DIONYSIS LAPPAS

DEVELOPMENT OF A CONTAINER MANAGEMENT TOOL FOR WEB APPLICATIONS USING DOCKER

DEVELOPMENT OF A CONTAINER MANAGEMENT TOOL FOR WEB APPLICATIONS USING DOCKER

DIONYSIS LAPPAS
SUPERVISOR: VASSILIS SAMOLADAS

COMMITTEE:
VASSILIS SAMOLADAS
MINOS GAROFALAKIS
ANTONIOS DELIGIANNAKIS

Submitted to the Department of Electrical and Computer Engineering in partial fulfilment of the requirements for the degree of Diploma in Engineering

Technical University of Crete

Chania July 2016 –

Dedicated to my family.
My father Petros, my mother Amalia and my sister Constantina.

ABSTRACT

In recent years, due to the rise of microservices, cloud computing and now the Internet of Things (IoT) the development, deployment and management of distributed services is more important than ever. In order to circumvent the challenges that arise from this evolution, we need tools that, among others, abstract the inherent complexities, manage dependencies, maximize portability across systems and enhance scalability. Software containers encapsulate many of the aforementioned features. The Docker project, started in 2013, has enabled users to easily build, ship and run applications based on containers, through an automated workflow. Running and managing multi-container web applications in the cluster infrastructure though, is not a trivial task, which requires to deal with orchestration, service discovery, data and configuration management, networking e.t.c.

We designed and implemented a container management tool for multi-container web applications, on single and multi-host (cluster) environments for development and production. We consider containers as components offering and requiring services, empowering a container-agnostic design. We offer a standard way to map services for web applications into containers, complete life-cycle management for containers and provided services, coordination between dependent services, data management, service discovery, network isolation with per-application custom networks, support for private container image registries and a solution to N-to-N configuration problem between containers.

Creativity can be a social contribution,
but only in so far as society is free to use the results.

— Richard Stallman

ACKNOWLEDGMENTS

My journey in science and engineering would not have started without the unconditional support of my family. I would like to thank my parents Petros and Amalia whose love has always been the constant good in my life and whose encouragement has helped me fulfill my dreams. I cannot thank enough my beloved sister, Constantina, who has been a beacon of light in my life, and through her love and guidance I was shaped into the person I am today.

Many thanks to my supervisor Vasilis Samoladas, who was an inspiration to me throughout my under-graduate course and helped me realize the fine connection between simplicity in science and greatness in mind. Moreover, I want to thank him for trusting me with this Thesis and helping me throughout with the difficulties encountered. I would also like to thank the members of my committee Professor Minos Garofalakis and Associate Professor Antonios Deligiannakis.

Of course, I saved the best for the end, I want to express my deepest gratitude towards my friends Babis, Manos, Anna-Maria, Panos, Theo who have been by my side all those years and through the endless conversations and the moments we shared, good and bad, helped me become a better man and realize that real friends are invaluable. Thank you all! One person stands out, whose free spirit, good-heartedness and constant support had a great impact on my life. Thank you Babis!

Finally, I would like to acknowledge Babis and Constantina for reading this Thesis and helping me with corrections and Theo for his valuable help with images.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

IoT	Internet of Things
LXC	Linux Containers
IP	Internet Protocol
OS	Operating System
DAG	Directed Acyclic Graph
POJO	Plain Old Java Object
ORM	Object Relational Mapping
JPQL	Java Persistence Query Language
JNDI	Java Naming and Directory Interface
CRUD	Create/Read/Update/Delete
API	Application Programming Interface
JAXB	Java Architecture for Xml Binding
XML	Extensible Markup Language
XSD	XML Schema Definitions

Part I

INTRODUCTION AND BACKGROUND

1 INTRODUCTION

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

— Leslie Lamport

Distributed systems advancements have nurtured the development of new and exciting technologies. Decentralization of services, resources and user access have expedited the development of distributed applications. Towards that end, new software architectural patterns have emerged such as microservices. Cloud computing, a new computing model for enabling ubiquitous, on-demand network access to a shared pool of configurable computing resources, along with the advent of the Internet of Things (IoT) where items in the physical world and sensors within or attached to these items, are connected to the Internet, show the imperative need for distributed services.

Many challenges arise when developing, deploying and managing distributed applications. The most common problems faced, usually concern the overwhelming system complexities, service security, service discovery, resolving and managing dependencies, resource manipulation and control, reliable fail-over mechanisms guaranteeing overall stability, portability across heterogeneous environments and the ability to scale. Those problems are difficult in themselves. In order to address them effectively, a new approach is required.

1.1 CONTAINER TECHNOLOGY

Containers offer the necessary abstractions to overcome the predefined issues. They provide lightweight operating-system-level virtualization that lets us isolate processes and resources without the need to provide instruction interpretation mechanisms and other complexities of full virtualization.[6] As a result, we have an ideal environment for service deployment in terms of speed, isolation and life-cycle management.

This is not a new concept. UNIX systems have had for decades a simple form of filesystem isolation, the chroot command. Since 1998, FreeBSD has had the jail utility, which extended chroot

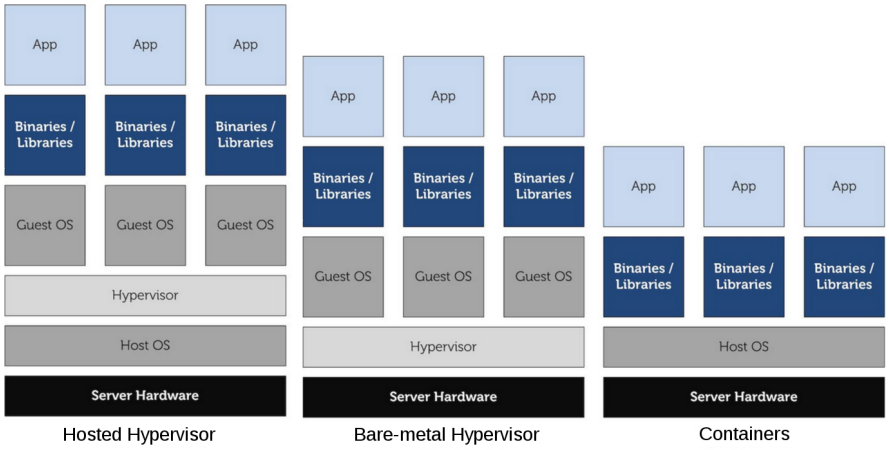

Figure 1: *A comparative view of containers and Hypervisors.* Virtualization with hypervisors adds extra overhead which -in best case, is a guest OS. Containers on the other hand, share the same kernel with the underlying OS, adding no overhead in the infrastructure.

sandboxing to processes. This Jail had access to the operating system kernel but other than that it could only get to a very limited set of other system resources. For example, a FreeBSD jail typically only has access to a preassigned Internet Protocol Internet Protocol (IP) network address. Later in 2001, Solaris Zones offered a containerization technology limited to the Solaris OS while in 2005 OpenVZ, a container technology for Linux, was open-sourced, but required a patched kernel. While each of these technologies has matured, these solutions have not made significant strides towards integrating their container support into the mainstream Linux kernel.

