directories on dedicated, and above all, real devices. This enables devicemapper to run in `direct-lvm` mode, which is much more efficient due to faster block operations. Moreover, dedicated disks offer lots of capacity for the thin pool to be able to grow as needed. While having to configure a dedicated storage device for production systems is acceptable, this is certainly too much effort for test-

ing environments which should be able to use the devicemapper driver in `loop-lvm` mode without concerns [63, 26].

For our prototype, we went with the overlay2 driver which did not cause any issues throughout our work on this project. Nevertheless, although the RHEL documentation [65] claims the built-in OverlayFS interoperability to be mature, it might not yet be the best choice for production environments.

In order to switch the storage driver to *overlay2* on CentOS 7.3, the Docker daemon's `--storage-driver` option can be applied to override the default settings [63]. The Docker installation routine should already have created a systemd [97] unit file similar to the one shown in listing 4.2, by means of which the necessary configuration adjustments can conveniently passed to the daemon as soon as it gets launched.

```
[Unit]
Description=Docker Application Container Engine
# further unit information ...
[Service]
ExecStart=/usr/bin/dockerd --storage-driver=overlay2
# further configuration steps ...
```

Listing 4.2: How to change the storage driver to *overlay2* with systemd [63].

4.3 Hosting a private Docker registry

With a working Docker installation in place, our next step was to bring up a self-hosted Docker registry in order to store our infrastructure and demo application images and to conveniently distribute them across the host VMs. Docker must be installed beforehand, as the image registry is a Docker image by itself and also runs as a container. A Docker registry can basically be operated in two different modes, either as an *insecure* registry or with TLS enabled [16].

4.3.1 Running an insecure registry

An insecure registry can neither be trusted by a Docker daemon, nor can any (possibly malicious) Docker host in the same network be barred from downloading or uploading images. In order to be able to pull images from an unprotected HTTP registry, the Docker daemon residing on the requesting host must explicitly be instructed that it is trustworthy. Otherwise, the daemon refuses to establish a connection with the HTTP registry service as it enforces Hypertext Transfer Protocol Secure (HTTPS) by default and thus fails if no valid certificate is provided. To skip all verification steps and allow traffic over plain HTTP, the Docker daemon must be started with the `--insecure-registry` flag pointing to the registry host's IP address [28]. Although

we considered this configuration sufficient for the development phase, it should under no circumstance be applied in production.

4.3.2 Running a secured registry

For a registry server that shall also be accessible by other machines in the network and not only by localhost, the Docker documentation recommends to obtain a valid certificate from any trusted authority and configure the registry to only accept HTTPS connections. As an alternative, a self-signed certificate can be used to establish an acceptable level of trust for test environments within a closed network. In both cases, each registry client must be able to verify a certificate can actually be trusted [28]. This is usually not a problem for certificates issued by well-known certificate authorities, but self-signed certificates must explicitly be added to the set of the daemon host's trusted certificates. Besides, using TLS certificates also requires checking their expiration dates periodically and renewing them on time. Once a registry certificate has expired, the Docker daemon refuses to establish a connection to the registry server.

Once the registry serves images over HTTPS, client-side authentication can be enabled to ensure that pull and push operations require passing the correct credentials via basic authentication. Enabling this feature requires setting up HTTPS beforehand, as otherwise the credentials would be transferred in plain text [28].

4.4 A ZooKeeper ensemble in Docker

Setting up ZooKeeper on a group of three amd64-based VMs is a straightforward procedure that can be done by simply following the instructions in the Docker image repository[2] and shall therefore not be deepened at this point. What is much more interesting is a new feature we discovered in the upcoming version 3.5 of ZooKeeper, which is currently in alpha state. So far, a ZooKeeper quorum has always been statically defined, meaning that the servers making up the ZooKeeper group have been determined at startup time. Consequently, is was not possible to dynamically add additional servers to a running quorum in order to make it more resilient or to replace a faulty server. The same restriction applies to the removal of a ZooKeeper server from the ensemble. In such a case, the entire setup had to be restarted after the configuration had been changed.

With the arrival of ZooKeeper 3.5, such "rolling restarts" [93] are no longer necessary, due to the newly introduced concept of *dynamic configuration parameters* that, for example, allows the group membership to change at runtime. Shraer et al. [69] de-

[2] https://hub.docker.com/_/zookeeper/

scribe the details of how this behavior can safely be implemented without risking data corruption or inconsistencies.

However, we faced some difficulties during our tests with the 3.5 version that prevented the ZooKeeper instances on our three servers from successfully forming a quorum. Because it was not clear what might be responsible for that problem, we opened an issue[3] to discuss this behavior with the development community and, for the moment, decided to adhere to the stable 3.4 version for our prototype.

4.5 Deploying Mesos in Docker containers

The Mesos project by itself does not offer any official amd64 images for Mesos on Docker Hub. The most popular publicly available Mesos Docker images are maintained by Mesosphere [47], the company behind the Marathon framework. Our first evaluation of these images raised two substantial concerns:

1. The Mesosphere images are built upon the Ubuntu base image, which is very large in size and could be replaced with the much smaller Alpine Linux image[4].
2. Mesosphere uses custom albeit public DEB repositories[5] in order to install Mesos to their images.

Nevertheless, as both preparing a Docker image based on Alpine as well as building a custom DEB package would have introduced lots of additional work in terms of compiling, assessing the arguments lead to the decision of sticking with the Mesosphere images for our project. The necessary steps to setup the Mesos master and agent container are well-documented in the Mesosphere Github repository [47] and will not be covered in depth. Though, there are some aspects to pay attention to when operating Mesos in containers.

4.5.1 Mesos master containers

- **Usage of the host network:** In their Github *README* [47], Mesosphere indicates that containers running Mesos should be directly attached to the host network due to performance reasons and decrease of configuration complexity. Unless specified otherwise, Docker containers get their own virtual network interface assigned that is connected to the host network stack by means of the *docker0* bridge network [57]. This indirection can be bypassed by changing a container's

[3] https://issues.apache.org/jira/browse/ZOOKEEPER-2766
[4] https://hub.docker.com/_/alpine/
[5] http://repos.mesosphere.io/ubuntu/

network settings to *host* on startup. While this may actually make certain configuration steps easier as the bridge network must not be considered, that approach also revokes some of the isolation mechanisms that are the justifications for using containers in the first place.

- **Volumes for log and working directory:** It seems reasonable to use Docker volumes for a Mesos master's working and log directories, in order to allow its persistent data and log files to survive in case the container has to be restarted or exits unexpectedly [88].

4.5.2 Mesos agent containers

- **Usage of the host network:** See above.
- **Volumes for log and working directory:** Besides storing log files in a separate folder, each Mesos agent has a working directory where executor sandboxes and checkpointing state are located. Like for the master containers, Docker volumes should be used to persist log files and critical data on disk [88].
- **Grant Mesos agent access to `/cgroup` and `/sys`:** For a Mesos agent to be able to report free resources to the leading master, it must have access to the host's `/sys` virtual filesystem as well as the `/cgroup` directory. These directories must therefore be accessible from the Mesos agent container, so that it can use the hosts's kernel interfaces to monitor and allocate resources as needed [47]. Note that when using CentOS, mounting the `/sys` directory does not work and instead the `/sys/fs/cgroup` directory must be passed into the container.
- **Give Mesos agent access to Docker socket:** Launching tasks as Docker containers from within the Mesos agent containers requires them to have access to the host's Docker UNIX socket *docker.sock* residing in the `/var/run` directory. There may be no other option in this case, but mounting the Docker socket within a container should always be treated with caution, as the Docker daemon executes with root permissions on the host system. As a consequence, having access to the Docker socket is synonymous with actually having root privileges on the host.
- **Avoid running Mesos agent container in privileged mode:** Mouat [57] explains that Docker containers can be given extended privileges by launching them in privileged mode. While containers are normally launched with a limited set of capabilities by default, privileged containers are, inter alia, allowed to access the host's devices. While Mesosphere instructs users to set the `--privileged` flag [47], we could not find any problems with running the Mesos agent con-

tainer in unprivileged mode during our tests. So this should be avoided in order to not give the container any permissions it does not actually need.

4.5.3 Remarks to Mesos-DNS in Docker

As for Mesos-DNS, we will skip the basic installation steps just as we did for Mesos and ZooKeeper, as they are described in the official documentation [56] in detail. The only critical point that must be noted for Mesos-DNS is that it must be configured as the primary DNS server for the Mesos agent hosts by adding the respective nameserver entry at the beginning of their `/etc/resolv.conf` file [56]. We furthermore attached the Mesos-DNS container to the host network, so no further configuration steps were necessary.

At the time this thesis has been written, there was no Docker image for Mesos-DNS built and maintained by Mesosphere, so we decided upon creating our own Docker Hub repository[6]. The related Dockerfile can also be found in appendix A.1.

4.6 Marathon and Aurora in Docker

The Mesos frameworks were last microservice infrastructure components which had to be installed. In 3.3, it has been mentioned that there are currently two popular frameworks that might pose suitable candidates for our use case, namely Marathon and Apache Aurora. We have drawn a comparison between these two frameworks instead of focusing on merely one of them, because it might be a quite conceivable approach to run both of them side by side, depending on the exact application requirements.

4.6.1 Marathon

Mesosphere supplies an up to date Marathon image repository for amd64-based machines on Docker Hub[7] that receives periodic updates and comprises a comprehensive documentation. The documentation includes detailed instructions on how to start Marathon from the Docker image and how to connect it to an existing Mesos setup. In our case, following this guide worked on first try and the only adjustment to make was attaching the Marathon containers to the host network, as we just did with the other containers, in order to avoid ending up with a mixed network configuration.

[6] https://hub.docker.com/r/pkleindienst/mesos-dns-docker/
[7] https://hub.docker.com/r/mesosphere/marathon/

4.6.2 Aurora

In terms of Aurora, obtaining topical Docker images has been far more time-consuming, since there were no official repositories related to Aurora on Docker Hub. Admittedly, we did find several custom repositories but at least determined to build our own ones for our prototype as we had substantial doubts about these repositories being updated and maintained regularly. The most challenging task about this was that Aurora is made up of four essential components (scheduler, executor, observer and CLI) that must be deployed independently (apart from executor and observer), whereas Marathon is just a single artifact. Consequently, while Marathon can be operated in a single container, separate images had to be created for each of the Aurora components.

Aurora scheduler

Putting the Aurora scheduler, which is implemented in Java, along with its JavaScript based UI in a custom Docker image was straightforward and did not require any special considerations. Getting the scheduler to run properly, however, took us quite a long time. The main obstacle from our point of view was that the installation guide [87] lacks a complete sample configuration for the scheduler, so we had to experiment at lot with the command line options until we were successful. The Dockerfile as well as the accompanying shell script can be viewed in the appendices A.2 and A.3, and we also published them in a Github[8] and Docker Hub[9] repository.

Thermos executor and observer

The Thermos executor and observer components must be co-located and are consequently published as a single *DEB* package for the amd64 architecture. As a result, they also have to be deployed in a single container. Because the executor, in turn, must be available in the same filesystem where the Mesos agent lives, we derived a single Aurora *worker* Docker image, that contains all Aurora worker components (Thermos executor and observer), from the Mesos agent image maintained by Mesosphere. For our setup, we used this image instead of the Mesosphere image directly in order to launch our Mesos agents. These Mesos agent instances were unrestrictedly interoperable with the Marathon framework as the default executor it requires comes with the Mesos agent base image anyway. The observer configuration, that we moved into a short shell script (see appendix A.5), is well-documented [87] and did not cause any problems. Our Dockerfile, that we attached in appendix A.4, is also available on

[8] https://github.com/apophis90/aurora-docker
[9] https://hub.docker.com/r/pkleindienst/aurora-scheduler/

Github[10], the Docker image can be downloaded from Docker Hub[11].

Aurora CLI

Creating a Docker image for the Aurora client application has also been uncomplicated as the necessary efforts limited themselves to taking a base image with Python installed and adding the Aurora CLI DEB file (see appendix A.6). The CLI configuration is done by means of a *clusters.json* file that follows a clear and concise specification [86]. In addition to the Aurora CLI Dockerfile, we also provide a sample *clusters.json* (see appendix A.7) file for demonstration purposes. Again, all files can also be found on Github[10], while the Docker image repository can be inspected on Docker Hub[12].

4.7 Facilitating container management

After a while, it became clear to us that manually starting and stopping the containers that make up our core infrastructure (ZooKeeper, Mesos masters/agents and framework schedulers) across three VMs is very cumbersome. Of course, we could simply have condensed the necessary Docker commands in a shell script which, however, becomes increasingly unclear the more containers have to be managed. Besides, we would have to had covered many edge cases like stopping and restarting just a single container, so the shell scripting approach would probably have been complex and time-consuming.

As an alternative, we turned towards Docker Compose [24], a convenience tool for multi-container applications developed by Docker. Docker Compose employs a declarative approach that allows the definition of a set of containers within a single YAML file [24]. Appendix A.8 shows a slightly shortened version of the *docker-compose.yml* file we used for the purpose of managing our Mesos infrastructure containers. Note that the image names refer to our self-hosted Docker image registry that we assumed to be located on *Host A* (see figure 4.1). By means of Docker Compose, we were able to interact with the entirety of infrastructure containers or just a subset of them, so that even starting, stopping and updating only a single container could be achieved without custom shell scripts [24].

[10] https://github.com/apophis90/aurora-docker
[11] https://hub.docker.com/r/pkleindienst/aurora-worker/
[12] https://hub.docker.com/r/pkleindienst/aurora-cli/

Chapter 5

A hybrid cluster management setup

The cluster management setup from the previous chapter is relatively straightforward with respect to the target environment, which has been entirely amd64-based so far. To evolve it towards a serious option for platforms like mainframes, it must be demonstrated that our current software stack can be expanded to heterogeneous infrastructures as well. Throughout this thesis, IBM System z, which incorporates amd64 servers and s390x processor modules, serves as the reference target environment for a hybrid microservice infrastructure. Trying to bring open source software to different kinds of machines is usually a nontrivial challenge, as in the majority of cases only the most relevant OSes (Mac, Windows* and Linux) and CPU architectures (amd64 and sometimes ARM) are officially supported by the project maintainers. Consequently, it follows that using such software on platforms like mainframes, which are less common, requires compiling the code and building suitable packages from scratch, causing substantial efforts that mainly on factors like the programming language in use.

5.1 A hybrid prototype as the next step

Starting from the amd64-based prototype that has been presented in the last chapter, our next objective was expanding our infrastructure to an s390x LPAR and integrate it into the existing setup. For that purpose, we established a virtualized, RHEL-based host OS running on a real IBM System z mainframe. The resulting simulation (see figure 5.1) was very close to the actual mainframe scenario. Although this is currently is not true on IBM System z, we had a fully functional Ethernet connection between the VMs acting as the SEs on the one side and the host system residing in a LPAR on the other.

* Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

The first question that had to be answered was if there were already installation-ready s390x packages available for the components of our infrastructure stack. Unfortunately, the research we did right at the beginning of the hybrid prototype development phase revealed that this was not the case. As a consequence, time and efforts had to be spend to do the necessary porting and packaging manually.

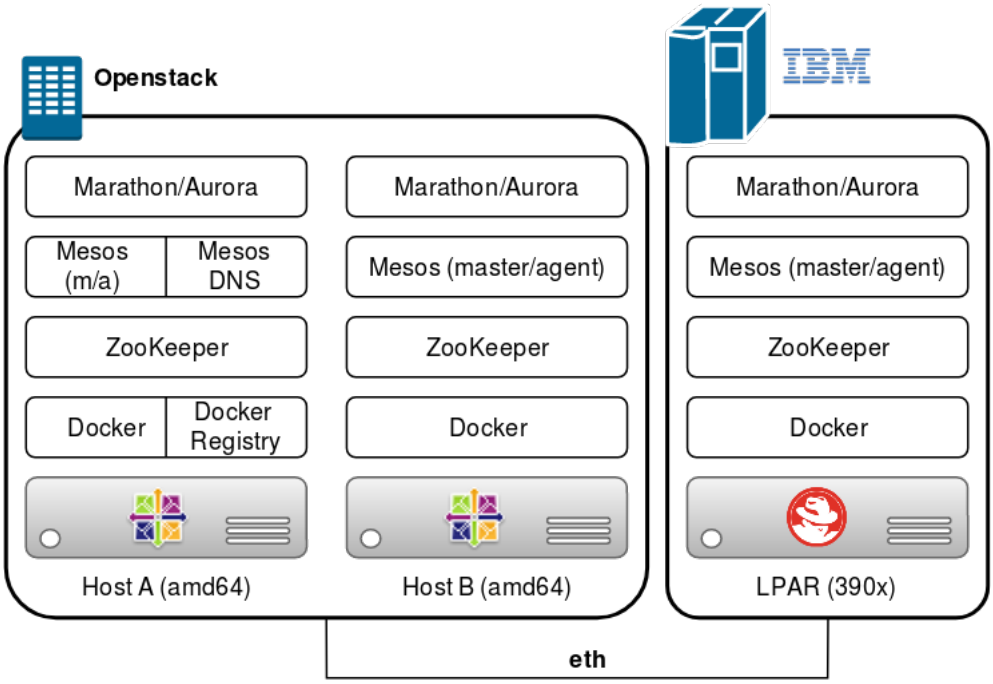


Figure 5.1: The next stage in the evolution of our hybrid microservice infrastructure prototype.

The following sections will therefore step through our basic software stack, comprising Docker, ZooKeeper, Mesos as well as the Mesos frameworks Marathon and Aurora, and explain the experiences we made but also the problems we had while trying to bring its pieces to s390x. We have to admit we did not have success with all components and projects so far, for instance we were not able to make Aurora available on the mainframe by the time this thesis was written. As a consequence, the draft shown in figure 5.1 running both frameworks next to each other in a multi-architecture fashion is still much more a vision than the current state of our work.

5.2 Status quo of Docker on s390x Linux

As our microservice platform has been built upon Docker containers so far, getting the Docker engine up and running on s390x machines was the first problem that had to be tackled. For a long time, there has been no official support for Docker on System z by Docker Inc. themselves, making it hard to setup fairly up to date Docker installations on System z mainframes. This situation somewhat changed over the last couple of

months and it could be seen that Docker is going to target a steadily growing number of platforms, for example IBM System z (s390x) [68]. This section sums up the state of Docker on the IBM mainframe at that time this thesis has been written, looking beyond our RHEL use case and thus also taking a look at other Linux distributions that are relevant for System z, namely Ubuntu and SUSE Linux Enterprise Server (SLES).

5.2.1 Docker CE on Ubuntu

Although there are, for quite a long time, s390x-based Docker binaries available for Ubuntu which can be installed through the *docker.io* package residing in the *universe* archive [10], this approach comes with two major drawbacks. On the one hand, software that is distributed via the universe archive is entirely under the community's control rather than being supervised by Canonical and the Ubuntu core development team [58]. Moreover, the binaries in the docker.io package were slightly outdated compared to the official releases for amd64. With the arrival of Docker v17.06 on June 28, 2017 Docker Inc. released their first Docker CE binaries for s390x Ubuntu systems [3]. In the case of Ubuntu on System z, Docker can thus be installed regularly via the official Docker Inc. repository for a few weeks now. The actual installation procedure will not be covered here since the necessary stages are described in the Docker documentation [20] extensively. Note that, as far as s390x is concerned, this guide only holds for Ubuntu version 16.04 LTS ("Xenial Xerus") and later.

5.2.2 Docker CE on RHEL

The situation on RHEL is slightly different compared to Ubuntu because until recently, there has been no Docker support for RHEL at all. By the end of June 2017 [2], Docker Inc. started providing statically compiled Docker CE binaries[1] even for s390x, which can principally be installed on any s390x Linux system regardless of the concrete distribution, as all dependencies in their correct versions ship with the executable. The positive aspect here is that the basic installation procedure only requires downloading the collection of binaries (there are single ones for the Docker daemon, CLI etc.) and placing them anywhere on the local system instead of having to deal with package managers and the setup of a third party repository [2].

However, this approach comes at the cost of additional configuration efforts, as the manual installation does not involve integrating Docker with other parts of the OS, for example the init system. While this is not mandatory, it enables more comfortable life cycle management of the Docker daemon and also automatic restarts after reboot. Furthermore, because the distribution's native package management system is bypassed,

[1] https://download.docker.com/linux/static/stable/s390x

upgrading Docker to a new version that includes new features or fixes critical security vulnerabilities needs manual intervention. Doing such things by hand is error-prone and may even be forgotten over time. If, in extreme cases, a large amount of machines must be managed and kept in sync simultaneously, reliable and regular maintenance procedures are almost impossible without using powerful automation tooling.

Just a few weeks ago, another option for Docker on RHEL showed up. With the arrival of the Docker v17.06.1 release, Docker Inc. announced professional support for their Docker EE edition not only on RHEL, but also on SLES and Ubuntu [68]. Although this is a major improvement not only for Docker but also for container technology on mainframes in general, these announcements are restricted to their fee-based flavor of Docker for now. For this thesis, however, we determined to stick to the free version only, which is why our focus is on Docker CE exclusively. Because we were restricted to RHEL for our s390x VM (see figure 5.1), we went with the static binary installation variant.

Just as for the installation of Docker on amd64 CentOS, we had to fix the storage driver settings in exactly the same way as we did in section 4.2.2, since RHEL for s390x likewise does not come with the AUFS kernel module for the aufs storage driver that Docker uses by default.

5.2.3 Docker CE on SLES

Just as for RHEL, there are still no Docker CE packages for s390x SLES provided by Docker Inc. Though, SLES offers the possibility of installing the free version of Docker via its *Containers Module* [70]. This comes with two important advantages compared to the static binary. First, the packages contained in the SUSE modules benefit from reliable vendor support and periodic updates. Moreover, the integration with other parts of the system, like the systemd init system, is done automatically during the installation process. This protects the user from having to setup lots of configuration details manually, as it has been necessary with RHEL in our case.

Since all SUSE modules are optional, they have to be enabled explicitly before packages can be downloaded and installed from them [71]. Afterwards, Docker CE can be installed by means the on-board package management utilities, as described by SUSE [70].

Some remarks on Docker storage drivers in SLES

SLES has a special characteristic that differentiates it from other distributions like Ubuntu or RHEL. Regarding version 12, it utilizes *Btrfs* as its standard filesystem [70]. Btrfs is a copy-on-write filesystem that is also included in the mainline Linux kernel. Its advantages include block-level operations and copy-on-write snapshots [31].

Docker implements the *btrfs* storage driver which can leverage many of these features. For Docker packages with a special focus on SLES, like the latest Docker EE for SLES 12 or the Containers Module releases, the btrfs storage driver is enabled by default. In case Docker CE has been setup by means of the raw binaries, the Docker daemon should crash because, just like RHEL, SLES does not ship with AUFS filesystem support included. As a workaround, the storage driver must be switched to btrfs by hand, at it has been demonstrated in section 4.2.2. If AUFS along with the aufs driver is obligatory for some reason the necessary kernel module might can still be added manually [70].

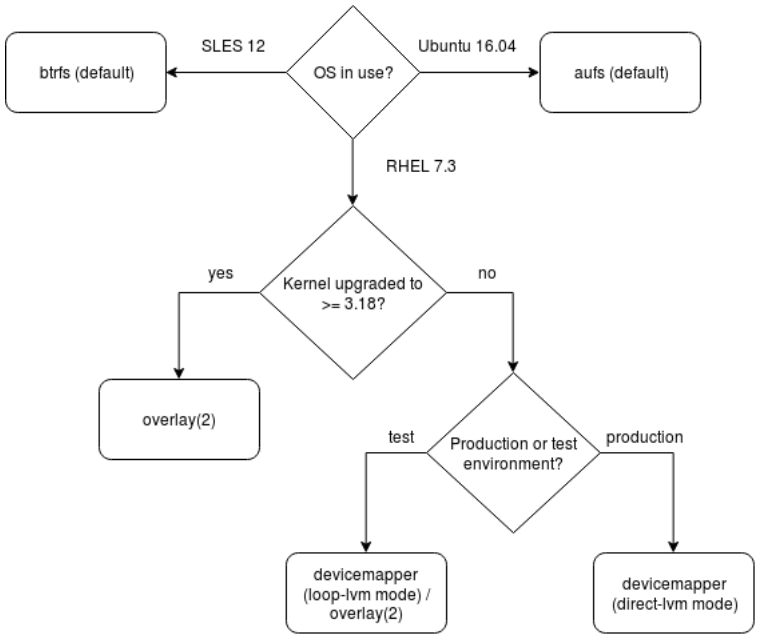


Figure 5.2: Storage driver recommendations for Ubuntu, SLES and RHEL.

Figure 5.2 sums up our recommended Docker storage driver settings for RHEL and besides shows the default settings for Ubuntu and SLES. For our hybrid prototype, which already used the overlay2 driver in its amd64-only stages, we adhered to the same configuration for the Docker daemon located on our s390x RHEL system for simplicity.

5.2.4 Summing up: Docker on System z

The subsequent tables underline how the situation of Docker availability on IBM System z has changed during our work on this thesis. At the beginning (see table 5.1), there basically was almost no reliable Docker support for s390x. SLES was the only distribution that offered reasonably up to date Docker releases via their Containers Module. As for Ubuntu, only a slightly outdated Docker package could be installed

by means of the community-driven *docker.io* repository, while RHEL was entirely lost out and there was absolutely no option to install Docker rather than compiling it on one's own [3, 2, 4, 10]. Docker EE was not available on System z at all.

Linux distribution	Docker CE	Docker EE
Ubuntu 16.04	(✓) (1.13.1)[1]	**X**
SLES 12	(✓) (<= v17.03.x-ce)[2]	**X**
RHEL 7.3	**X**	**X**

[1] Available via the *docker.io* repository maintained by the Ubuntu community [10], no support by Docker Inc. or Canonical.

[2] Docker v17.03-ce available via the SLES 12 Containers Module. This is just a rough estimation based on Bacher [4], as SUSE themselves seems not to publish any details related ot package versions. The Containers Module, however, does also not benefit from direct support by Docker Inc.

Table 5.1: Available Docker packages and versions for s390x at the beginning of this thesis.

When Docker v17.06 was first published on June 28, 2017, Docker CE support for s390x finally came to Ubuntu. For the remaining Linux distributions, statically compiled binaries have been shared from then on which, however, still lacked distribution-specific vendor support and caused additional configuration and maintenance overhead [3, 2]. In terms of SLES, the Containers Module maintained by SUSE has been an option further on and stepped forward to Docker CE v17.04 by the end of July 2017 [4].

With the arrival of the most recent v17.06.1 release on August 16th, 2017, Docker EE has been made generally available for all major Linux distributions (see table 5.2), whereas the situation for Docker CE remained unchanged [68].

Linux distribution	Docker CE	Docker EE
Ubuntu 16.04	✓(v17.06.1-ce)[1]	✓(v17.06.1-ee)[2]
SLES 12	(✓) (v17.06.1-ce)[3]	✓(v17.06.1-ee)[2]
RHEL 7.3	(✓) (v17.06.1-ce)[3]	✓(v17.06.1-ee)[2]

[1] Already supported since Docker v17.06.0 (June 28, 2017) [3].

[2] Docker EE officially available for Ubuntu, SLES and RHEL on s390x since August 16th 2017 [68].

[3] Still no official Docker CE packages for RHEL and SLES, but statically compiled binaries available. For SLES, the Containers Module is still maintained, the most recent Docker version it includes is v17.04.0-ce [4] (as at August 2017).

Table 5.2: Available Docker packages and versions for s390x since the v17.06.1 release.

5.3 Docker Compose on IBM System z

Because we already utilized Docker Compose for more comfortable container management regarding our pure amd64 prototype, we considered having this tool also available on the s390x VM very useful. In fact, we found an up to date package repository on *PyPI*[2], which offered the source code as well as a prepackaged universal wheel [40]. Since such wheels do not depend on any native code, they can be installed on any system that has a Python interpreter available, regardless of the underlying system's CPU architecture. Thus, the latest Docker Compose for s390x could simply be installed by means of the *pip* package management tool, without the need for additional efforts.

5.4 The porting of cross-platform components to s390x

As for the Docker engine and Docker Compose, we were able to install the respective packages on our s390x host without having had to fix any source code or build scripts and the need for compiling anything from source. While Docker itself was available in the form of precompiled s390x executables, Docker Compose gained from the fact that it is implemented in pure Python, which means we could run and install it seamlessly on every platform that provided a Python installation. The cross-platform property of some programming languages is a substantial aspect when porting software projects between different types of machines. If an application is written in languages like C

[2] https://pypi.python.org/pypi/docker-compose/1.15.0

or C++ and directly compiles to native (i.e. processor-specific) machine code, the original source code has to be compiled and packaged from scratch for every new target platform that shall be supported. This problem will be examined in depth in the next section, which covers the transferring of Apache Mesos to the s390x architecture. First, we want to pay attention to the platform-independent parts of our cluster management stack, and we will show how to take advantage of this property if a software project should be moved to a new type of machines. The infrastructure components in question and the programming languages they are written in are listed in table 5.3.

It is notable that, with the exception of the Aurora executor and CLI, all the other infrastructure components are fully implemented in Java or its derivatives, like Scala in the case of Marathon. What Java and Scala have in common is they both compile into *.class* files containing platform-independent intermediate code, or byte code. This byte code can be executed wherever a Java Virtual Machine (JVM) is present, regardless of the underlying machine's architectural details [1].

The usage of portable programming languages brings up the benefit of only minor adjustments that have to be made in order to make some of the components, which have already been used for the homogeneous setup in the previous chapter, usable on a hybrid landscape. Hereafter, we will focus on the tools listed in table 5.3 and the necessary adaptations required to build s390x-compliant Docker images which are compatible to IBM System z.

Infrastructure component	Programming Language
Apache ZooKeeper	Java
Marathon	Scala
Aurora scheduler	Java
Aurora executor	Python
Aurora CLI	Python

Table 5.3: Cross-platform infrastructure components and their code base languages.

5.4.1 Apache ZooKeeper on s390x

For the porting of ZooKeeper, which is written in plain Java, the Dockerfile[3] used for the amd64 Docker image in section 4.4, can remain nearly unchanged. The only thing that must be fixed is the base image, since the standard OpenJDK Docker images target

[3] https://github.com/31z4/zookeeper-docker/blob/master/3.4.10/Dockerfile

amd64 systems exclusively. However, Docker Hub also includes a constantly growing number of s390x images in a dedicated repository, which are officially supported by *Docker Inc.* and also maintained by the Docker community. One of these images that perfectly fits as a base image for ZooKeeper on System z is the *s390x/openjdk:8-jre-alpine*[4] image. Swapping the base image used in a Dockerfile is as simple as changing the corresponding image name after the `FROM` instruction, as listing 5.1 shows.

```
# Derive from the official s390x OpenJDK image
FROM s390x/openjdk:8-jre-alpine
# Rest of the file is identical to the amd64 version ...
.
.
```

Listing 5.1: Dockerfile for ZooKeeper on s390x.

As before, the Alpine-flavored base image is used due to its leanness (55MB) compared to base images like Ubuntu or Debian.