LXC or Linux Containers.

LINUX CONTAINERS Over time, all of these efforts have consolidated into the Linux Containers (LXC) project, started in 2008. With LXC, applications can run in their own container. Each container shares a common Linux system kernel, but unlike a Hypervisor there is no attempt made to abstract the hardware (Hosted Hypervisor) nor there is any requirement for a guest OS (Hosted and Bare-metal Hypervisor). Several components are needed for Linux Containers to function correctly, most of them are provided by the Linux kernel:

- namespaces (ipc, uts, mount, pid, network and user) to ensure process isolation. A namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their

own isolated instance of the global resource. Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes [10].

- cgroups to control the system resources. CGroups allow you to allocate resources - such as CPU time, system memory, network bandwidth, or combinations of these resources - among user-defined groups of tasks (processes) running on a system [15].

- chroots (using pivot_root) to control the location of the file system root.

- capabilities to allocate privileges to a process for more fine-grained security control. On UNIX family operating systems, the traditional privilege separation scheme divides processes into two categories:

  1. those whose effective user ID is 0 (root), which bypass all privilege checks.
  2. all other processes which are subject to privilege checking according to their user and group IDs.

  With Linux capability scheme the handling of this problem if refined. Instead of using a single privilege (i.e., effective user ID of 0 for security checks in the kernel, the root privilege is divided into distinct units, called capabilities [8].

- seccomp policies to filter system calls. A secomb policy will allow only pre-defined system calls for a process and reject all others.

- appArmor and SELinux profiles to assure separation between the host and the container and also between the individual containers[2].

Using containers has been a best practice for a long time. But manually building containers can be challenging and easy to do incorrectly.

DOCKER In 2013, the Docker project, offered a new automated workflow for containers along with new functionality that finally made container technology mainstream.

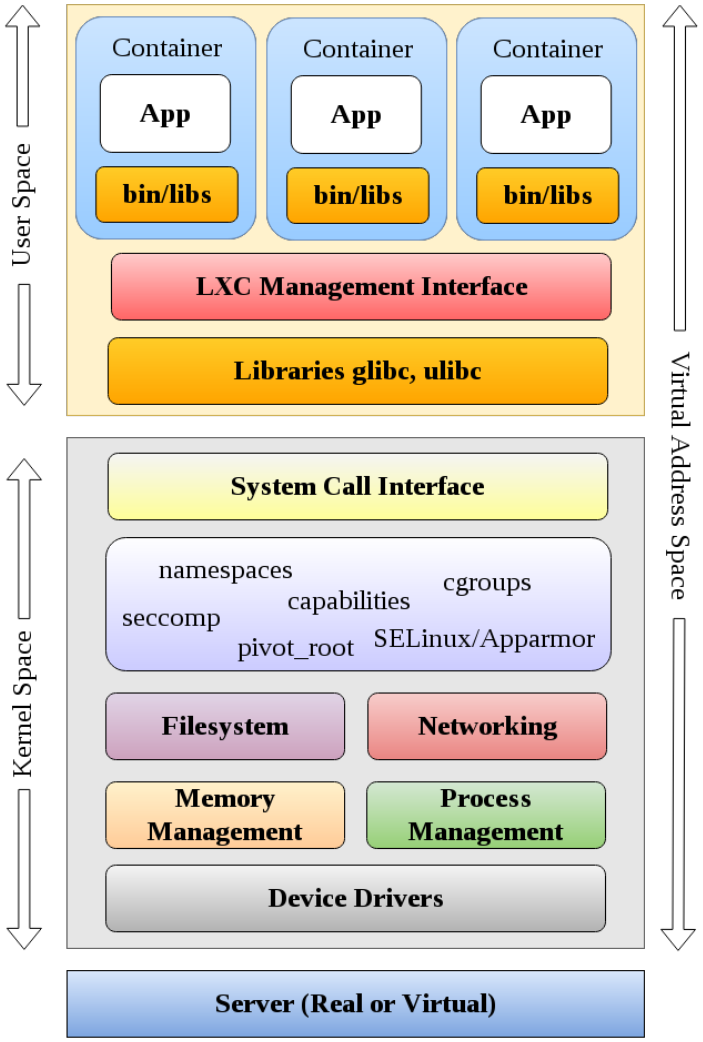

Figure 2: LXC Container view and user-kernel address space

1.1.1 *Containers vs Hypervisors*

A hypervisor is a piece of software in charge of[17] :

- assigning resources from one or several physical machines to virtual machines.
- managing the virtual machines.

There are generally two types of Hypervisors:

1. *Type 1*: Native or bare-metal hypervisor runs on the bare metal of the hardware and creates a virtualization layer below the OS layer.
2. *Type 2*: Hosted hypervisor runs an application of the host OS and creates a virtualization layer above the OS layer.

	CONTAINERS	HYPERVISORS
Scope	application scoped	application and operating-system scoped
Start/Stop time	very fast (ms)	normal (min)
Overhead	minimal from container engine/tools	big from guest OS
Isolation	strong but still under heavy development	best offered
Adoption	small has not yet matured	proven in production

Table 1: Compariston of Containers and bare-metal Hypervisors

1.1.2 *Docker*

Docker is an open source project that revolutionized the containerization process for applications. Using Docker any user can build, ship and run applications in software containers. Essentially, Docker allows developers to package an application with all of its dependencies into a standardized unit. The portability and isolation guarantees of containers ease collaboration with other developers and operations. As a result, developers do not have to worry for any inconsistencies when running their code across heterogeneous environments and operations can focus on hosting and orchestrating containers rather than worrying about the code running inside them.

DOCKER COMPONENTS The Docker platform has two distinct components:

- Docker Engine: the open source containerization platform which is responsible for creating and running containers.
- Docker Hub: a Software-as-a-Service platform for sharing and managing Docker containers. It provides a centralized resource for container image discovery and distribution.

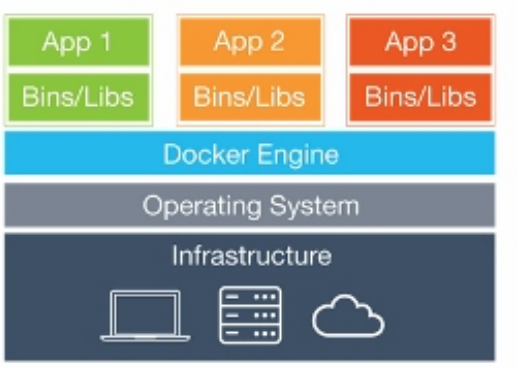

Figure 3: Server with Docker

DOCKER ARCHITECTURE Docker architecture is based on the client-server model. A daemon runs in the background accepting requests from local and/or remote clients, performing all the necessary tasks to build images, run and distribute containers.