5.4.2 Marathon on s390x

In contrast to ZooKeeper, the Dockerfile for the Marathon s390x image had to be rewritten from scratch. The original amd64 Docker image maintained by Mesosphere Inc. (accessible via the *mesosphere/marathon* repository[5] on Docker Hub), which is built upon a Debian base image, first installs the Mesos libraries from their own DEB repositories and builds Marathon from source afterwards. Compiling Marathon from source was not a requirement for us, though, because Mesosphere Inc. already provides a precompiled version that is available for download as a *.tgz* archive.

Since we could not rely on the amd64 version of the *libmesos.so* shared library, we had to use a custom base image for the Marathon s390x Docker image instead, which includes the corresponding binaries compiled for s390x. Marathon requires *libmesos.so* to be present at runtime in order to be able to communicate with the leading Mesos master via Java Native Interface (JNI) [53]. From our perspective, this design is much rigid and inflexible, but we expect it to be gone in near future as soon as Marathon will shift towards using the Mesos RESTful API. Consequently, we have derived the Marathon image from our custom Mesos image (see section 5.5) that, apart from the required shared libraries, already includes a JRE for executing the precompiled cross-platform Marathon byte code. The complete version of the Marathon Dockerfile for s390x, which is quite extensive, can be found in appendix B.1.

[4] https://hub.docker.com/r/s390x/openjdk/tags/
[5] https://hub.docker.com/r/mesosphere/marathon/

5.4.3 Apache Aurora on s390x

Previously, we noted there are no officially supported amd64 Docker images for Aurora on Docker Hub and thus created our own ones. Likewise, custom Aurora images had to be built for the s390x architecture. At the beginning, the efforts of porting Aurora were expected to be limited to only having to switch the base images in our amd64 Dockerfiles as we did for ZooKeeper due to Aurora's platform-independent code base. In fact, it turned out to be much more complicated, because the Aurora community only provides DEB packages targeting amd64 platforms. This limitation by itself is not an exclusion criterion, as it should always be possible to create a suitable distribution package given that the source code is available.

In that sense, we started trying to compile the Aurora executor for s390x, without having been successful so far. Our build of the Aurora client application also failed with an error message (see listing 5.2) which is related to the *Pants* build tool [101]. Further investigations in the Pants source code indicated that on Linux systems the output of the `uname` command is used to detect the underlying CPU architecture and then load an appropriate binary from an *Amazon S3* bucket[6] for assembling the binary. Examining the bucket's contents revealed that there are currently no binaries for s390x, which explains why the Pants build process exited with an error.

```
$ ./pants binary src/main/python/apache/aurora/client:aurora
.
.
FAILURE: Update --binaries-path-by-id to find binaries for (u'linux',
's390x')
```

Listing 5.2: Compilation of Aurora components written in Python fails for s390x.

Fixing the Pants build tool at this point would have caused substantial effort in order to get the build process working for s390x. Moreover, because prepackaged DEB artifacts for the Java-based Aurora scheduler component are only available for amd64, we would also have had to create a s390x version of it. Consequently, all efforts to port Aurora have been stopped at this point, as we considered this work beyond the scope of this thesis and decided upon tackling this in the future. Subsequently, Marathon currently is the only Mesos framework we can operate in a hybrid manner without any restrictions.

[6] https://s3.amazonaws.com/binaries.pantsbuild.org

5.5 Compiling Apache Mesos for IBM System z

The previous section has stated that preparing software projects which are written in portable programming languages for new target platforms can be done with minimal endeavors in most cases. Using Python and Java as an example, the intermediate byte code produced by the compiler can be executed wherever an appropriate interpreter is available.

With programming languages like C or C++ the situation is slightly different, as they directly compile into platform native machine code that cannot simply be run by varying types of CPUs. Instead, these languages require explicit compiling of the source code for each new target platform, which can be a difficult and tedious process. For code that additionally relies on specific OS-native features that are only available in very modern versions of the Linux kernel, things get even more complicated in contexts like IBM System z, where customized kernels are regularly in use.

Regarding Mesos, which comprises about 10,000 lines of C++ code [39], the initial approach was to search for already existing s390x packages in order to avoid the efforts of having to compile at package it up from scratch. However, our investigations did not yield any satisfying results as we only found some outdated RPM repositories[7], which did not make it beyond the Mesos 0.22.1 release. By the time this thesis was written, the latest Mesos release was 1.3.1 [78]. So the only way to obtain an up to date s390x Mesos distribution was building it from source at this point.

The following sections will demonstrate the steps we went through while having built Mesos from source, with a special focus on the pitfalls we had to cope with. Furthermore, we will explain how to end up with a Mesos Docker image based on the Ubuntu base image, that allows to launch Mesos master and slave nodes on s390x systems as Docker containers. A basic understanding of the *GNU build system* as wee as the *GNU Autotools*, which form the basis for working with the Mesos souce code, will be required. Moreover, all build steps are assumed to be executed on an Ubuntu s390x system running on top of a 4.4.0 Linux kernel.

5.5.1 Preparing the compile process

The Apache Mesos documentation [82] provides useful information on the steps that have to be followed in order to successfully build Mesos from source. It all starts with the installation of several dependencies, which is straightforward and will be skipped for clarity. A list of those packages can also be found in the documentation [82].

Adhering to the compile instructions, the first step is running the `bootstrap`

[7] https://www.rpmfind.net/linux/rpm2html/search.php?query=mesos

shell script residing in the project's root folder after having cloned the repository[8]. In essence, what the script does is running the `autoreconf` utility which is part of *Autoconf* [9]. Its main purpose for the Mesos project is running multiple tools in the right order and finally creating a `configure` script out of the *configure.ac* template file. This happens through the invocations of the following Autotools [9]:

1. **libtoolize:** Prepares the Mesos project to use *Libtool* by generating a custom `libtool` script. Libtool eases dealing with shared libraries across different Unix platforms.
2. **autoconf:** Processes the contents of the *configure.ac* file in order to generate the `configure` script.
3. **automake:** Takes the *Makefile.am* file from the top level directory and creates a standard makefile template named *Makefile.in*.

Up to this point, we did not find any problems with our s390x build environment and the `bootstrap` script exited successfully.

5.5.2 Repairing the configuration phase

For the next stage, the Mesos documentation [82] states that the previously generated `configure` script shall be launched. Beforehand, we created a dedicated directory and set is as the installation path for the final *Makefile* (see listing 5.3). In this way, we accomplished ending up with a single folder containing a full Mesos installation which could then be used as the working directory for the DEB package build.

```
$ mkdir build/
$ cd build/ && ../configure --prefix=/tmp/mesos-deb
.
.
configure: error: failed to determine linker flags for using Java
(bad JAVA_HOME or missing support for your architecture?)
```

Listing 5.3: The `configure` step fails due to IBM JVM incompatibility.

However, as listing 5.3 shows, the configure phase failed at first, confronting us with the first serious issue related to the compile process. Inspecting the *configure.ac*[9] file which served as a template for the `configure` script, we found that there was an attempt to compile and link against some shared libraries for making use of JNI in

[8] git://git.apache.org/mesos.git
[9] https://github.com/apache/mesos/blob/master/configure.ac

order to invoke native C/C++ code from Java and vice versa [1]. The relevant excerpt of the *configure.ac* file (see listing 5.4) that made the process fail sets out that s390/s390x is indeed considered as a possible target platform, but also reveals that the path to the folder containing the *libjvm.so* library is strictly hard coded.

```
elif test "$OS_NAME" = "linux"; then
  for arch in amd64 i386 arm aarch64 ppc64 ppc64le s390 s390x; do
    dir="$JAVA_HOME/jre/lib/$arch/server"
    if test -e "$dir"; then
      # Note that these are libtool specific flags.
      JAVA_TEST_LDFLAGS="-L$dir -R$dir -Wl,-ljvm"
      JAVA_JVM_LIBRARY=$dir/libjvm.so
      break;
    fi
  done
```

Listing 5.4: Relevant excerpt of the *configure.ac*[4] file

Normally, given that `$JAVA_HOME` points to an Oracle Java Development Kit (JDK) or OpenJDK, these assumptions related to the location of the *libjvm.so* shared library are perfectly fine. If, however, an IBM Java Software Development Kit (SDK) is in use they do no longer hold because the shared library paths are slightly different, as listing 5.5 proves.

```
$ echo $JAVA_HOME
/usr/lib/jvm/java-ibm-s390x-80
$ find $JAVA_HOME/jre/lib -name "libjvm.so"
/usr/lib/jvm/java-ibm-s390x-80/jre/lib/s390x/default/libjvm.so
```

Listing 5.5: The location of the *libjvm.so* shared library for the IBM JVM.

Further research has shown that this issue had already been reported by other users of the IBM SDK in the Mesos project's issue tracking system[10] in January 2015. The fact that the issue is still unresolved appears to suggest that the IBM Java SDK's peculiarities will not be taken into account by the development team. This is comprehensible from our perspective, as favoring the interests of the community over company-specific demands is an important criterion for Apache projects.

It is a legitimate question if building the Java code of the Mesos project is really necessary at all. We asked us the same question and thus started examining the code, trying to get an understanding of why linking against the *libjvm.so* library is required

[10] https://issues.apache.org/jira/browse/MESOS-2216

in the first place. As fas as we understood, the Mesos compile process includes building the Java client API, which relies on JNI for communicating with the Mesos master, by default. Because we considered the Java client API not absolutely necessary for our use case, we could have easily skipped the Java build part by setting the `--disable-java` flag specified by the *configure.ac* file[9]. For fixing the shared library path issue instead, we finally came up with three different ways how this could be achieved in conjunction with the IBM Java SDK:

1. Setup another JDK flavor next to the IBM Java SDK (e.g. OpenJDK) and make the `$JAVA_HOME` environment variable point to the corresponding installation folder.
2. Move the build process into a Docker container that includes OpenJDK.
3. Adjust the shared library path in the *configure.ac* file manually.
4. Populate the shared library path environment variables beforehand, so that they already refer to the correct folder when `configure` is started [45].

In our opinion, the last option poses the fastest and easiest solution for making the build process compliant with the IBM Java SDK. All that has to be done, as proposed by the Linux on z Systems Open Source team [45], is exporting the environment variables containing the correct linker flags and shared library path (see listing 5.6. Even though that approach worked as expected it turned out to be not our first choice here, as it breaks the build's repeatability. Assuming that the build is triggered in a different shell with the necessary environment variables not set it will fail as it did before. One possible way out of this might be forking the repository and moving the environment variable export statements to a separate file in the repository, so that the shell environment can be prepared right before `configure` gets executed.

```
$ export JVM_DIR=$JAVA_HOME/jre/lib/s390x/default
$ export JAVA_TEST_LDFLAGS="-L$JVM_DIR -R$JVM_DIR -Wl,-ljvm -ldl"
$ export JAVA_JVM_LIBRARY=$JAVA_HOME/jre/lib/s390x/default/libjvm.so
```

Listing 5.6: Environment variables for linker flags and shared library path [45].

For this thesis, we went with the third option in order to achieve a consistent and repeatable build. What made fixing the *libjvm.so* path in the *configure.ac* file easier for us was that some contributors of the related JIRA issue mentioned above already submitted a patch file (see appendix C.1) that makes the `configure` script setting up the correct paths in case an IBM Java SDK is found under `$JAVA_HOME`. We applied the patch file to our copy of the Mesos repository in order to make the necessary changes

permanent and came to the conclusion that this is a more intuitive solution than compelling other users to fix the shell environment.

```
$ wget -O ibm-jdk.patch https://issues.apache.org/jira/secure/attachment/\
> 12746484/MESOS-2216_2.patch
$ patch < ibm-jdk.patch
patching file configure.ac
```

Listing 5.7: Applying the necessary patch for IBM Java SDK compliance.

After having applied the patch as demonstrated in listing 5.7 the `configure` phase finally succeeded. Note that `bootstrap` must again be executed before, so that an updated version of the `configure` script is produced from the patched *configure.ac* file.

5.5.3 Maven build issues

As the final build step, we had to issue a `make install` according to the documentation [43] in order to compile the code and install Mesos to the directory (`/tmp/mesos-deb`) we determined as the target folder earlier. However, the Java client API build step again turned out as a major obstacle, since the related Maven build repeatedly run out of memory while executing the Javadoc plugin. Going to the *pom.xml.in* file located in `mesos/src/java/` and removing the Javadoc plugin section would have been most simple solution, albeit not a very elegant one. Rather than skipping Javadoc creation, we finally fixed that issue by increasing the default heap memory limit for the Maven build instead (see listing 5.8) [45].

```
@@ -131,6 +131,7 @@
    <plugin>
      <artifactId>maven-javadoc-plugin</artifactId>
      <configuration>
+       <maxmemory>512m</maxmemory>
```

Listing 5.8: Increasing the heap space memory limit for the Maven Javadoc plugin [45].

After that `make install` finished without an error and Mesos was installed to the specified target folder. In order to be able to properly install Mesos within a Ubuntu-based s390x Docker image we decided upon packaging up the compilation output as a reusable DEB artifact.

5.5.4 Creating a s390x DEB package for Mesos

Packaging up Mesos as an installable artifact for Ubuntu/Debian is not absolutely necessary since we could simply have copied the installation directory into a Docker

image. Though, creating a package in a standardized format improves reusability and is a much cleaner way of distributing and installing software in our opinion. Moreover, DEB files allow for explicitly declaring runtime dependencies, so potential users can inspect them and install required packages by means of a package manager, for example Advanced Package Tool (APT).

As a convention, building a DEB package requires a `DEBIAN/` directory in the same folder where the software that is about to be packaged up is located. The `DEBIAN/` directory must in turn include a `control` file that provides necessary information for creating the package. Listing 5.9 gives an impression of how the `control` file looks like [58]. A complete version of this file can be found in appendix C.2.

```
$ cat > /tmp/mesos-deb/mesos-1.1.1/DEBIAN/control << EOF
Package: mesos-1.3.0
Architecture: s390x
.
.
EOF
```

Listing 5.9: Basic `control` file structure for building a s390x DEB package.

With the `control` file in place, building the Mesos DEB package can be done by means of the `dpkg-deb` utility [58]. As this is again straightforward, we will skip the details.

5.5.5 Installing Mesos to an Ubuntu-based Docker image

At least, we created the s390x Docker image for Mesos by means of a Dockerfile that installs the DEB package we had built before. Because the only difference between the Mesos master and slave images is the Mesos executable which must be invoked, we decided upon building a universal Mesos base image (see apendix B.2) from which we derived the master (appendix B.3) and agent (appendix B.4) images that merely specify different entry points for containers.

5.6 Remaining s390x porting

At the time this thesis has been written, not all software components that we cover here as part of our microservice infrastructure stack have already been prepared to run on s390x systems. The primary focus was on the most fundamental tools (ZooKeeper, Mesos, Marathon and Aurora) and we deferred some other parts that can still be ported in the future once we have a working prototype with basic functionality in place. For that reason, there are still components, like the Docker Registry or Mesos-DNS, that we can only run on amd64 machines at the moment. This is, however, not a

problem with respect to our hybrid prototype, since these parts do not absolutely need to be available on both machine types for a basic setup. Instead, making progress with Aurora on s390x for having more options in terms of job scheduling is much more urgent to us.

5.7 Job scheduling on hybrid clusters

In section 2.5, we have made a difference between homogeneous and heterogeneous jobs as the two basic workload categories on multi-architecture computing landscapes. This raises the question of how our Mesos infrastructure can be used to put these kinds of jobs into practice. While Mesos solely provides simple mechanisms like agent node labeling, using these mechanisms for the implementation of sophisticated scheduling behavior is a matter of the Mesos frameworks. We will therefore now take a look at how we achieved to realize the concepts of homogeneous and heterogeneous jobs by using existent primitives and APIs of the Marathon framework. We decided to skip the details for Aurora here because its scheduling constraint capabilities are not yet equally mature and powerful.

5.7.1 Homogeneous job scheduling

Distributing a homogeneous job's tasks over a group of uniform machines can be achieved by means of Mesos' labeling mechanism. Mesos allows to tag agent nodes with user-defined attributes [79]. Attributes in the context of Mesos are key-value pairs in the form `attribute: text ":"( scalar | range | text )` [79].

```
$ docker run mesosphere/mesos-agent --attributes="arch:amd64"
```

Listing 5.10: Mesos agents can be assigned custom attributes [79].

Listing 5.10 gives an example of how attaching attributes to mesos agent nodes works in practice [52]. Further Docker command line options have been omitted for clarity. Whenever the Mesos master sends resource offers to framework schedulers it passes along these agent attributes. From a Mesos framework's perspective, these attributes can be used to make arbitrary scheduling decisions. Marathon, for example, facilitates the definition of *constraints* [52] based on Mesos attributes. Constraints are created by applying operators on attributes and their respective values. Table 5.4 lists all operators which are currently available in Marathon.

Operator	Description
UNIQUE	Attribute value must be unique for each task to be scheduled.
CLUSTER	Tasks are distributed across nodes whose attribute value matches exactly.
GROUP_BY	Distribute tasks across nodes with different attribute values evenly.
LIKE	Matches a regular expression against an attribute value.
UNLIKE	Only choose nodes whose attribute values do not match.
MAX_PER	Limit number of tasks on hosts with a certain attribute value.

Table 5.4: List of constraint operators in Marathon [52].

According to the explanations in table 5.4, the CLUSTER operator is an appropriate instrument to partition a collection of servers and group them by their CPU architecture property. Although the GROUP_BY operator might also be a possible choice at first sight, it must be noted that this operator does not actually evaluate an attribute value, but instead only checks if the values provided by two or more agents are different from each other [52].

Listing 5.11 shows a sample request that creates a Marathon job consisting of Docker container tasks. Besides, it defines a constraint that instructs the scheduler to only accept resource offers from agents labeled as amd64 hosts.

```
$ curl -X POST -H "Content-type: application/json" 10.0.0.2:8080/v2/apps -d
'{
   "container": {
      "type": "DOCKER",
      ...
   },
   "constraints": [["arch", "CLUSTER", "amd64"]]
}'
```

Listing 5.11: Job with execution environment limited to amd64 hosts [52].

The only problem we met with respect to the definition of homogeneous jobs was that, before we examined the Docker Registry in more detail, we could not use the same image name for both the amd64 version and the s390x version for a Docker

image. This is because, as far as we knew at the beginning of our work, the Docker registry only allows 1:1 relationships between identifiers (i.e. names) to images. Assuming that, for example, a job consisting of a well-defined number of *nginx* Docker container tasks shall be launched on amd64 machines, we would have to supply *nginx* as the image name, while *s390x/nginx* would be the correct name for s390x-compliant Docker images. Although having to use different image names for the same application on different types of machines basically works and thus is not an exclusion criterion, it still undermines our goal of achieving maximum transparency for platform users. More precisely, users have to be aware of architecture-dependent image names instead of being able to leave choosing the appropriate image for a job to the infrastructure. In section 5.7.2, we show how more than one image can be stored behind a single identifier within a registry.

5.7.2 Heterogeneous job scheduling

Apart from the administrative overhead caused by several architecture-specific names for a single job's amd64 and s390x Docker images, launching homogeneous jobs could easily be achieved by means of Mesos attributes [79] and scheduling constraints [52]. However, this is not true for the scheduling of heterogeneous jobs. From a Mesos framework perspective, dropping all scheduling constraints is everything that is required to allow a job's tasks to be spread across machines regardless of their CPU architecture. Thereby, the framework simply ignores the "arch" attribute when receiving resource offers. As an alternative, the LIKE operator can be used along with a regular expression that matches both possible attribute values (s390x and amd64) [52]. Listing 5.12 illustrates both approaches by the example of a *nginx* job.

However, as we could only specify exactly one Docker image per job, defining multi-architecture jobs was still not possible at first. As we have already learned, the original implementation of the Docker registry was restricted to 1:1 mappings of identifiers to images, which are architecture-specific by nature. Taking these aspects into account, we can conclude that trying to launch containers from a Docker image, after is has been pulled from a registry, on different types of machines must fail. As a consequence, the job definition shown in listing 5.12 is effectively invalid.

```
# Target all types of hosts by simply dropping all constraints.
$ curl -X POST -H "Content-type: application/json" 10.0.0.2:8080/v2/apps -d
'{
   "container": {
       "type": "DOCKER",
       "docker": {
          "image": "nginx",
          ...
       },
```

```
        ...
    },
    "constraints": []
}'
# Target all types of hosts by defining LIKE to mach both amd64 and s390x.
$ curl -X POST -H "Content-type: application/json" 10.0.0.4:8080/v2/apps -d
'{
    ... ,
    "constraints": [["arch", "LIKE", "*."]]
}'
```

Listing 5.12: Two possible Marathon constraint definitions for heterogeneous jobs [52].

The problem of multi-architecture Docker images

So far, it has been shown that the platform-sensitivity of Docker images along with the fact that their names must be unique and can only refer to one image repository at the same time (which is, as we will show soon, no longer the case for the latest major version of the Docker Registry) causes additional overhead for the creation of homogeneous jobs, while these aspects render heterogeneous jobs entirely infeasible. Consequently, our next step was to think about possible workarounds for that issue. Here are some ideas we came up with after a while, accompanied by an assessment concerning their feasibility:

1. **Specify one image for each kind of machine type:** An obvious solution would be to introduce the possibility of specifying one Docker image for each possible value of the *arch* attribute in job definitions on the framework side. In this way, the scheduler could instruct the executor to either pull the *nginx* (for "arch:amd64") or the *s390x/nginx* (for "arch:s390x") image. Although such an approach might be functional, it is neither implemented in Marathon nor Aurora. As a consequence, this would not only require the development of a non-trivial patch but would also not help us with our goal of maximum transparency, as architecture-specific image names still remain.

2. **Specify one image tag for each kind of machine type:** As a tag is just a unique extension of a Docker image's name (see section 3.5.2) and points to a certain version of an image (e.g. myimage:s390x) [18], that solution cannot be used to bypass the 1:1 relationship of names to images and therefore did not prove as an option.

3. **Build images locally on target machines and use the same name:** This strategy proposes that on each type of machine, the images are built locally from different Dockerfiles (separate ones for amd64 and s390x), but under the same

name. Consequently, containers could be run on SEs and LPAR hosts by passing identical image names to the `docker run` command. However, that produces considerable effort for manually building the images on each machine separately and eliminates the advantages of having them in a single Docker registry. Establishing two different registries (one for s390x and another one for amd64 images) is also not an option, since the registry name is part of an image name (see 3.5.2) and thus is different for two registry servers. One Docker registry per host might be a functional solution, as each machine could refer to images by `localhost:$REGISTRY_PORT/image-name`. Although a high level of transparency could be preserved by means of this approach, managing one registry per host and keeping all of them in sync is simply too much work and thus is not a real option from our point of view.

A brief summary of the Docker Registry

In fact, it turned out that the second major release of the official Docker registry [15] supplies capabilities that can be employed to store multiple images for different CPU architectures under a common name. How that works can be best understood by taking a look at the traditional way of how storing images in a Docker registry works.

In section 3.5.2, it has already been clarified that each Docker image is made up of n layers which are stacked to a single consistent filesystem as soon as a container is launched from it. If an image is not present on disk when a `docker run` command is issued, it first gets pulled from a public or private Docker Registry [57]. From the registry's perspective, keeping each image as a single blob would be pointless and, since many layers are usually shared by multiple images [14], would quickly congest the disk with redundant data. Instead, storing each image layer separately and only once is much more efficient. This also applies in case multiple containers are launched from the same image, where having only one copy of a read-only image in memory is sufficient [14].

This approach, however, requires additional metadata per image that captures the layers that together form a certain image. In the context of the Docker registry, this metadata is represented by special data structures called *image manifests*. A manifest contains a set of *digests* (combinations of an algorithm and a hex-encoded hash value, e.g. `"sha256:123abc..."`) which are actually references to image layers residing in a registry or at the local disk [57].

So what happens under the hood when an image gets pulled from a registry is that, as a first step, the image manifest is retrieved from the registry server (see figure 5.3). Afterwards, the Docker engine goes through all the digests it finds in the manifest, downloads the layers as compressed tarballs [22] and unpacks them. The latest major

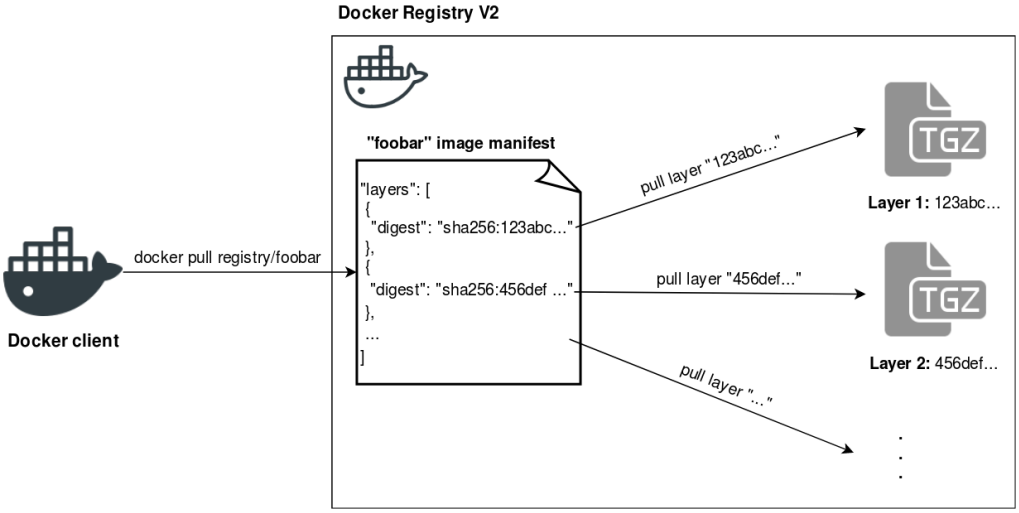


Figure 5.3: Docker Images are represented by manifests that point to n image layers.

release of the registry, Docker Registry V2, behaves in a content-addressable manner, because a digest not only serves as a unique identifier for a layer but, besides, is a cryptographically verifiable hash of a layer itself. A digest is calculated by applying a hash algorithm to the raw bytes of an image layer and append the result to the textual description of the algorithm that has been used [21].

In a more formal way, the digest d of an arbitrary image layer can be computed as follows:

$$d = "algorithm : " + EncodeHex(h(x)) \tag{5.1}$$

where x represents a layer's raw bytes which get passed to a hash function h (e.g. SHA-256). Appending the hex-encoded hash to the textual representation of the hash algorithm yields the image digest d [21].

Besides distinctively identifying an image layer, a layer's self-verifiability property facilitates ensuring data integrity by performing the hash computation at the client-side and compare the result to the digest in the image manifest. Of course, this leaves open how to guarantee the correct identity of a Docker image, i.e. from where to get the correct image manifest digest that corresponds to the image, and only the image, we originally intended to download. For that purpose, Docker applies the *Notary* service which shall not be further discussed [29].

Image Manifest Version 2, Schema 2

Version 2 of the image manifest schema specification for Docker Registry V2 [22] introduced the concept of *fat manifests* in order to enable multi-architecture images. Fat manifests are an extension of the original idea of having an image name pointing to exactly one manifest (i.e. Docker image). Instead, a fat manifest acts as an indirec-

tion, representing a list of one or more manifests that hold different values for the `platform` field in their manifest JSON structure (see listing 5.13). Each of these manifests is backed by a Docker image consistent with the indicated target architecture [22].

```
$ curl -X PUT -H "Content-type:application/json" \
> 10.0.0.2:$REGISTRY_PORT/v2/myimage/manifests/mytag
{
   "schemaVersion": "2",
   "manifests": [
      {
         /* manifest A */
         ...,
         "platform": {
            "architecture": "amd64"
         }"
      },
      {
         /* manifest B */
         ...,
         "platform": {
            "architecture": "s390x"
         }
      },
      ...
   ]
}
```

Listing 5.13: A fat manifest for multi-architecture images [22].

From a user's perspective, there is nothing special that has to be respected when pulling a multi-architecture image from a registry. As can bee seen from the output of the `docker info` command, the Docker client is completely platform-aware. Thus, it is able to traverse the entries of a fat manifest and to autonomously pull the correct image manifest whose `platform` property matches. This enables operation at maximum transparency, because everything that is required to distribute a job across a couple of heterogeneous machines is the name of the (platform-independent) fat manifest [22, 33].

Since there is currently no Docker-native functionality to build and push multi-architecture images in a convenient way and because typing things manually is cumbersome and error-prone, some third party tooling emerged around that issue. One of these projects is the *manifest-tool* started by Estes [33] (IBM Senior Tech Staff Member and Docker maintainer), which facilitates condensing multiple platform-specific images into a fat manifest and pushing it to a registry. The manifest tool, whose source

code can be found on Github[11], uses the *YAML* format for its input files [33]. Listing 5.14 gives an impression of how such a YAML-based definition file for creating a multi-architecture *nginx* image might look like.

```
image: myrepo/nginx-universal:1.13
manifests:
  -
    image: nginx:1.13
    platform:
      architecture: amd64
      os: linux
  -
    image: s390x/nginx:1.13
    platform:
      architecture: s390x
      os: linux
```

Listing 5.14: Sample YAML file for creating a fat manifest.

[11] https://github.com/estesp/manifest-tool

Chapter 6

Evaluation

While the previous chapter has introduced a possible approach to build a hybrid microservice infrastructure upon Mesos, the focus shall now be laid on its evaluation. We assessed our prototype with regard to three aspects we considered the most essential ones: First of all, an overview of the its basic computing resource demands (disk space and memory) shall be provided. Second, we present the results of some tests we carried out in order to check how switching between Mesos frameworks affects job deployment, scaling and cleanup performance. Third, we performed several experiments in order to verify the availability and failover guarantees given by the individual components of our infrastructure setup.

Executing the resource consumption and performance measurements on a CentOS VM running on a single workstation equipped with an AMD A4-400 APU processor and 8GB of RAM was sufficient and especially prevented network latencies from distorting our performance tests. For local testing purposes, we developed and released a Mesos playground[1] which launches a complete Mesos infrastructure on a single host by means of Docker Compose. For reliability evaluation, we moved back to our virtualized multi-host setup from section 5.1.

At least, it is examined the solution we came up with complies with the remaining basic requirements which have been summarized in section 2.5.1. This brings us to the existing limitations of the current setup as well as possible improvements for the future.

6.1 Resource consumption

We started our evaluation process with a glance on our infrastructure setup's resource requirements. This is a highly critical point, because we aim at keeping them as low

[1] https://github.com/apophis90/mini-mesos

as possible in order to leave a maximum of RAM and CPU capacity to application deployments. This is even more important on mainframes, where the amount of available computing resources is more strictly limited compared to data centers.

In that sense, the first part of this section is about the observations we made when examining the amount of disk space that is occupied by the Docker images for Mesos as well as its peripheral components. Afterwards, we show the results of our measurements in terms of memory consumption when putting the infrastructure software stack under load. At least, we present the outcome of some tests we carried out in order to check how switching between Mesos frameworks affects job deployment and scaling performance.