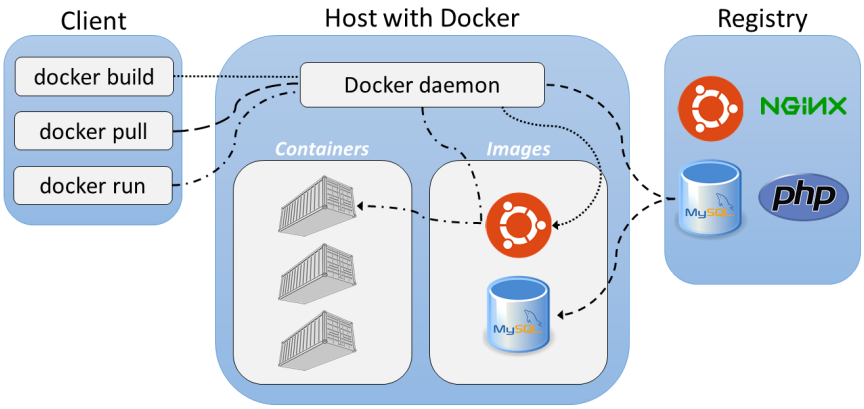

Figure 4: Docker Architecture

images

DOCKER IMAGES Docker containers are based on read-only image templates, that are build in a layered manner. By taking advantage of union file systems, the layers are combined into a single image instance forming a file system. When an image gets updated, a new layer is built on top of the others. As a result, there is no need to rebuild the whole image and only the added layer is distributed as an update. Tracking changes to image layers is accomplished with built-in Git support. New images may be build from base images or from scratch.

Supporting a portable image format, Docker allowed for application to be packaged with all their dependencies, libraries and files they needed to run inside a container. The underlying image file system is read-only. When a new container is started from an image, a read-write layer is added on top of the image. As a result, multiple containers can be spawned from the same image without sharing changes to their file systems.

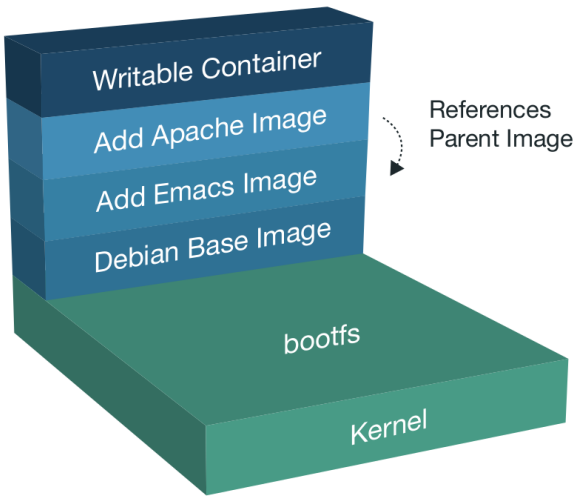

Figure 5: Docker image layered structure

DOCKER CONTAINERS A Docker container is built from an image. The image holds all the necessary data for a process to run in the container. As discussed earlier, when the container starts a new read-write layer is added on top of the base-image layers.

The supported functionality of Docker containers, is empowered by the following technologies [14][13]:

1. Linux Kernel feutures:
   - *namespaces*
     - PID namespace - Process identifiers and capabilities.
     - UTS namespace - Host and domain name.
     - MNT namespace - File system access and structure.
     - IPC namespace - Process communication over shared memory.
     - NET namespace - Network access and structure.

    - USR namespace - User names and identifiers.
  - *chroot()* - Controls the location of the file system root.
  - *cgroups* - Resource protection.

2. Union file systems (UnionFS). They allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system.

3. Container format or the execution interface. The default execution interface since v0.9 is LibContainer. It is meant to be a cross-system abstraction layer as an attempt to standardize the way applications are packaged, delivered, and run in isolation. This way, container features available in Linux kernel API are provided as a unique library in a consistent way. LibContainer addresses the problem of having an unique kernel API and several implementations. (LXC, libvirt, lmctfy e.t.c.)

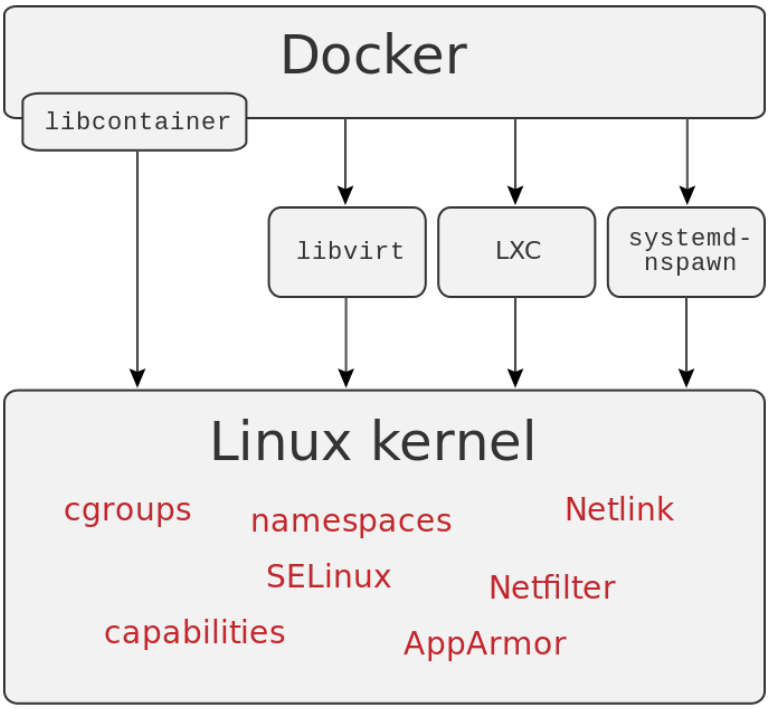

Figure 6: Docker and the different execution interfaces used to access Linux Kernel technologies.

DOCKER CONTAINERS VS LXC Docker used LXC as the default execution interface right from the start, taking advantage of all the low-level capabilities offered. On top of that, it offered a high-level tool to facilitate the containerization of software. As the technology matured, shifted to a new execution interface, named libContainer that used linux kernel technologies like namespaces and cgroups directly. Support for LXC was not removed (until the time of writing).

DOCKER VOLUMES *Docker Volumes* offer a means to manage data in and between containers. When a container starts, Docker adds a read-write layer on top of the read-only layers of the base image. If a file is modified, it is copied out of the read-only layer into the read-write layer where the changes are applied. When the container is deleted, all data lying on the writable container layer are lost (except committed to a new image). In order to be able to persist and share data between containers, Docker introduced the concept of volumes. *volumes*

A volume is a directory/file outside of the default Union File System that exists as normal directory/file on the host file system.

DOCKER NETWORKS Docker creates on the host, by default, three networks which may be used to connect containers: *networks*

- none. The none network adds a container to a container-specific network stack. That container lacks a network interface.
- host. The host network adds a container on the hosts network stack.
- bridge. The bridge network called docker0 is the default network joined by containers, except specified otherwise. This is a virtual Ethernet bridge attached to the host[14].

A user may also create a user-defined network to connect the deployed containers, using:

- a Docker network driver. Docker provides the following network drivers:
  1. bridge
  2. overlay (for cluster environments)
- a plugin network driver.

Finally, another option to network a container is to use the network stack of another container.

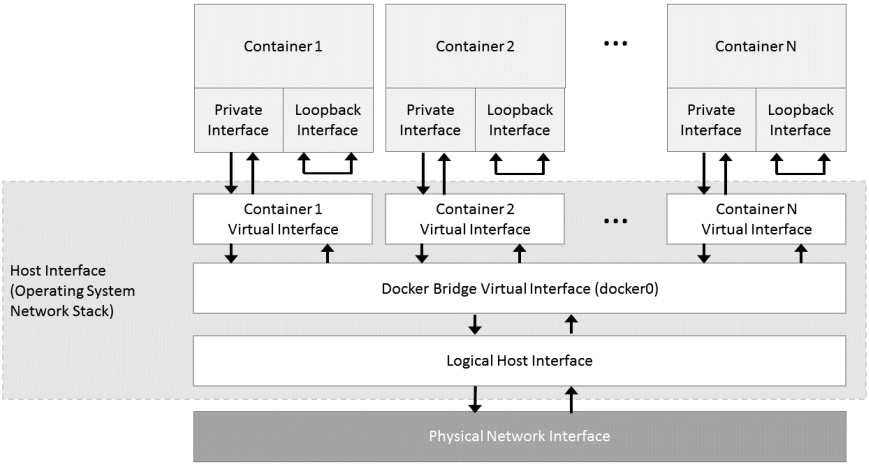

Figure 7: The default Docker bridge network topology with connected containers.

1.2 DEFINITION OF THE PROBLEM

Software containerization is greatly simplified with the tools provided by the Docker platform. Packaging an application into a container and then running it on a provisioned environment is a simple task.

problem definition

THE PROBLEM: *Let a multi-host environment consisted of* N *nodes. Let a distributed application consisted of* $S = s_1, s_2, \ldots, s_n$ *services and packaged in* $|S|$ *containers. The problem we are facing is how to deploy and manage the application that is composed of many containers to the multi-host environment.* The main reason why this is a difficult problem is because the available middleware for distributed services does not offer support for containers yet. In the context of distributed applications, the middleware offers general services that support distributed execution of applications [16]. We are going to analyze the multiple aspects of the stated problem.

SERVICE DISCOVERY Service discovery is a very important component for distributed systems, service oriented architectures and microservices. It involves a directory of services, registering services in that directory and finally being able to lookup

and connect to services from that directory. How do we accomplish that with containers? There are many good middleware solutions to support a directory of services like Zookeeper -a consistent and highly available key-value store, Consul, Etcd e.t.c. However, this is not enough to support service discovery. What we are missing is essentially the mechanism to register containers as services, monitor container health and finally lookup and connect to containers offering services.

CONTAINER LIFE-CYCLE Containerized distributed applications consist of many containers. As a result, we are faced with the problem of manually managing the life-cycle of all the containers, which can be quite challenging. In order to manage the life-cycle of containers in an automated way, two things are necessary:

- a way to define a description of how containers are created, started, stopped and deleted. The create stage includes all the characteristics the container must have. The start and stop stages contain the processes that must be executed in each stage, to control the provided service. At last, the delete stage may specify to delete or not the created resources.
- an implementation of an execution mechanism for the previously defined stages, to support operations in the deployed environment.

CONTAINER CONFIGURATION Every container has to be initialized with some configuration in order to run a service. Usually, services are codependent and require configuration settings from dependent services. If we have N containers that require $N-1$ services, then the magnitude of required configuration settings is proportional to N^2. In that case, how do we configure containers and how do we re-configure them? This is a special case of the well known $N-to-N$ problem for container configuration.

SERVICE COORDINATION When there are dependencies between services of an application, it is often necessary to apply coordination between the dependent services. Required services must be started first and services depending on them must wait for them to start and initialize, before performing any operations. If this requirement is not satisfied, then the ap-

plication will crash or be in an unpredictable and/or inconsistent state.

CONTAINER NETWORKING The underlying network topology for the deployed containers of an application determines the networking isolation of the containers, along with their ability to communicate. If the containers are deployed on a custom virtual private network, an extra layer of security is added. In multi-host environments where every container may be deployed in a different host, it is essential to get container networking right.

CONTAINER DATA Many services usually create data that need to be persisted and then shared with other components. What is more, in development it is often necessary to transfer data to containers, e.g. source code and reflect to applied changes. Such mechanisms are provided by Docker, but we need to automate the process for all the containers of a multi-container application.

There are tools that handle some aspects of the problem very well and other that are more generally implemented to support mostly the concept of environments (development, test, e.t.c.) but not services for distributed applications.

1.3 OUR APPROACH

From the previous analysis, we conclude that in order to address the stated problem and its versatility, new tools are necessary that will encapsulate the required capabilities while utilizing the available technologies from existing middleware.

Fidelio, a container management tool for Web apps, using Docker.

In this Thesis, we designed and implemented *Fidelio*[1]*, a container management tool for Web applications in local and cluster environments, utilizing Docker as the underlying container platform*. Our tool transparently integrates with Apache Zookeeper, a highly available coordination service[7][20], that is used to implement

1 Fidelio is an opera written by Ludwig van Beethoven, his only opera. Opera (the Latin plural for opus, meaning "work") can involve many different art forms (singing, acting, orchestral playing, scenic artistry, costume design, lighting and dance). Accordingly, a distributed containerized application involves many containers/services and Fidelio tries to manage all those components to support the application's execution.

service discovery, store configuration, enable re-configuration and synchronize services for applications.

In our approach, we implement a *container-agnostic design*, in order to decouple containers from their dependencies. We consider *containers as components that offer or require services*. Configuration of components is abstract and not bind to specific instances. As a result, a component may be substituted by another without having to reconfigure all its dependencies. That way, we can manage components as a pool of dynamically changing resources. Automatic re-configuration is supported when the application is restarted.

We offer complete life-cycle control for containers. An advanced process execution mechanism is provided, so that the user has full control over the execution of start and stop sequences for the deployed services.

An application is deployed with respect to service dependencies. Declared service dependencies must form a Directed Acyclic Graph (DAG) for that matter. Service discovery is provided along with synchronization using Zookeeper. Services with no dependencies start first. Dependent services wait for the required services to start and initialize. In shutdown, services stop in reverse order to guarantee graceful application shutdown and data consistency.

For every application a new custom network is created, using Docker's default network drivers for local and cluster environments to which containers connect. Network isolation offers increased security for services.

Finally, there is support for environment variables substitution to files, data management with four different mechanisms, private images registries and swarm cluster deployment.

1.4 THESIS OVERVIEW

Chapter 2 describes the related work around the defined problem. In Chapter 3 we present analytically the design of our tool and discuss the design choices we made. In Chapter 4 we inspect thoroughly the implementation and describe the internal structure of the tool. Finally, in Chapter 5 we state our conclusions and suggest future work.

2

RELATED WORK

In this chapter, we will present existing technologies and tools. We have divided our analysis into two major categories, one concerning environments that provide services to components which are not containers but try to resolve aspects of the same problem and the other currently developed tools for containers.