6.1.1 Disk space requirements

Depending on the programming language in use, possible runtime dependencies and the size of a containerized application itself, the surrounding Docker image tends to grow up to a size of hundreds of megabytes quickly. We took this as a reason to examine the amount of disk space that is occupied by all the images that make up our infrastructure stack. When discussing the size of Docker images, a difference must be made between an image's *virtual* size and its - as we call it - *actual* size. We chose this as a designation to underline that there are actually two different dimensions in terms of image size that must not be confused. Mouat [57] explains that *virtual* size describes the size of a Docker image including all the layers from its base image. The base image relation is transitive, meaning that a Docker image's direct base image might also be derived from another base image as well. On the other hand, by *actual* size we mean the size of a Docker image without including the base image layers, considering only the layers that have been introduced by the uppermost Docker image. To express it more formally:

$$actualSize(image) = virtualSize(image) - virtualSize(parentImage), \tag{6.1}$$

where $virtualSize(image)$ denotes the size returned by the `docker images` command for *image*. The graph in figure 6.1 illustrates the virtual and actual Docker image sizes we measured for the amd64 variants of our images. The sizes of the s390x images, as far as they already exist, are similar.

While the `/var/lib/docker/overlay/` directory on our local CentOS test VM, where the image layers for the overlay driver reside, was about 3.0GB in size, adding up the actual image sizes from figure 6.1 results in about 1.8GB of disk capacity. Note that this is not a perfectly accurate calculcation, we we count the Mesos installation

for the master and worker image twice.

Consequently, what this means is that from 3.0GB of storage for the image layers in our case, roughly 1.2GB ($\approx 40\%$) were occupied by the base images that provide a root filesystem, miscellaneous dependencies and runtime environments. Looking at the virtual image sizes in figure 6.1 shows that adding them up results in far more than 3.0GB. The reason for this discrepancy is that some of the layers are shared between several images and so they are included in the virtual sizes of different images multiple times. Putting it the other way around, the total amount of disk space required by all Docker images on a host cannot be calculated by simply adding up their virtual sizes.

As soon as we launched a single instance of our Mesos infrastructure on our local VM, including ZooKeeper as well as Marathon and Aurora, the size of the `/var/lib/docker/overlay/` folder grew up to a size of about 3.3GB, which means that the container layers amount to approximately 300MB of disk space. Summing up, our infrastructure setup required about 3.0GB disk capacity when Mesos was stopped, and about 3.3GB assuming that it was running.

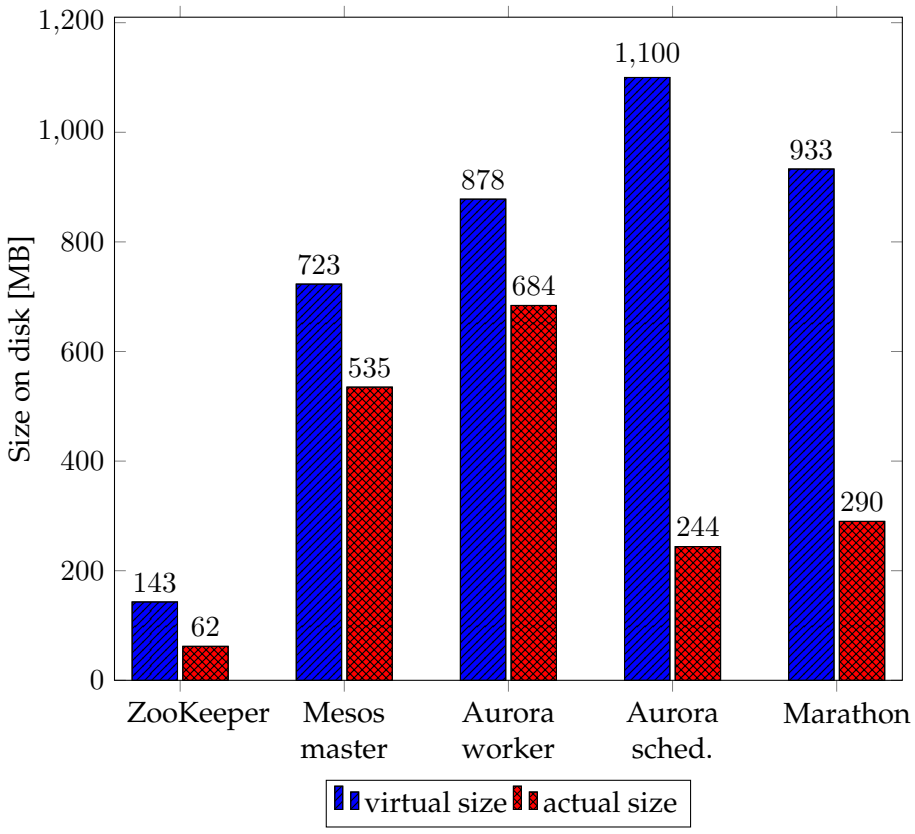


Figure 6.1: Virtual and actual sizes of the infrastructure Docker images.

The difference between virtual and actual image size is particularly noticeable for the Mesos framework images, which is caused by the full Mesos installation they have

to carry along as a runtime dependency. We expect this situation to improve once the maintainers start migrating Marathon and Aurora towards using the scheduler HTTP API [84] for communicating with the Mesos master.

Summary

The most significant findings we could extract from these experiments was that striving at keeping base images as lean as possible is very important. This is suggested by the significant differences we measured between the virtual and actual sizes of our images which proved that about 40% of disk space is occupied by their base images. Because we think that the share of the base image layers but also the 3.0GB of disk space needed in total is just too much, we plan to invest time and efforts in the future in order to replace the Debian and Ubuntu base images, which are usually quite extensive, with much smaller images like Alpine [23].

6.1.2 Memory consumption

For the memory usage measurements, we launched a full Mesos stack based on our Mesos playground and used the `docker stats` command to observe the amount of RAM used by each of the infrastructure containers. As each of the components is replicated three times, we took the memory usage of the three containers per component and calculated the average value. Our goal was to benchmark the memory consumption per infrastructure component as a function of the number of tasks managed by Meso. We started in idle state (task count of 0) and scaled up a single job by 10 tasks per step up to a task count of 100.

In order to observe how employing different Mesos frameworks for task scheduling influences memory consumption, we split the experiment into two parts: In both cases, we used exactly the same environment with the exception of having swapped the Mesos framework. While Marathon was in use for the first part, it has been replaced with Aurora for the second part. As for the tasks we deployed to simulate different levels of load during our tests, we packaged up a minimal statically compiled Go application in a *scratch* Docker image[2] (which basically consists of an empty root filesystem), resulting in a final image size of about 5.8MB. In this way, we were able to execute a large number of instances of a fully functional web application with minimal resource overhead with respect to the tasks. Throughout the entire benchmark procedure, the workload could be assumed to be evenly distributed across the three agent instances.

[2] https://hub.docker.com/r/pkleindienst/hello-go/

Figure 6.2 shows the outcome of the first part of our memory benchmarking experiments. It can be seen that the average memory usage for the Mesos agent containers grows approximately linear with the number of tasks. The connection between agent workload and memory usage seems appropriate and is exactly what could be expected. Another interesting observation is that the RAM demands of the other infrastructure parts only increases moderately and is, for example, nearly constant for the Mesos masters.

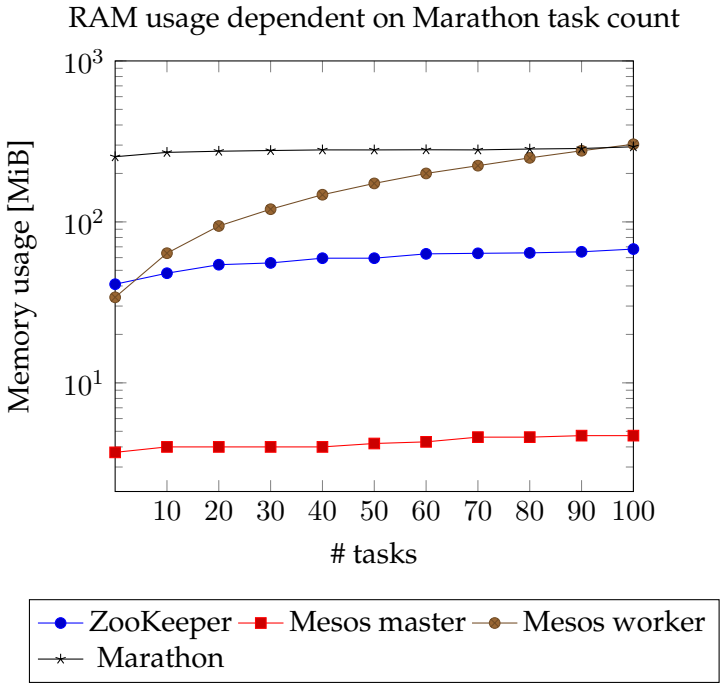


Figure 6.2: Mesos and peripherals RAM consumption with Marathon.

Our second memory benchmarking setup (see figure 6.3), which used the Aurora framework instead of Marathon for task deployment, shows similar curve characteristics. As before, the graph points out that the memory demands of ZooKeeper, the Mesos masters and the framework are almost constant, with the framework allocating about 300MB of RAM on full load. In contrast to the first graph, however, the Mesos agent group's average memory requirements are much greater when Aurora is applied as the scheduler.

We concluded that this is caused by the custom Thermos executor used by Aurora which seems to have much higher memory demands than the Mesos built-in command executor. During our first measurements with Aurora, we were unable to scale up the Go application to 100 instances on our local VM as each Thermos executor process allocated about 128MB of RAM and a CPU share of 0.25 (25% of CPU time) by default. We finally managed running 100 tasks in parallel by limiting a Thermos executor's requirements to 32MB of RAM and a CPU share of 0.01 respectively. Interestingly, we experienced a sudden drop in the Mesos agents' RAM allocation when the Aurora

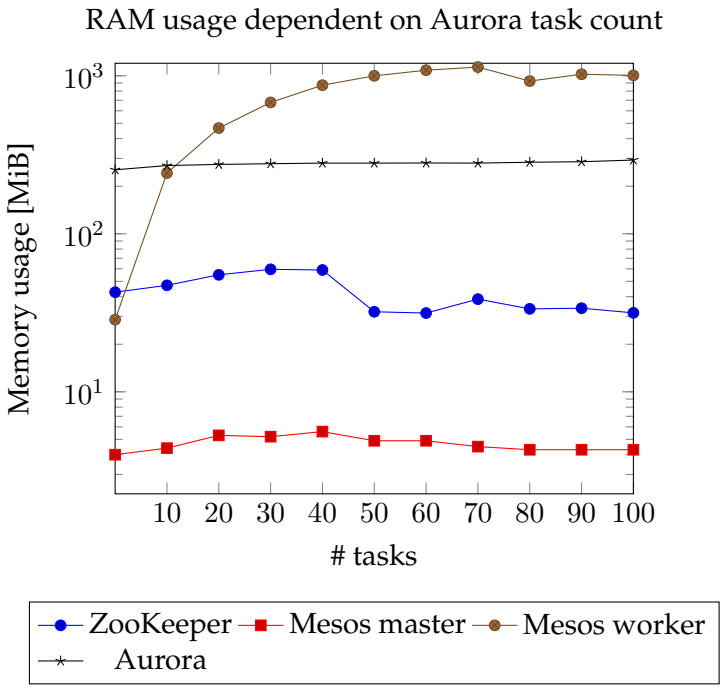


Figure 6.3: Mesos and peripherals RAM consumption with Aurora.

task count reached 80 without being able to find a particular reason for this behavior. We guessed that this could the result of both memory being freed and/or measurement inaccuracy, as we would expect the increase in an agent's memory reservation to be linear with its workload, as we have measured in figure 6.2.

Summary

Table 6.1 summarizes our results, showing that the average RAM allocation by the Mesos agent group grows roughly linear with the number of tasks when either Marathon or Aurora is used. With Aurora, we experienced a sudden rise followed by fall in the measurement curve with a task count between 50 and 80. On the basis of our settings, especially in terms of the resource constraints for the Thermos executors, it can be determined that when Aurora is applied instead of Marathon, the Mesos agent memory consumption is about three times as high. In order to lower the resource demands for Aurora, it would be an interesting experiment, from our point of view, to find the lowest possible resource specifications that the Thermos executor can run on for a certain task.

# tasks	with Marathon	with Aurora
0	34.0MiB	28.6MiB
20	94.3MiB	242.4MiB
40	147.5MiB	466.7MiB
60	200MiB	1084.3MiB
80	249.9MiB	935.3MiB
100	303.6MiB	1005.8MiB

Table 6.1: Average Mesos agent resource consumption depending on the framework in use.

6.2 Performance

Another metric of interest is how fast both framework schedulers can create, scale up, scale down and remove jobs. The graph shown in figure 6.4 illustrates our results. With respect to our job creation or rather scale-up measurements, the y axis indicates the elapsed time between the submission of the request and the point when the last task was reported to be in RUNNING state. Accordingly, it shows the period between the request and the point when the last remaining task was in STOPPED state for our job deletion or rather scale-down tests. Every measurement has been performed on the basis of a clean cluster state (i.e. all infrastructure components have been restarted), and we allocated 32MB of RAM and a CPU share of 0.01 per task.

We found that with Marathon, it took about $10s$ to launch a job comprising 10 tasks (10 instances of our sample Go web application). Scaling up from x number of tasks to $x + 10$ tasks again completed after about $10s$. We also measured a delay of about $10s$ when reducing the amount of tasks from x to $x - 10$ with the Marathon scheduler. Surprisingly, fully destroying a job of 10 tasks with Marathon only took about half its creation time ($5s$). Further measurements showed that these durations roughly increase linearly with the number of tasks.

Switching to the Aurora framework produced clearly different results for the same measurements. While initially deploying a job comprising 10 tasks (Go web application instances) finished after approximately $20s$, which is twice the time it took with Marathon, we also found that scaling up by 10 tasks caused a delay of 480s (or 8 minutes) in average! Because we specified exactly the same resource requirements for our reference job when performing both experiments, we assume that the resource allocation and startup of the Thermos executors is responsible for this delay. Interestingly, reducing the number of tasks by 10 and revoke their resources, as we did

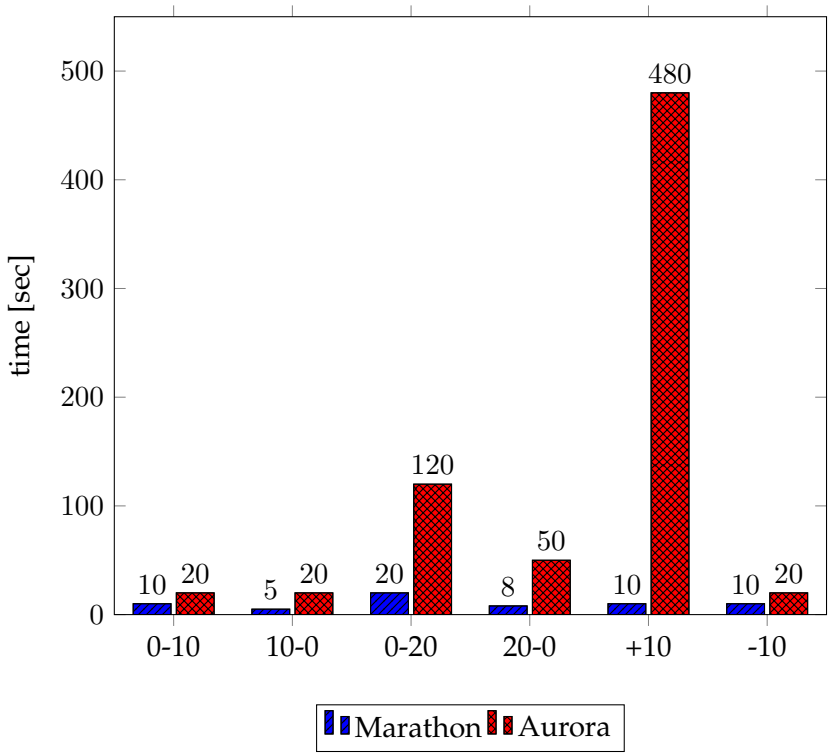


Figure 6.4: Average job startup, scale-up, scale-down and shutdown duration per framework.

with Marathon previously, complies with the initial startup time for 10 tasks, which was about 20s for Aurora. This also holds for the time required to fully cleanup a job comprising 10 tasks, which takes Aurora another 20s.