2.1 ENVIRONMENTS

2.1.1 *Application Servers*

Over the years, the two-tier client-server architecture evolved to multi-tier architecture (n-tier). Using the n-tier architecture, applications were more flexible to write and maintain while the overall performance of the system was increased. The most widespread use of multi-tier architecture is the three-tier architecture. The three-tier architecture is typically composed of a client tier, a middle tier, and a data storage tier. The middle tier controls the application's functionality and communicates with the other tiers. Access to the middle tier is provided with Application Servers.

The Application Server provides access to a set of components through APIs, making available a set of services. Such a component model is the EJB (Enterprise Java Bean) found on Java Enterprise Edition. Proceeding with our analysis, we are going to inspect the Java EE framework that defines the core set of APIs and features of a Java EE application server.

JAVA EE FRAMEWORK Java EE is a set of specifications implemented by different containers. Containers, in this context, are Java EE runtime environments hosting software components and providing a collection of services to the components[5]. Such an environment provides execution support to hosted components in a way that is similar to an operating system hosting processes. Essentially, all the low-level platform specific functionality to support a component is abstracted by containers forming an in-between interface.

JAVA EE COMPONENTS The components supported by Java EE platform are[3]

- application clients (client components).
- applets.
- java servlet, javaServer faces and javaServer pages (web components).
- ejb (business components).

A component must be assembled into a standard unit and deployed into a container, in order to be executed.

Java EE containers are runtime environments hosting components and providing services.

JAVA EE CONTAINERS The Java EE specification lists 4 types of containers[5]:

- *EJB container* is responsible for managing the execution of the enterprise java beans.
- *Web container* handles the execution of web components.
- *Application Client container* manages the execution of application client components.
- *Applet container* manages the execution of applets.

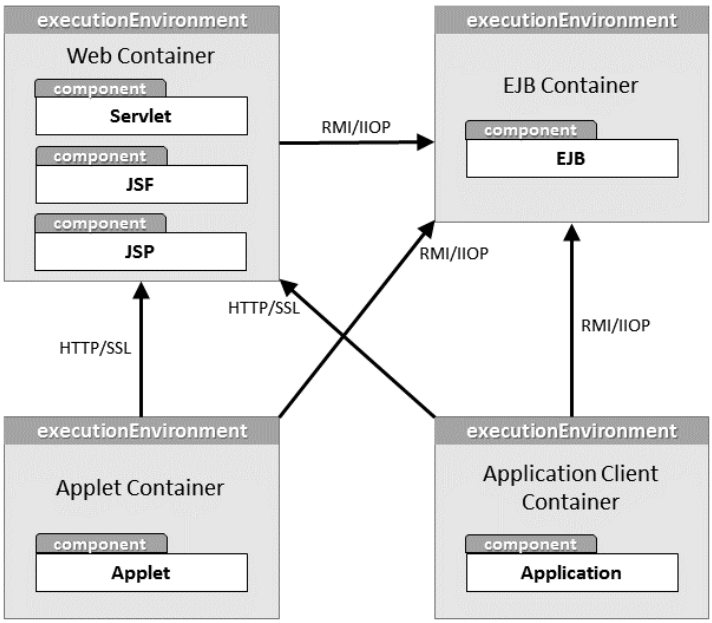

Figure 8: The container types specified in Java EE.

JAVA EE CONTAINER SERVICES The Java EE containers offer a wide range of services to the deployed components [5][3]. Some of the most important are:

- *life-cycle management*. When using a Plain Old Java Object (POJO) its life-cycle is simple: the developer creates an instance of a class and then waits for the Garbage Collector to remove it. With a Managed Bean the container handles the life-cycle as follows: creates the instance, performs any necessary dependency injection and invokes any method annotated with @PostCostruct. Then, the bean is ready for any business method invocation. At the end of the life-cycle, the container invokes any method annotated with @PreDestroy. The bean's instance is then ready for garbage collection.

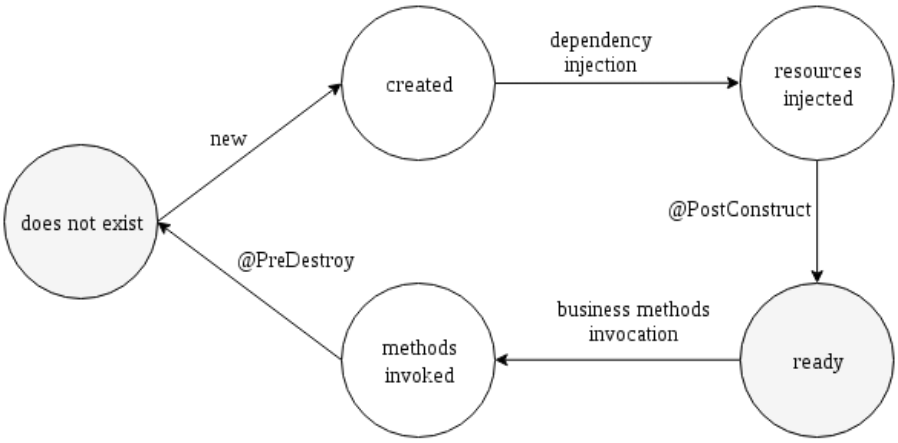

Figure 9: Managed Bean life-cycle.

- *service discovery*. Java Naming and Directory Interface (JNDI) is used to access naming and directory services. Objects are bind with names and can be looked up in a directory.
- *persistence*. Object Relational Mapping (ORM) and Java Persistence Query Language (JPQL), allow objects to be stored and then queried from a database.
- *networking*. Deployed components can be invoked with different network protocols. The supported protocols are:
  1. Http
  2. Https
  3. RMI-IIOP

- *dependency injection*. Injection of resources to managed components. Decouples dependencies from code.

As we have seen, application servers implement a model where an abstract unit, the component, is categorized into types which have access to a set of services through interfaces. The concept of managing components is extended to containers where all the previous services are necessary in order to accomplish container management.

2.2 TOOLS

With the container technology on the rise, quite a few approaches have been made to enable management of the containers for distributed applications in single and multi-host environments. We are going to inspect all those tools and analyze their characteristics in order to understand how they attempt to resolve the problem and what is their merit in the solution.

2.2.1 *Docker Compose*

Docker Compose helps the user define and run applications consisted of many containers. Compose uses a .yml configuration file, where services are defined along with networks and volumes.

Starting with networks, for every application, a new network is created (for single and multi-host), named after the project's name and containers join it automatically. Container names are used as hostnames with which services are discovered. Except from the default network, the user can specify custom networks, creating more complex network topologies and using custom network drivers. It is possible for a container of the network to be updated in case a configuration change in necessary, but it requires to remove the old container and re-deploy a new instance with different IP under the same container name. Any open connections to the old container will be lost and it is up to the container to look up the name again and reconnect.

The services section is used to describe application services. Declarations to build a new image for a service are supported, if there is no suitable base image available. In order to start the service a new command may be specified that overrides the one declared in the container image. Among other configuration options, a user may declare environment variables, networks and

volumes per service. Finally, services may declare their dependencies, which will result in instructing the program to start services in dependency order, but will not coordinate services to wait for their dependencies to initialize first before they start.

Data for the application can be managed through volumes and the *copy* command which is only supported when building an image.

Moreover, it integrates with Swarm, the native Docker solution for clustering, enabling deployment in multi-host environments.

Compose is generally feature rich and incorporates many configuration options for containers. However, there is no reliable service discovery mechanism, only the hostnames of the services are available. Data management is still not very flexible in development or production without a data transfer mechanism where the user defines data per service to be transferred to the container before or after it starts. Service coordination for distributed applications is not implemented at all. Finally, there is no standard way to control process execution to stop the container, except for waiting for the PID 1 to be signaled to do so.

2.2.2 *MaestroNG*

MaestroNG is an orchestrator of Docker-based, multi-hosts environments. MaestroNG basically takes care of two things:

- controlling the start/stop order of services, taking into consideration the dependencies defined between services.
- passing environment variables to each container in order to have all the information to function properly in the environment, in particular information concerning its dependencies.

The environment is described using a .yml file, it is named and composed of three main mandatory sections:

- registries. The registries section defines authentication information about private image registries that can be used to pull Docker images for services.
- ships. The ships section describes hosts that will run the Docker containers.

- services. The services section defines the services that compose the environment, their dependencies and instances of the services that the user wants to run. Each instance must, at minimum, define the ship its container will be placed on.

Environment variables are injected into every container in order to discover its dependencies and configure itself properly for the environment. After that, the start order is controlled by the program with respect to dependencies (the stop order too).

Volumes for containers are also supported along with bind mounts to the host to share data.

MaestroNG does not integrate with a coordination service to support service discovery or service coordination and although services get some basic information from the tool, this solution cannot enable dynamic re-configuration. Furthermore, service coordination is centralized around the program's execution. In addition, it does not support Docker networks for applications and does not offer an easy way to transfer data on remote hosts.

2.2.3 *Vamp*

Vamp is a platform for managing (micro)service oriented architectures based on containers. Vamp takes care of complex, multi-step actions like canary releases, route updates, metrics collection and service discovery.

The platform provides a model for describing microservices and their dependencies in blueprints. There is a runtime/execution engine for deploying these blueprints.

The basic entities of Vamp are:

- Breeds: describe single services along with their dependencies.
- Blueprints: describe how defined breeds work in runtime and their properties.
- Deployments: running blueprints.
- Gateways: routing endpoint, defined by its port (incoming) and routes (outgoing).

Breeds and blueprints can have a list of environment variables that will be injected into the container at runtime.

Vamp uses a service discovery pattern called server-side service discovery. When making a request to a service, the client

makes a request via a router (load balancer) that runs at a well known location. The router queries a service registry, which might be built into the router, and forwards the request to an available service instance.

2.2.4 *Capitan*

Capitan is a tool for managing multiple Docker containers. The tool provides commands that are applied to a collection of containers. Containers are described in configuration files to be read by the program.

The order of starting containers is included in the container configuration files. The order of stopping is the reverse of the starting order.

Capitan allows the use of bash as hooks for commands. For containers it allows for a custom shell command to be evaluated at the following points for each container:

- Before/After run.
- Before/After start.
- Before/After stop.
- Before/After kill.
- Before/After remove.

2.2.5 *Kontena*

Kontena is an open source project for orchestrating and running containerized workloads on a cluster. Kontena system is comprised of a number of Kontena Nodes (machines or VMs that run containerized workloads) and a Kontena Master that controls and monitors the Nodes.

An application can be described with Kontena Services definition. A service definition includes all the necessary information and configuration for the service like the container image, networking, scaling and stateful/stateless attributes. Linking of services is supported to create a desired architecture. Each service is automatically assigned with internal DNS address that can be used inside an application for inter-Service communications.

The most important features of Kontena are:

- Load-balancing for services.
- Built-in private registry for Docker images.
- Scheduler with affinity filtering.
- Log and statistics aggregation with streaming.
- Access control and roles for Kontena users.
- Remote VPN access for workload services.

Kontena supports any application that can run in a Docker container and runs on any cloud provider or local servers.

2.2.6 *Kubernetes*

Kubernetes is an open source platform developed by Google. It provides a lot of functionality for automating deployment, scaling and operations of application containers across clusters. The platform is quite broad, incorporating features usually offered from orchestration, management and deployment container tools.

Kubernetes uses the concept of the Pod. The Pod is the smallest deployable unit of computing that can be created and managed. It is a group of containers that are deployed together and are started, stopped and replicated as a group[13].

Some of the features provided are:

- service discovery.
- load balancing.
- environment variable injection.
- resource management.
- persistent volumes.
- replication of pods.
- rolling updates that update one pod at a time, rather than taking down the whole service.
- health checks on container processes and user implemented application health checks.
- secrets that store sensitive data like passwords, encryption keys e.t.c.

2.2.7 *Rancher*

Rancher is an open source software platform for running containers in production. It is an all-in-one platform. Many infrastructure services are provided for containers to facilitate deployment and management. In addition, it integrates with multiple container orchestration engines like Docker Swarm, Kubernetes and Cattle.

Some of the key features of Rancher are:

- *Stacks*. A Stack is typically a group of services that make up an application (the same concept as docker compose).
- *Networking*. Cross-host networking is supported.
- *Service Discovery*. When a service is linked to another within the same stack, a DNS record mapped to each container instance is created, enabling discovery of the required service. For that purpose, the tool implements a distributed DNS service by using its own DNS server. All services in the same stack are added to the DNS service.
- *Load balancing* for services.
- *Health checks* for containers/services.
- *Persistent Storage Services*. Enables developer to deploy storage with containerized applications.
- *Service upgrades*. An existing service may be upgraded, by allowing service cloning and redirection of service requests.

Part II

DESIGN AND IMPLEMENTATION

The following chapters present the design and implementation of Fidelio in full detail.

3

SYSTEM DESIGN

Everyday life is like programming, I guess.
If you love something you can put beauty into it.

— Donald Knuth

In this chapter, we are going to analyze in full detail the design of *Fidelio*. We are going to present the general system structure and philosophy, its functions along with its usage.

3.1 CONTAINERS AS COMPONENTS

As stated earlier in Section 1.3, *Fidelio* is a container management tool for multi-container Web Applications. The primary object of interaction with the tool is a container. Therefore, it was necessary to decide on a model that would describe the container and its interactions with other containers.

For that purpose, we introduced the concept of the *component*. Introducing this abstraction allows us to disregard all the low-level details of the container entity and helps us concentrate on its interactions with the environment and other components.

We consider an interaction with a component a *service*. *A component provides and/or requires a service*. There cannot be a component that does neither of them because that would essentially mean that there is a service of the web application that does not offer or require any service of the application which is invalid.

Based on those concepts, we implement a container-agnostic design.

Figure 10: Component providing and requiring services.