Summary

In sum, our results show that Marathon seems to be much more performant in every aspect of job administration. Admittedly, we cannot definitely rule out our local workstation testing environment as a reason for the bad performance of Aurora. What, however, can be said against this assumption is that Marathon performed surprisingly well on exactly the same setup.

6.3 Reliability testing

Apart from low resource consumption, the second key characteristic of a microservice infrastructure should be fault tolerance and high availability from our perspective. The different pieces of our software stack, namely ZooKeeper, Mesos, Marathon and Aurora, provide a large number of reliability guarantees under certain failure conditions. These guarantees should be verified, though, as for example a faulty configuration or bugs might cause some tools to behave different from what is actually expected. Thus, we decided upon simulating several error condition scenarios by purposefully

infiltrating failures into our running system and comparing our expectations to the actual outcome.

6.3.1 ZooKeeper

ZooKeeper follower failure

We started our reliability tests with killing one of the ZooKeeper follower process, leaving the leading master intact. In case a ZooKeeper follower fails, we expect no ZooKeeper leader election to take place and no harmful side effects to occur on the other infrastructure components. That means frameworks and their tasks should continue executing without interruption. If any of the dependent processes has an active session with a follower which has just failed, this process is expected to reconnect to another ZooKeeper node after a timeout has elapsed. Of course, these expectations assume that, after a ZooKeeper follower has gone away, a quorum (for example 2 out of 3 nodes) is still available. If this is not the case, then the remaining ZooKeeper servers will stop serving requests.

Our tests have shown that ZooKeeper behaves as expected and keeps itself as well as the other infrastructure components alive. But we also found that in very rare cases, the leading Marathon scheduler went down when a ZooKeeper follower had been killed before. Further investigations have proved that this only happened if, by accident, we had stopped the ZooKeeper process the leading Mesos master had been connected to. It turned out that, inadvertently, we had sometimes triggered a Mesos master leader election, which caused the leading Marathon scheduler to fail. More details on this are given below.

ZooKeeper leader failure

When we simulated the failure of the current ZooKeeper leader, we have met the exact same behavior as above, with the exception of a leader election taking place given that a majority of ZooKeeper servers was still available. Assuming this is not the case, for instance because 2 out of 3 ZooKeeper servers have become unavailable, we would expect the entire stack to be still alive but rejecting any requests that aim to modify the state of the cluster, like the deployment of new tasks.

When a majority of ZooKeeper servers was still active, our setup behaved as we expected. If this was not the case, then, contrary to our expectations, the Mesos and Marathon processes also terminated immediately because there were unable to connect to ZooKeeper. According to the ZooKeeper logs, the remaining servers immediately close any incoming Transmission Control Protocol (TCP) connection attempts in order to avoid state inconsistencies. The Aurora processes were the only ones that stayed alive for a while, but they also committed suicide after a while when they could still not find a ZooKeeper quorum.

6.3.2 Mesos

Mesos agent failure

Another error condition that might occur is a Mesos agent going down, which could have two different reasons: On the one hand, the respective Mesos worker container could have exited. In that case, we would expect the agent's tasks to keep running, while affected frameworks should schedule replacement tasks to the remaining Mesos agents. For our setup, we could verify that this works as intended. However, there is one aspect that we missed at the beginning, as we also expected the failed agent to wipe off its redundant tasks as soon as it recovers, as described in the documentation [80], which was not the case during our experiments. It turned out we initially overlooked that when a Mesos agent container terminates, so do the task executors which are running in the same container. Consequently, a restored agent is neither aware of the tasks it managed before, nor is it able to destroy them as their executors are gone. Without manual intervention, these tasks would keep running forever. That reveals an obvious drawback of operating Mesos in containers and requires additional work, for example in the form of a startup hook which deletes all containerized tasks as soon as the Mesos agent container restarts. We do not have a concrete implementation for this yet, though.

The second reason, that consists in the entire host system going offline, can be handled much more comfortably, as all containers, including Mesos agent along with its tasks, are killed. Consequently, one does not have to worry about cleaning up any unreachable tasks.

Standby master failure

Because all dependent processes, namely the Mesos agents and framework schedulers, only have active TCP connections with the leading Mesos master, we did not expect the shutdown of a standby master to have any side effects on our infrastructure. We could confirm this hypothesis during our experiments, except for the situation when all 2 standby masters were offline. In this case, we observed that Aurora started rejecting read and write requests, which makes sense as Aurora uses the Mesos replicated log for backing up its state. If no Mesos quorum is available, the remaining Mesos masters reject all write operations to the log in order to keep it in consistent state. In terms of the configuration, we found that setting the correct Mesos quorum size for Mesos and Aurora is important to obtain the correct replicated log behavior.

Leading master failure

We tested the Mesos master failover procedure by deliberately killing the leading master process and keeping an eye on the other components' reactions. Assumed that ev-

erything works as expected, the remaining master nodes should elect a new leader, which the agents and frameworks can reconnect to afterwards. Our investigations verified this is actually the case, but also revealed a substantial difference between the Marathon and Aurora in terms of handling Mesos master failover: While all Aurora scheduler instances continued without interruption, the leading Marathon scheduler exited immediately. We first thought this might indicate a bug, but after having studied the logs in detail it seemed likely to us that this is just how Marathon implements the notion of "fail fast". What that means is the leading scheduler, after having lost its connection to the active Mesos leader, gives up upon searching for a new leading Mesos master instantaneously. Instead, it self-terminates quickly in order to open the way for another scheduler instance to become leader and eventually reconnect to a newly elected leading Mesos master. Aurora, on the other side, chooses another approach and rather waits for a certain period of time after the leading scheduler has lost its Mesos master connection. If it has no success in reestablishing this connection within a predefined timeout or it cannot detect a Mesos quorum to be live, it finally self-terminates. The common idea behind these different implementations is the prevention of inconsistencies concerning the overall cluster state. For automatically restarting terminated scheduler processes, it might be reasonable to rely on the OS-native init system, for example systemd.

6.3.3 Marathon

Following scheduler failure

Because the way the Marathon framework uses ZooKeeper for leader election is not majority-based and its entire state is also stored in ZooKeeper, we came to the conclusion that, out of 3 scheduler instances, it should be able to remove the 2 following schedulers from the cluster without causing any harm to our infrastructure or running tasks. Our results showed that this is actually the case, and we could issue read as well as write operations which were carried out by the the leading Marathon scheduler without problems.

Leading scheduler failure

When the leading scheduler fails instead, a standby master is expected to take over its role. Of course, this requires at least one more scheduler instance to be live. We could not find any anomalies when killing the Marathon leader and thus leader election worked without any difficulties. That also includes that all existing tasks continued to run without interruption and that the cluster state as seen by the newly elected leading scheduler did not diverge from the actual cluster state as it was before the previous leader was taken down. This observation was important, as we strived for making sure we did not end up in an inconsistent cluster state.

6.3.4 Aurora

Following scheduler failure

We tested the outage of a following Aurora scheduler the same way as we did in the case of Marathon. Fortunately, we also experienced the same results, so we could be sure that even a single remaining Aurora scheduler carries out read and write operations without any issues.

Leading scheduler failure

In terms of leading scheduler failover in Aurora, we had exactly the same expectations as of Marathon. Our experiments yielded the same results, as we could observe an uninterrupted execution of deployed tasks as well as the preservation of a consistent cluster state.

6.3.5 Summary

On the basis of the results we received and the new insights we gained through our experiments, we have essentially learned four important things: First, we were able to improve our configuration settings of several cluster components and found various critical parameters, like the size of the Mesos master quorum which manages the replicated log, that deserve special attention.

Second, we found that, all in all, the different infrastructure components comply with the promises they make related to reliability and availability. The only thing that is left for future work is dealing with obsolete tasks that became detached from their executors due to a failed Mesos agent container. As there is no chance for these tasks to rejoin the cluster, a thoughtful procedure must be defined which identifies and kills such tasks after a Mesos agent has terminated. Simply ignoring these tasks is not an option as we cannot assume the host systems to be rebooted in short intervals, so remaining tasks could quickly accumulate and occupy valuable resources.

Third, putting all infrastructure components into containers, apart from the that issue, proved to be an efficient approach. It not only simplified installation due to dependency isolation, but also facilitated life cycle management and accelerated bootstrapping within our hybrid demo environment as well as on our local workstation. Additionally, containers made it much easier to deploy multiple versions of the same software side by side, as we did in order to test ZooKeeper's dynamic reconfiguration capabilities. Another advantage we see is that containers and their isolation mechanisms can help the microservice infrastructure with becoming extremely robust against host OS software updates.

At least, we have shown that we can reliably operate our microservice management stack even at small scale rather than needing hundreds or even thousands of

servers. It should therefore be possible to safely apply Mesos in mainframe environments.

6.4 Evaluation of remaining mandatory requirements

We will now elaborate on to what extent the remaining mandatory requirements we presented in section 2.5.1 are met by our prototype.

Resource abstraction

In terms of computing resource abstraction, we have shown that Mesos can be used as an abstraction layer for fine-grained resource sharing even beyond architectural boundaries. Developers, operators and other users of our infrastructure can transparently allocate the resources they need for their deployments. The available resources per user can furthermore be limited through the definition of Mesos roles [79] to keep a single user from blocking other deployments by allocating a great quantity of computing resources in advance.

Different types of jobs

By means of agent labels and multi-architecture manifests in the Docker registry, we even achieved to provide a way for homogeneous and heterogeneous deployments that reduces the need for detailed knowledge of the underlying physical system and frees operators from having to deal with the platform dependency of Docker images. Moreover, is has been explained how the Mesos approach of pushing responsibility of task scheduling to dedicated frameworks enables the coexistence of multiple application-specific scheduling policies which coordinate the distribution of their tasks with respect to different goals. That also includes the various demands coming along with different types of jobs, like long-lived services or ad-hoc jobs.

Optimized hardware utilization

We are aware that having multiple frameworks which optimize task scheduling with regard to various goals conflicts with the desire for globally maximized hardware utilization, as pursued by a centralized scheduler. By and large, we think that Mesos' inherent flexibility in terms of workload and scheduling must be weighed against overall utilization and that the added value of using Mesos increases with the diversity of the workload that shall be deployed. Performing additional experiments in order to examine compatibility of globally optimized utilization and application-aware scheduling was beyond the scope of this thesis.

Deployment and scaling automation

Our setup reduces the expense of application deployment to the submission of a few commands via the Marathon UI or the Aurora CLI, and we consider this as an adequate degree of deployment automation. Scaling existing deployments up and down also requires manual framework interaction and only the replacement of erroneously terminated tasks works without intervention. We think this is a sufficient level of scaling automation, as a system that scales workload in a fully autonomous manner would be exposed to Denial of Service (DoS) attacks.

Dynamic service discovery

By using Mesos-DNS, we were able to setup our own lightweight DNS server that allows tasks deployed on top of Mesos to discover each other by following a uniform naming schema.

Isolation of tasks

As Mesos is agnostic with regard to the actual format of deployments, we were able to run Mesos frameworks that optionally allow for running tasks as Docker containers. In this way, workload can be isolated from the underlying host system which significantly reduces the amount of packages that must be installed on the host and ensures that dependencies from different services do not interfere in a harmful manner.

6.5 Current limitations and thinkable improvements

Aside from the positive characteristics of our solution, its downsides as well as the remaining work must not be ignored. At the beginning, it has been clarified that, for example, authentication and authorization are essential subjects, but that additionally covering them would have massively enlarged the scope of our work and, besides, makes much more sense as soon as a basically functional prototype is in place from our point of view. Nevertheless, we decided upon including these and other aspects that currently constitute serious limitations into this review.

Authentication and Authorization

Section 2.4 already indicated that securing a Mesos-based infrastructure setup has multiple dimensions, like authenticating not only users but also frameworks and agents joining the cluster. None of these features is mandatory or active by default, and we did not activate any of them during our work on this project. However, we appreciate that taking care of security mechanisms should be done as early as possible and that

systems should be secure by design, as pursuing security by the end of a project is not only complicated but also much more expensive.

Monitoring

Our current setup does not include any instruments for watching the health state of deployments and infrastructure components. Adding appropriate capabilities by integrating mature third party tools should also be done in an early phase of microservice infrastructure projects. During the prototype development phase, we already performed first experiments with *Prometheus* [64], a tool for exporting and collecting process metrics as time series.

Persistent volumes

Another important point that is missing so far is a fully functional solution for efficiently sharing disk space between multiple containerized tasks deployed on our Mesos stack. A possible solution should meet two essential conditions: First, it must not be framework-specific. Second, it should allow creating volumes backed by one or more local disks instead of cloud storage only, as this is not an option for critical businesses.

Docker images storage demands

In section 6.1.1, we presented our measurement results regarding the disk space consumption of our core infrastructure Docker images. These include ZooKeeper, Mesos master and slave, Marathon and Aurora. It has been demonstrated that the individual images sizes amount to about 3.0GB in total per host system. Even though packing infrastructure components into containers gives us various benefits, we consider 3.0GB of disk space per host too much. A possible solution to that problem is the replacement of bloated Ubuntu and Debian base image with their much smaller Alpine counterparts. Furthermore, reusing as much image layers as possible also helps with reducing storage complexity.

Rigid quorum size

Currently, the master quorum size in Mesos [43] (throughout all versions) and also in ZooKeeper [93] (up to version 3.4) cannot be reconfigured with zero-downtime. While ZooKeeper integrates the necessary capabilities in the upcoming 3.5 version, there is no explicit indication when this will be possible in Mesos. This means that whenever the quorum size of a ZooKeeper or Mesos master group shall be changed the entire group must be restarted, which, especially in the case of ZooKeeper, could easily cause

dependent processes to terminate. Dynamic reconfiguration would be a favorable feature for our mainframe environment, as in this way, for example a SE that has failed could simply be replaced with another LPAR. Until dynamic reconfiguration is fully supported, we will, however, have to adhere to a static quorum configuration.

Detached Mesos tasks

Section 6.3.2 already described the issue of containerized tasks becoming detached from their executors if a Mesos agent container fails for some reason. Given that an OS-native init system like systemd is used for automatically restarting Mesos master and agent containers, we propose the definition of a pre-start hook as a simple and effective approach to that problem. This could be implemented as a simple shell command that uses the Docker CLI to filter and destroy the detached tasks, which is adequate as we can assume the frameworks creating replacement tasks for them. Another thinkable option is running executors in their own Docker containers outside of the agent container, which introduces the problem of having to find a way to start the task container from within the executor container. We are currently investigating how this is supported by Mesos and existing frameworks.

Chapter 7

Conclusions and future work

7.1 Summary

In this thesis, a reference implementation of a microservice infrastructure for hybrid computing environments has been introduced, using IBM System z as an exemplary target platform. We started with describing our concrete motivation behind this project, explaining that shifting systems management applications from external hardware devices onto the platform itself could help with eliminating or at least reducing the need for this hardware. As a side effect, the application architecture is pushed towards a highly distributed microservice-based design. A major challenge that shows up when adapting such an approach with regard to mainframes is the life cycle management of a large number of services across a amd64/s390x multi-architecture platform. In order to examine how this can be achieved, we engineered a basic prototype which is able to manage generic workload on a setup comprising two amd64 SEs as well as single s390x LPAR.

We proved that, by plumbing together various existing infrastructure components around Apache Mesos and relying on OS-native process isolation capabilities in the form of containers, a reliable and flexible microservice platform for homogeneous and also heterogeneous jobs can be built. Our solution has entirely been developed on top of available features like node labeling or fat manifests for Docker images, without the need for adding new features to open source projects. Based on what was already there, we finally ended up with a setup that enables the definitions of deployments, whether containerized or not, to be highly transparent with regard to the underlying hybrid computing landscape. Furthermore, it abstracts away the platform-specific details of Docker images and makes picking the correct executable for a particular type of machine a concern of the infrastructure itself, rather than leaving this to the operator.