3.2 SYSTEM ARCHITECTURE

We begin with a high-level view of the system's architecture. We follow a top-down approach on presenting all the system's components and their architecture as well.

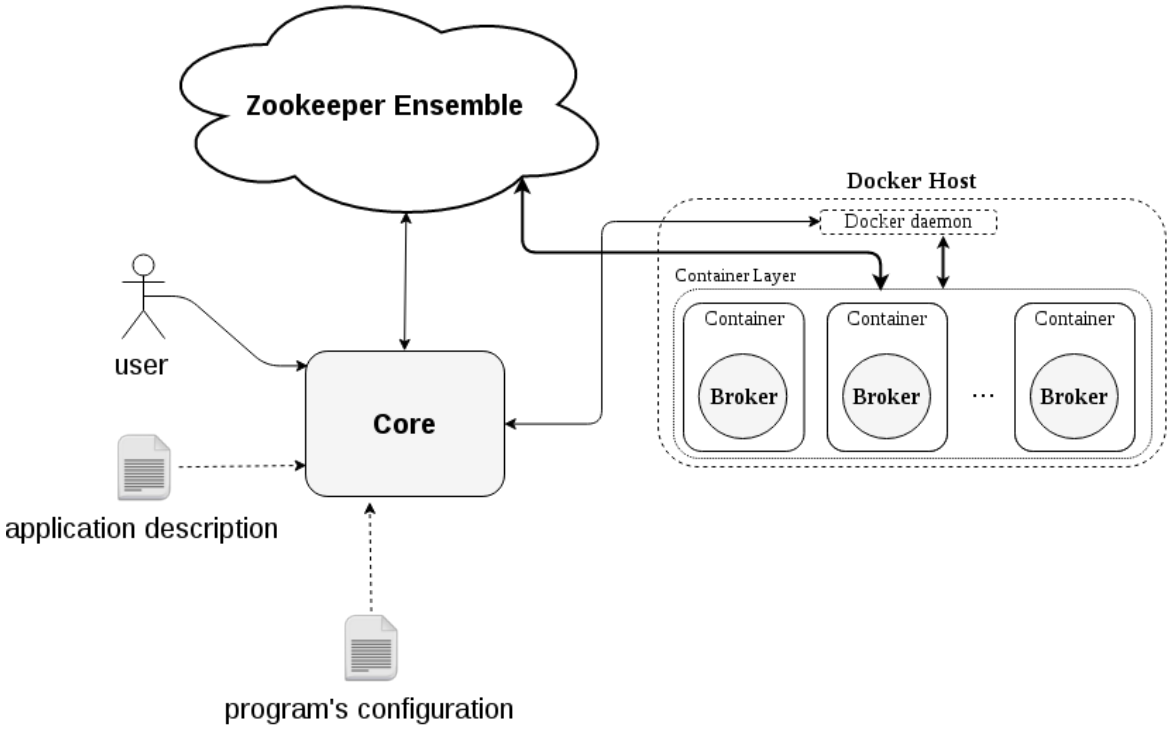

Figure 11: System architecture

TOOL MODULES As we can see in Figure 11, the tool is comprised of two modules:

- Core. The core module is the core of the tool. A configuration file is necessary to initialize the system. The user inputs commands through the user interface and uses an xml file, that describes a containerized web application, for deployment. The module communicates with Zookeeper.
- Broker. The *Broker module* is an agent. It runs inside the container providing all the necessary services and communicates with Zookeeper.

The modules coordinate their actions through Zookeeper. The architecture is completely distributed.

ZOOKEEPER ENSEMBLE It is necessary for our tool to use a service for:

- maintaining configuration information.
- maintaining a service registry for service discovery.
- providing distributed synchronization.

For that purpose, we use Apache Zookeeper, a high-performance coordination service for distributed applications [7].

DOCKER HOST The Docker Host is where the containers of the web application get deployed. It can be a single host running a Docker daemon or a Swarm cluster (see Section 3.15).

3.3 TOOL CONFIGURATION

The tool requires a bare minimum of configuration information to initialize. It cannot function without it. The configuration file is logically divided into sections, with each section referring to a different component of the system. The general structure of the configuration is:

1. *Zookeeper configuration*: This section contains the necessary information to establish communication with the Zookeeper ensemble.
2. *Xml Schema configuration*: Here we specify the xml schema file that will validate the xml application description file. The schema is provided.
3. *Docker configuration*: All initialization information concerning the Docker platform is specified here.
4. *Logging configuration*: The tool uses the log4j framework for logging. The required settings for the framework are set in this section.

The configuration file is presented in detail with analytic description and examples of all its fields in Section A.1.

3.4 WEB APPLICATION DESCRIPTION

The input data to the tool is an xml application description file. This file provides an interface to the user to map services

to containers, define services for discovery, define service dependencies for coordination, control the life-cycle of containers, manage container data, customize container networking, manage configuration and apply tasks. Overall, it describes how the application is containerized to enable its execution.

All the rules concerning the format of the xml file are dictated by an xml schema file [19] that we provide. It defines the available fields and attributes, type of data, order and occurrence.

The user creates a description for the containers of a web application using the xml file. The description is processed by the tool that enables the deployment of the application according to the user's instructions.

A top-level view of the xml file is shown at Listing 1

Listing 1: Top level elements of application description xml file.

```
<?xml version="1.0" encoding="UTF-8"?>
<webApp>
    <containers>
        <webContainer> ... </webContainer>
        <businessContainer> ... </businessContainer>
        <dataContainer> ... </dataContainer>
    </containers>
</webApp>
```

3.4.1 *Container Types*

Web applications are multi-tiered, with the 3-tier architecture being most widely used. A 3-tier architecture involves the following tiers:

- web tier.
- business tier.
- data tier.

Every service of the web application that runs on a specific tier is packaged into a container. Consequently, we define *three container types*, one for every tier, as follows:

- *WebContainer* type.
- *BusinessContainer* type.

- *DataContainer* type.

The container type is a design concept that is used to offer a high level view of the containerized services in respect to the 3-tier architecture. We do not differentiate the supported functionality or the environment for the containers of different container types in favor of flexibility. As a result, the user can perform any customization required (see Section 3.9).

A web application description *must* declare at least one container of each type (see Listing 1), with the maximum cardinality being unbounded.

3.4.2 *Container Description*

All the containers of an application must be described in the application description. We have designed a specific format for the container description. The available fields and attributes are presented in Listing 2

Listing 2: Fields and attributes of the container description.

```
<ContainerType>
    <serviceName>
    <requires>
    <docker>
        <image>
        <volumes>
        <volumesFrom>
        <bindMnt accessMode="rw/r">
            <hostPath>
            <containerPath>
        </bindMnt>
        <copy withRootDir="true/false">
            <hostPath>
            <containerPath>
        </copy>
        <publishPort protocol="tcp/udp">
            <hostIp>
            <hostPort>
            <containerPort>
        </publishPort>
        <publishAllPorts>
        <privileged>
    </docker>
    <start>
        <preMain abortOnFail="true/false">
```

```
        <main>
        <postMain abortOnFail="true/false">
    </start>
    <stop>
        <preMain>
        <main>
        <postMain>
    </stop>
    <tasks>
        <substEnv>
            <filePath restoreOnExit="true/false">
        </substEnv>
    </tasks>
    <env>
        <host_port>
        <env_declaration>
    </env>
</ContainerType>
```

In addition to the available fields/attributes, we need to know which fields/attributes are required, which can be omitted, what are the default values of attributes if they are omitted, what is the minimum and maximum occurrence of a field, what is the initialization format of a field -in case multiple values are allowed and which are the accepted data values. All those concepts are determined by the xml schema that we provide and validates the xml file. In the following sections we will illustrate those aspects and we will analyze all the fields of the description format.