Moreover, as we operate Mesos and its peripheral components in containers as well, we can make use of community-driven Linux distributions like Ubuntu or Alpine and their rich ecosystems as an abstraction layer for the oftentimes highly customized

enterprise Linux flavors used in the field of mainframes. Because Docker images can be equally used on any host that has the Docker engine installed, only one set of images per target architecture had to be built for being able to bootstrap our setup on all Linux distributions. Besides, this simplifies the provisioning of amd64 and s390x VMs as the number of packages which must be installed beforehand is reduced to Docker and its dependencies.

7.2 Outlook

The hybrid microservice infrastructure we presented is a prototype and still in an early stage of its development. At the end of the last chapter, a short overview of existing limitations has been provided, serving as a rough guiding framework regarding the aspects our focus will be placed on for the next steps. This will certainly also involve bringing a lot more tools to the s390x platform, which is lightened by the increasing popularity of programming languages like Go, which are not only type and memory safe, but also simplify the creation of statically compiled binaries for easy deployment. Besides, we are already exited to see how our work is actually applicable to IBM System z firmware, where the motivation for this project originated from.

In conclusion, we want to emphasize that from our perspective, turning towards hybrid computing is not only relevant with regard to special environments like mainframes. As ARM chips, which show a much lower energy consumption than their amd64 counterparts, will become cheaper, they will probably become even more attractive for large-scale data centers. Consequently, issues and challenges similar to these we faced during our work will surely appear on other computing platforms. While we of course cannot raise a claim to have answered all questions related to hybrid clustering with this thesis, we still hope we have made valuable contributions with regard to the upcoming evolution of multi-architecture computing.

Appendices

Appendix A

amd64 Dockerfiles, shell scripts & configuration files

A.1 Mesos-DNS Dockerfile

```
FROM ubuntu:latest

RUN apt-get update && apt-get install gnupg wget bash -y

ARG MESOS_DNS_VERSION=0.6.0
ARG ARCH=amd64
ARG OS=linux
ARG MESOS_DNS_PORT=53
ARG MESOSPHERE_PUB_KEY=111A0371BD292F47

WORKDIR /mesos-dns

RUN wget https://github.com/mesosphere/mesos-dns/releases/download/\
v$MESOS_DNS_VERSION/mesos-dns-v$MESOS_DNS_VERSION-$OS-$ARCH \
    && wget https://github.com/mesosphere/mesos-dns/releases/download/\
v$MESOS_DNS_VERSION/mesos-dns-v$MESOS_DNS_VERSION-$OS-$ARCH.asc

RUN gpg --keyserver pgpkeys.mit.edu --recv-key $MESOSPHERE_PUB_KEY \
    && gpg --verify mesos-dns-v$MESOS_DNS_VERSION-$OS-$ARCH.asc \
mesos-dns-v$MESOS_DNS_VERSION-$OS-$ARCH

RUN rm mesos-dns-v$MESOS_DNS_VERSION-$OS-$ARCH.asc \
    && mv mesos-dns-v$MESOS_DNS_VERSION-$OS-$ARCH mesos-dns \
    && chmod 700 mesos-dns

EXPOSE $MESOS_DNS_PORT

ENTRYPOINT ./mesos-dns -config=./config.json
```

A.2 Aurora scheduler Dockerfile

```
FROM mesosphere/mesos-master:1.1.1

ARG AURORA_VERSION=0.17.0
ARG MESOS_REPLICATED_LOG_PATH=/var/lib/aurora/scheduler/db
ENV USER=aurora

RUN apt-get update && \
    apt-get install -y software-properties-common && \
    add-apt-repository -y ppa:openjdk-r/ppa && \
    apt-get update && \
    apt-get install -y openjdk-8-jre-headless wget && \
    update-alternatives --set java /usr/lib/jvm/\
    java-8-openjdk-amd64/\jre/bin/java

RUN wget -c https://apache.bintray.com/aurora/ubuntu-trusty/\
aurora-scheduler_${AURORA_VERSION}_amd64.deb && \
    dpkg -i aurora-scheduler_${AURORA_VERSION}_amd64.deb

RUN service aurora-scheduler stop && \
    sudo -u aurora mkdir -p $MESOS_REPLICATED_LOG_PATH && \
    sudo -u aurora mesos-log initialize --path=$MESOS_REPLICATED_LOG_PATH

COPY scheduler.sh ./scheduler.sh

RUN chown $USER:$USER /home/$USER/scheduler.sh && \
    chmod 500 /home/$USER/scheduler.sh

USER $USER

WORKDIR /home/$USER

ENTRYPOINT ["./scheduler.sh"]
```

A.3 Aurora scheduler.sh script

```
#!/bin/bash

AURORA_HOME=/usr/sbin
AURORA_FLAGS=(
-http_port=${HTTP_PORT:-8080}
-cluster_name=${CLUSTER_NAME:-aurora}
-backup_dir=${BACKUP_DIR:-/tmp/aurora/backup}
-mesos_master_address=${MESOS_MASTERS:?"MESOS_MASTERS must not be empty"}
-serverset_path=${AURORA_ZK_PATH:-/aurora}
-zk_endpoints=${ZK_ENDPOINTS:?"ZK_ENDPOINTS must not be empty"}
-native_log_file_path=${NATIVE_LOG_FILE_PATH:-/var/lib/aurora/scheduler/db}
-native_log_zk_group_path=${NATIVE_LOG_ZK_GROUP_PATH:-/tmp/aurora/\
replicated-log}
-hostname=${HOSTNAME:?"HOSTNAME must not be empty"}
-native_log_quorum_size=${NATIVE_LOG_QUORUM_SIZE:-1}
-allowed_container_types=${ALLOWED_CONTAINER_TYPES:-MESOS,DOCKER}
-thermos_executor_path=${THERMOS_EXECUTOR_PATH:-/usr/share/aurora/bin/\
thermos_executor.pex}
-allow_docker_parameters=true
)
exec "$AURORA_HOME/aurora-scheduler" "${AURORA_FLAGS[@]}"
```

A.4 Aurora worker Dockerfile

```
FROM mesosphere/mesos-slave:1.1.1

ARG AURORA_VERSION=0.17.0
ENV USER=aurora
ENV HOME=/home/aurora

RUN apt-get update && \
    apt-get install -y python2.7 wget libcurl4-nss-dev

RUN wget -c https://apache.bintray.com/aurora/ubuntu-trusty/\
aurora-executor_0.17.0_amd64.deb && \
    dpkg -i aurora-executor_0.17.0_amd64.deb && \
    useradd $USER --create-home --shell /bin/bash && \
    usermod -aG docker $USER

COPY thermos.sh /home/$USER/thermos.sh

RUN chown $USER:$USER /home/$USER/thermos.sh && \
     chmod 500 /home/$USER/thermos.sh
```

```
WORKDIR /home/$USER

ENTRYPOINT ["./thermos.sh"]
```

A.5 thermos.sh startup script

```
#!/bin/bash

if [ -z "$MESOS_PORT" ]; then
        echo "MESOS_PORT must not be empty"
        exit 1
fi

if [ -z "$MESOS_MASTER" ]; then
        echo "MESOS_MASTER must not be empty"
        exit 1
fi

if [ -z "$MESOS_SWITCH_USER" ]; then
        echo "MESOS_SWITCH_USER must not be empty"
        exit 1
fi

if [ -z "$MESOS_CONTAINERIZERS" ]; then
        echo "MESOS_CONTAINERIZERS must not be empty"
        exit 1
fi

if [ -z "$MESOS_LOG_DIR" ]; then
        echo "MESOS_LOG_DIR must not be empty"
        exit 1
fi

if [ -z "$MESOS_WORK_DIR" ]; then
        echo "MESOS_WORK_DIR must not be empty"
        exit 1
fi

if [ -z "$MESOS_ROOT" ]; then
        echo "MESOS_ROOT is not set, will be set to \
        MESOS_WORK_DIR:$MESOS_WORK_DIR"
        MESOS_ROOT=$MESOS_WORK_DIR
elif [ "$MESOS_ROOT" != "$MESOS_WORK_DIR" ]; then
        echo "MESOS_ROOT is set to $MESOS_ROOT, but must match \
        MESOS_WORK_DIR:$MESOS_WORK_DIR"
        exit 1
fi
```

```
chown -R $USER:$USER $MESOS_ROOT $MESOS_WORK_DIR $MESOS_LOG_DIR

sudo -E -u $USER nohup /usr/sbin/thermos_observer \
--port=${HTTP_PORT:-1338} --mesos-root=${MESOS_ROOT} --app_daemonize \
--log_to_disk=NONE --log_to_stderr=google:INFO

sudo -E -u $USER nohup mesos-slave

wait
```

A.6 Aurora CLI Dockerfile

```
FROM ubuntu:latest

ARG AURORA_VERSION=0.17.0
ENV USER=aurora
ARG AURORA_CONFIG_ROOT=/home/$USER/.aurora/clusters.json

RUN apt-get update && \
    apt-get install -y python2.7 wget && \
    wget -c https://apache.bintray.com/aurora/ubuntu-trusty/\
aurora-tools_${AURORA_VERSION}_amd64.deb && \
    dpkg -i aurora-tools_${AURORA_VERSION}_amd64.deb && \
    useradd $USER --create-home --shell /bin/bash

USER aurora

WORKDIR /home/aurora
```

A.7 Aurora CLI sample clusters.json

```
[{
  "auth_mechanism": "UNAUTHENTICATED",
  "name": "aurora",
  "scheduler_zk_path": "/aurora",
  "slave_root": "/var/tmp/mesos",
  "slave_run_directory": "latest",
  "scheduler_uri": "localhost:5555",
  "zk": "localhost:2181"
}]
```

A.8 docker-compose.yml

```
version: '3'
services:

  zookeeper:
    image: ${HOST_A}:5000/zookeeper
    container_name: zookeeper
    network_mode: "host"
    environment:
      ZOO_MY_ID: 1
    volumes:
      - "./zoo.cfg:/conf/zoo.cfg"

  mesos_slave:
    image: ${HOST_A}:5000/aurora-worker:latest
    container_name: aurora_worker
    network_mode: "host"
    volumes:
      - "./log/mesos:/var/log/mesos"
      - "./tmp/mesos:/var/tmp/mesos"
      - "/var/run/docker.sock:/var/run/docker.sock"
      - "/cgroup:/cgroup"
      - "/sys/fs/cgroup:/sys/fs/cgroup"
    env_file:
      - ./env/mesos_slave_env

  mesos_master:
    image: ${HOST_A}:5000/mesos-master:1.1.1
    container_name: mesos_master
    network_mode: "host"
    volumes:
      - "./log/mesos:/var/log/mesos"
      - "./tmp/mesos:/var/tmp/mesos"
    env_file:
      - ./env/mesos_master_env

  marathon:
    image: ${HOST_A}:5000/marathon
    container_name: marathon
    network_mode: "host"

  aurora:
    image: "${HOST_A}:5000/aurora-scheduler:latest"
    container_name: "aurora_scheduler"
    env_file:
      - ./aurora_env
```

Appendix B

s390x Dockerfiles

B.1 Dockerfile for s390x Marathon

```
FROM ibm/mesos-base-s390x

ENV VERSION=1.4.3
ENV USER=marathon
ENV WORKDIR=/marathon \
ENV MESOS_NATIVE_JAVA_LIBRARY=/usr/local/lib/libmesos.so

RUN apt-get install wget

RUN useradd $USER --no-create-home --shell /bin/bash && \
    mkdir -p "$WORKDIR" && chown "$USER:$USER" "$WORKDIR"

WORKDIR "$WORKDIR"
RUN wget "http://downloads.mesosphere.com/marathon/v$VERSION\
/marathon-$VERSION.tgz" && \
    wget "http://downloads.mesosphere.com/marathon/v$VERSION\
/marathon-$VERSION.tgz.sha256"

RUN sha256sum -c marathon-$VERSION.tgz.sha256 && \
    if [ "$?" != 0 ]; then exit 1; fi

RUN tar -xf marathon-$VERSION.tgz && \
    rm marathon-$VERSION.tgz.sha256 marathon-$VERSION.tgz && \
    chown -R "$USER:$USER" marathon-$VERSION

WORKDIR $WORKDIR/marathon-$VERSION
RUN su - $USER

ENTRYPOINT ["./bin/start"]
```

B.2 Dockerfile for s390x Mesos

```
FROM s390x/ubuntu:16.04

RUN apt-get update && apt-get install wget build-essential python-dev \
    python-six python-virtualenv libcurl4-nss-dev libsasl2-dev \
    libsasl2-modules maven libapr1-dev libsvn-dev zlib1g-dev

WORKDIR /tmp

RUN wget https://download.docker.com/linux/static/stable/s390x/\
    docker-17.06.0-ce.tgz && \
    tar -xf docker-17.06.0-ce.tgz && \
    cd docker && cp docker /usr/bin/

ADD mesos-1.1.1_s390x.deb .
RUN dpkg -i mesos-1.1.1_s390x.deb

RUN rm -rf /tmp/*
```

B.3 Dockerfile for s390x Mesos master

```
FROM ibm/mesos-base-s390x

CMD ["--registry=in_memory]
ENTRYPOINT ["mesos-master"]
```

B.4 Dockerfile for s390x Mesos agent

```
FROM ibm/mesos-base-s390x

CMD ["--registry=in_memory]
ENTRYPOINT ["mesos-slave"]
```

Appendix C

Mesos build files

C.1 Patch file for IBM Java SDK

```
diff --git a/configure.ac b/configure.ac
index 6421ec6..2ad5493 100644
--- a/configure.ac
+++ b/configure.ac
@@ -891,6 +891,9 @@ __EOF__
fi
fi

+  # Determine if current JDK is IBMJDK or not.
+  test -f $JAVA_HOME/include/jniport.h; IS_IBMJDK=$?
+
# Determine linker flags for Java if not set.
if test "$OS_NAME" = "darwin"; then
dir="$JAVA_HOME/jre/lib/server"
@@ -898,10 +901,17 @@ __EOF__
JAVA_JVM_LIBRARY=$dir/libjvm.dylib
elif test "$OS_NAME" = "linux"; then
for arch in amd64 i386 arm aarch64 ppc64 ppc64le s390 s390x; do
-      dir="$JAVA_HOME/jre/lib/$arch/server"
+      if test "$IS_IBMJDK" = "1"; then
+        dir="$JAVA_HOME/jre/lib/$arch/server"
+      else
+        dir="$JAVA_HOME/jre/lib/$arch/default"
+      fi
if test -e "$dir"; then
# Note that these are libtool specific flags.
JAVA_TEST_LDFLAGS="-L$dir -R$dir -Wl,-ljvm"
+        if test "$IS_IBMJDK" = "0"; then
+          JAVA_TEST_LDFLAGS="$JAVA_TEST_LDFLAGS -ldl"
+        fi
JAVA_JVM_LIBRARY=$dir/libjvm.so
break;
```

```
fi
```

C.2 Mesos DEB build control file

```
Package: mesos-1.1.1
Architecture: s390x
Maintainer: Patrick.Kleindienst
Priority: optional
Version: 0.1
Depends: build-essential, python-dev, python-six, python-virtualenv, maven
libcurl4-nss-dev, libsasl2-dev, libsasl2-modules, libapr1-dev, libsvn-dev
Description: Apache Mesos for s390x
```

Bibliography

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java programming language*. Addison Wesley Professional, 2005.

[2] U. Bacher. Docker CE for all distributions. June 2017. URL: http://containerz.blogspot.com/2017/06/docker-ce-for-all-distributions.html (visited on 08/15/2017).

[3] U. Bacher. First CE for s390x by Docker. June 2017. URL: http://containerz.blogspot.com/2017/06/first-ce-for-s390x-by-docker.html (visited on 08/15/2017).

[4] U. Bacher. New Docker Engine in SLES 12 Containers Module. July 2017. URL: http://containerz.blogspot.com/2017/07/new-docker-engine-in-sles-12-containers.html (visited on 08/20/2017).

[5] L. A. Barroso, J. Clidaras, and U. Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Morgan & Claypool Publishers, San Rafael, Calif., Aug. 2013.

[6] N. Brown. The Overlay Filesystem. May 2015. URL: http://windsock.io/the-overlay-filesystem/.

[7] M. Brož. Device mapper (kernel part of LVM2 volume management). URL: https://mbroz.fedorapeople.org/talks/DeviceMapperBasics/dm.pdf (visited on 08/27/2017).

[8] M. Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 2006.

[9] J. Calcote. *Autotools: A Practitioner's Guide to GNU Autoconf, Automake, and Libtool*. No Starch Press, 2010.

[10] Canonical Ltd. Ubuntu – Package Search Results – docker.io. 2017. URL: https://packages.ubuntu.com/search?suite=all&arch=s390x&searchon=names&keywords=docker.io (visited on 08/15/2017).

[11] D. Cole. Data center infrastructure management. 2012. URL: http://iisgroupllc.com/wp-content/uploads/2013/02/Data-Center-Knowlede-DCIM-Guide.pdf (visited on 10/10/2017).

[12] M. E. Conway. Conway's Law. 2017. URL: https://www.melconway.com/Home/Conways_Law.html (visited on 07/28/2017).

[13] M. E. Conway. How do committees invent. *Datamation*, 14(4):28–31, 1968. (Visited on 08/22/2017).

[14] Docker Inc. About images, containers, and storage drivers. 2017. URL: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/#images-and-layers (visited on 08/30/2017).

[15] Docker Inc. About Registry. 2017. URL: https://docs.docker.com/registry/introduction/ (visited on 08/29/2017).

[16] Docker Inc. Deploy a registry server. 2017. URL: https://docs.docker.com/registry/deploying/ (visited on 09/21/2017).

[17] Docker Inc. Docker. 2017. URL: https://www.docker.com/ (visited on 09/11/2017).

[18] Docker Inc. Docker tag | Docker Documentation. 2017. URL: https://docs.docker.com/engine/reference/commandline/tag/ (visited on 08/28/2017).

[19] Docker Inc. Get Docker CE for CentOS. Sept. 2017. URL: https://docs.docker.com/engine/installation/linux/docker-ce/centos/ (visited on 09/22/2017).

[20] Docker Inc. Get Docker CE for Ubuntu. Aug. 2017. URL: https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/ (visited on 08/16/2017).

[21] Docker Inc. HTTP API V2. 2017. URL: https://docs.docker.com/registry/spec/api/ (visited on 08/30/2017).

[22] Docker Inc. Image Manifest V 2, Schema 2. 2017. URL: https://docs.docker.com/registry/spec/manifest-v2-2/ (visited on 08/30/2017).

[23] Docker Inc. Library/alpine - Docker Hub. 2017. URL: https://hub.docker.com/_/alpine/ (visited on 10/15/2017).

[24] Docker Inc. Overview of Docker Compose. 2017. URL: https://docs.docker.com/compose/overview/ (visited on 09/26/2017).

[25] Docker Inc. Plugins API. Sept. 2017. URL: https://docs.docker.com/engine/extend/plugin_api/ (visited on 05/09/2017).

[26] Docker Inc. Select a storage driver. Aug. 2017. URL: https://docs.docker.com/engine/userguide/storagedriver/selectadriver/.

[27] Docker Inc. Swarm mode overview. 2017. URL: https://docs.docker.com/engine/swarm/ (visited on 03/08/2017).

[28] Docker Inc. Test an insecure registry. 2017. URL: https://docs.docker.com/registry/insecure/ (visited on 09/21/2017).

[29] Docker Inc. Understand the Notary service architecture. Oct. 2017. URL: https://docs.docker.com/notary/service_architecture/ (visited on 10/09/2017).

[30] Docker Inc. Use the AUFS storage driver. 2017. URL: https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/ (visited on 08/16/2017).

[31] Docker Inc. Use the BTRFS storage driver. Aug. 2017. URL: https://docs.docker.com/engine/userguide/storagedriver/btrfs-driver/ (visited on 08/28/2017).

[32] Docker Inc. Use the Device Mapper storage driver. Aug. 2017. URL: https://docs.docker.com/engine/userguide/storagedriver/device-mapper-driver/ (visited on 08/27/2017).

[33] P. Estes. A big step towards multi-platform Docker images. Apr. 2016. URL: https://integratedcode.us/2016/04/22/a-step-towards-multi-platform-docker-images/ (visited on 08/30/2017).

[34] P. Estes. Storage Drivers in Docker: A Deep Dive. Aug. 2016. URL: https://integratedcode.us/2016/08/30/storage-drivers-in-docker-a-deep-dive/ (visited on 09/21/2017).

[35] M. Fowler. Microservice Trade-Offs. Jan. 2015. URL: https://martinfowler.com/articles/microservice-trade-offs.html (visited on 07/21/2017).

[36] M. Fowler and J. Lewis. Microservices. Mar. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 07/17/2017).

[37] J. Grundy. Firmware Validation: Challenges & Opportunities. 2013. URL: http://memocode.irisa.fr/2013/Final/Tutorial-3-GrundyMelham-FirmwareValidation.pdf (visited on 10/04/2017).

[38] M. Herlihy. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

[39] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 2011.

[40] D. Holth. Wheel — wheel 0.29.0 documentation. 2012. URL: https://wheel.readthedocs.io/en/latest/ (visited on 08/21/2017).

[41] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference*

on USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA, 2010.

[42] S. Hykes. Introducing runC: a lightweight universal container runtime. June 2015. URL: https://blog.docker.com/2015/06/runc/.

[43] R. Ignazio. *Mesos in Action*. Manning Publications, Shelter Island, NY, May 2016.

[44] M. Jones, S. Piersall, T. B. Mathias, J. Gay, D. Herrington, F. Schumacher, J. Eggleston, E. Berman, T. Simkulet, R. Planutis, B. Valentine, F. Heaney, E. Weinman, J. Miller, D. Simpson, P. Kirkaldy, B. Tolan, M. Clark, B. Boisvert, C. Smith, and B. Ogden. *Introduction to the System z Hardware Management Console*. Of *Redbooks*. Endicott, NY, Feb. 2010.

[45] Linux on z Systems Open Source team. Building Apache Mesos. Aug. 2017. URL: https://github.com/linux-on-ibm-z/docs (visited on 08/25/2017).

[46] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux j.*, 2014(239), Mar. 2014.

[47] Mesosphere. Docker-containers: Dockerfiles and assets for building Docker containers. Oct. 2016. URL: https://github.com/mesosphere/docker-containers (visited on 09/22/2017).

[48] Mesosphere. Marathon Documentation. 2016. URL: https://mesosphere.github.io/marathon/docs/ (visited on 09/06/2017).

[49] Mesosphere, Inc. DC/OS Documentation. 2017. URL: https://dcos.io/docs/1.10/ (visited on 09/17/2017).

[50] Mesosphere, Inc. Marathon: A container orchestration platform for Mesos and DC/OS. 2016. URL: https://mesosphere.github.io/marathon/ (visited on 10/18/2017).

[51] Mesosphere, Inc. Marathon: Application Deployments. 2016. URL: https://mesosphere.github.io/marathon/docs/deployments.html (visited on 10/13/2017).

[52] Mesosphere, Inc. Marathon: Constraints. 2016. URL: https://mesosphere.github.io/marathon/docs/constraints.html (visited on 10/08/2017).

[53] Mesosphere, Inc. Marathon: Deploy and manage containers (including Docker) on top of Apache Mesos at scale. 2017. URL: https://github.com/mesosphere/marathon (visited on 10/03/2017).

[54] Mesosphere, Inc. Marathon: Stateful Applications Using Local Persistent Volumes. 2016. URL: https://mesosphere.github.io/marathon/docs/persistent-volumes.html (visited on 10/18/2017).

[55] Mesosphere, Inc. Mesos-DNS: Installing and running Mesos-DNS. 2015. URL: https://mesosphere.github.io/mesos-dns/docs/ (visited on 09/09/2017).

[56] Mesosphere, Inc. Mesos-DNS: Installing and running Mesos-DNS. 2015. URL: https://mesosphere.github.io/mesos-dns/docs/ (visited on 09/24/2017).

[57] A. Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Dec. 2015.

[58] C. Negus and F. Caen. *Ubuntu Linux Toolbox: 1000+ Commands for Ubuntu and Debian Power Users*. John Wiley & Sons, 2008.

[59] S. Newman. *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., Feb. 2015.

[60] B. Nguyen. /proc. July 2004. URL: http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/proc.html (visited on 08/17/2017).

[61] C. O'Hanlon. A Conversation with Werner Vogels. *Queue*, 4(4), May 2006.

[62] J. R. Okajima. Aufs4 – advanced multi layered unification filesystem version 4.x. 2017. URL: http://aufs.sourceforge.net/ (visited on 09/21/2017).

[63] J. Petazzoni. Docker storage drivers. Technology. Mar. 2015. URL: https://www.slideshare.net/Docker/docker-storage-drivers (visited on 08/17/2017).

[64] Prometheus Authors. Prometheus - Monitoring system & time series database. 2017. URL: https://prometheus.io/ (visited on 10/01/2017).

[65] Red Hat, Inc. Chapter 39. File Systems. 2017. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/7.3_Release_Notes/technology_previews_file_systems.html (visited on 08/18/2017).

[66] Red Hat Inc. Red Hat Enterprise Linux Release Dates - Red Hat Customer Portal. 2017. URL: https://access.redhat.com/articles/3078 (visited on 08/18/2017).

[67] J. H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the acm*, 1974.

[68] V. Saraswat. Announcing the New Release of Docker Enterprise Edition. Aug. 2017. URL: https://blog.docker.com/2017/08/docker-enterprise-edition-17-06/ (visited on 08/20/2017).

[69] A. Shraer, B. Reed, D. Malkhi, and F. Junqueira. Dynamic Reconfiguration of Primary/Backup Clusters. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 2012.

[70] SUSE. Docker Guide | SUSE Linux Enterprise Server 12 SP2. 2017. URL: https://www.suse.com/documentation/sles-12/singlehtml/book_sles_docker/book_sles_docker.html (visited on 08/20/2017).

[71] SUSE. Modules - SUSE Linux Enterprise Server. 2017. URL: https://www.suse.com/products/server/features/modules/ (visited on 08/15/2017).

[72] The Apache Software Foundation. Apache Aurora. 2017. URL: https://aurora.apache.org/ (visited on 10/12/2017).

[73] The Apache Software Foundation. Apache Aurora: Aurora System Overview. 2017. URL: https://aurora.apache.org/documentation/latest/getting-started/overview/ (visited on 10/18/2017).

[74] The Apache Software Foundation. Apache Aurora Documentation. 2017. URL: https://aurora.apache.org/documentation/latest/ (visited on 09/06/2017).

[75] The Apache Software Foundation. Apache Aurora: Scheduler Configuration. 2017. URL: https://aurora.apache.org/documentation/latest/operations/configuration/ (visited on 10/18/2017).

[76] The Apache Software Foundation. Apache Aurora: Scheduling Constraints. 2017. URL: https://aurora.apache.org/documentation/latest/features/constraints/ (visited on 10/14/2017).

[77] The Apache Software Foundation. Apache Aurora: Service discovery. 2017. URL: https://aurora.apache.org/documentation/latest/features/service-discovery/ (visited on 10/18/2017).

[78] The Apache Software Foundation. Apache Mesos. 2017. URL: http://mesos.apache.org/ (visited on 10/10/2017).

[79] The Apache Software Foundation. Apache Mesos - Attributes and Resources. 2017. URL: https://mesos.apache.org/documentation/latest/attributes-resources/ (visited on 10/14/2017).

[80] The Apache Software Foundation. Apache Mesos: Agent Recovery. 2017. URL: https://mesos.apache.org/documentation/latest/agent-recovery/.

[81] The Apache Software Foundation. Apache Mesos Documentation. 2017. URL: http://mesos.apache.org/documentation/latest/ (visited on 08/23/2017).

[82] The Apache Software Foundation. Apache Mesos: Getting Started. 2017. URL: https://mesos.apache.org/gettingstarted/ (visited on 10/04/2017).

[83] The Apache Software Foundation. Apache Mesos: Persistent Volumes. 2017. URL: https://mesos.apache.org/documentation/latest/persistent-volume/.

[84] The Apache Software Foundation. Apache Mesos: Scheduler HTTP API. 2017. URL: https://mesos.apache.org/documentation/latest/scheduler-http-api/ (visited on 10/15/2017).

[85] The Apache Software Foundation. Apache Mesos: Shared Persistent Volumes. 2017. URL: https://mesos.apache.org/documentation/latest/shared-resources/.

[86] The Apache Software Foundation. Client Cluster Configuration. 2017. URL: https://aurora.apache.org/documentation/latest/reference/client-cluster-configuration/ (visited on 09/25/2017).

[87] The Apache Software Foundation. Installing Aurora. 2017. URL: https://aurora.apache.org/documentation/latest/operations/installation/ (visited on 09/25/2017).

[88] The Apache Software Foundation. Mesos Configuration. 2017. URL: https://mesos.apache.org/documentation/latest/configuration/ (visited on 09/24/2017).

[89] The Apache Software Foundation. Mesos: Containerizers. 2017. URL: http://mesos.apache.org/ (visited on 10/12/2017).

[90] The Apache Software Foundation. Task Health Checking and Generalized Checks. 2017. URL: https://mesos.apache.org/documentation/latest/health-checks/ (visited on 09/29/2017).

[91] The Apache Software Foundation. The Mesos Replicated Log. 2017. URL: https://mesos.apache.org/documentation/latest/replicated-log-internals/ (visited on 10/13/2017).

[92] The Apache Software Foundation. ZooKeeper: Because Coordinating Distributed Systems is a Zoo. Aug. 2014. URL: https://zookeeper.apache.org/doc/trunk/index.html (visited on 09/09/2017).

[93] The Apache Software Foundation. ZooKeeper Dynamic Reconfiguration. Nov. 2014. URL: https://zookeeper.apache.org/doc/trunk/zookeeperReconfig.html (visited on 09/22/2017).

[94] The Apache Software Foundation. ZooKeeper Recipes and Solutions. July 2017. URL: https://zookeeper.apache.org/doc/trunk/recipes.html (visited on 10/13/2017).

[95] The CentOS Project. About CentOS. 2017. URL: https://www.centos.org/about/ (visited on 09/22/2017).

[96] The ELRepo Project. ELRepo : HomePage. Nov. 2017. URL: https://elrepo.org/tiki/tiki-index.php (visited on 08/18/2017).

[97] The freedesktop.org contributors. Systemd. Mar. 2017. URL: https://www.freedesktop.org/wiki/Software/systemd/ (visited on 08/17/2017).

[98] The Kubernetes Authors. Extensions API Definitions. 2017. URL: https://kubernetes.io/docs/api-reference/extensions/v1beta1/definitions/ (visited on 09/05/2017).

[99] The Kubernetes Authors. Kubernetes. 2017. URL: https://kubernetes.io/ (visited on 03/08/2017).

[100] The Kubernetes Authors. Kubernetes Documentation. 2017. URL: https://kubernetes.io/docs/home/.

[101] The Pants Community. Pants: A fast, scalable build system. 2017. URL: https://www.pantsbuild.org/ (visited on 10/14/2017).

[102] B. Wootton. Microservices - Not a free lunch! - High Scalability -. Apr. 2014. URL: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html (visited on 07/27/2017).