3.5 CONTAINER BOOT

The boot process of a container is initiated with the execution of a boot command or a boot script. We *use an agent, the Broker module*, as the boot script for all deployed containers. It runs as the PID 1 in the container. Coordination between all the Brokers and the Core module and between Brokers is handled through Zookeeper.

The Broker communicates with Zookeeper in order to provide services to the container such as service discovery, service coordination, configuration management, service life-cycle control and tasks. Brokers need to coordinate their actions to effectively support many of those services.

3.6 SERVICE DISCOVERY

3.6.1 *Service name*

Every container of an application must declare a unique name for the service that it provides. The *service name is used for discovery by other services*. The service name must be defined using the <*srvName*> *tag*. This is a required field. The data type for the service name is *token*.

3.6.2 *Service Dependency*

Containers often depend on other provided services. When a container has service dependencies, it needs to discover the services on which it relies on using their service names. For that reason, we must define all the service dependencies of a container in application description using the <*requires*> tag. An example follows:

Listing 3: Example of <requires> tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<webApp>
    <containers>
        <webContainer> ... </webContainer>
        <businessContainer>
                <requires>data</requires>
                ...
        </businessContainer>
        <dataContainer>
                <srvName>data</srvName>
                ...
        </dataContainer>
    </containers>
</webApp>
```

The required services must be declared as *a list of space separated elements*.

3.6.3 *Client-side Discovery Pattern*

Dynamic service discovery is a key feature in containerized distributed services and our system design. There are various ways to address it. What we essentially need is:

- a service registry.
- a mechanism to register/un-register services and monitor service health.
- a mechanism to look up and connect to services.

There are two main service discovery patterns: server-side discovery and client-side discovery[12][18] which is supported in our design. When using client-side discovery, the client is responsible for determining the endpoints of the required services. The client queries a service registry, where all available services are registered, uses a load-balancing algorithm to select a service instance, if there are many instances of the available service and makes a request. In Figure 12 we can see how the client-side service discovery pattern works.

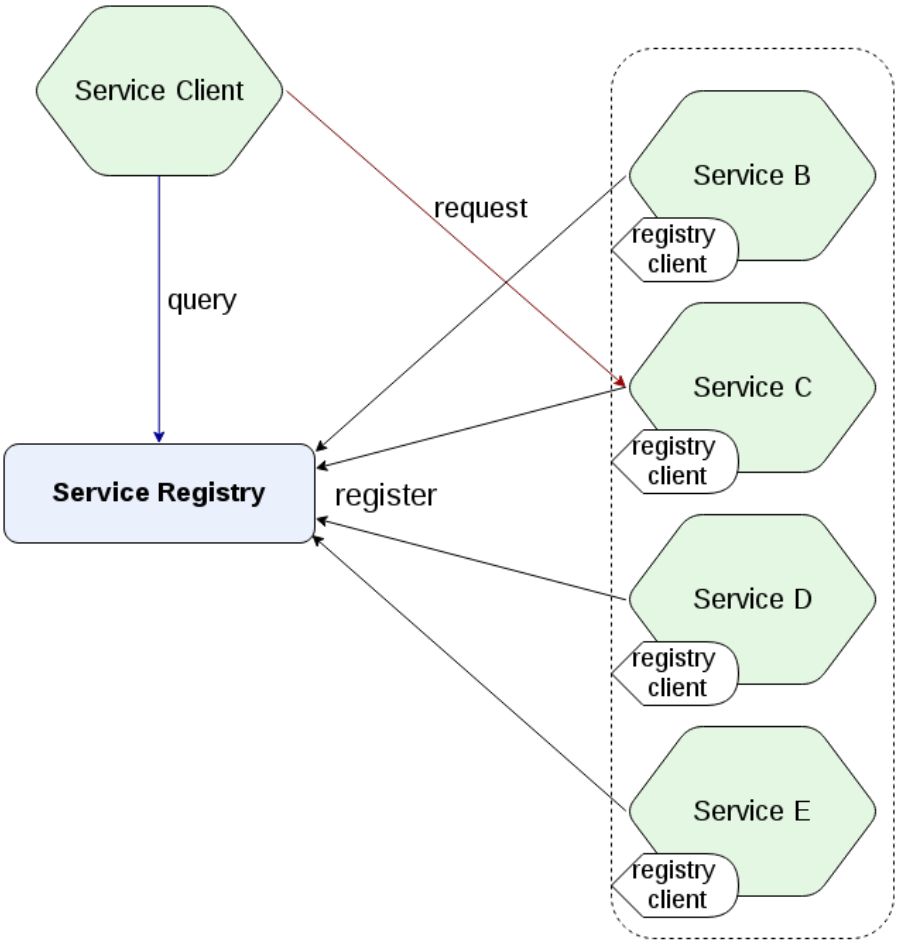

Figure 12: Client-side discovery pattern

3.6.4 *Service Discovery Mechanism*

The registry service requirement for service discovery is satisfied by using Zookeeper. The rest of the functionality is handled by the Broker module of the tool running inside the containers.

When a container starts, a service is registered to Zookeeper with the declared service name and when it stops, the service is de-registered from Zookeeper. Service registration/de-registration is accomplished through writing/deleting meta-data about the service to the registry. Discovery of required services is achieved with monitoring the service registry for Create/Read-/Update/Delete (CRUD) operations on those services. When an operation is discovered on a required service, the service's meta-data are downloaded from the registry and processed.

3.6.5 *Service Metadata*

Service metadata are data about the service. They provide the following information:

1. the service status.
2. the location in the Zookeeper namespace for the service configuration.

3.7 SERVICE STATUS

A service that is offered by a container has a *status*. The status represents *the current state of the container service* and have on one of the following values/states:

- NOT_RUNNING. The container service is not running
- NOT_INITIALIZED. The container service has not initialized yet.
- INITIALIZED. *A service is considered initialized when it is ready to accept connections and no other operation has to be executed in order to change its state or its environment* (see also Section 3.10.1).
- UPDATED[1]. The configuration of the service has changed. Signals dynamic re-configuration.

1 although update is integrated it is not supported yet.

The different states of the service status enable service coordination. One thing to notice is that there is no RUNNING state for the service status. The reason why we omitted this state is that it does not give us any functionality gain.

3.7.1 *State transitions of service status*

The state transition diagram of service status is presented at Figure 13.

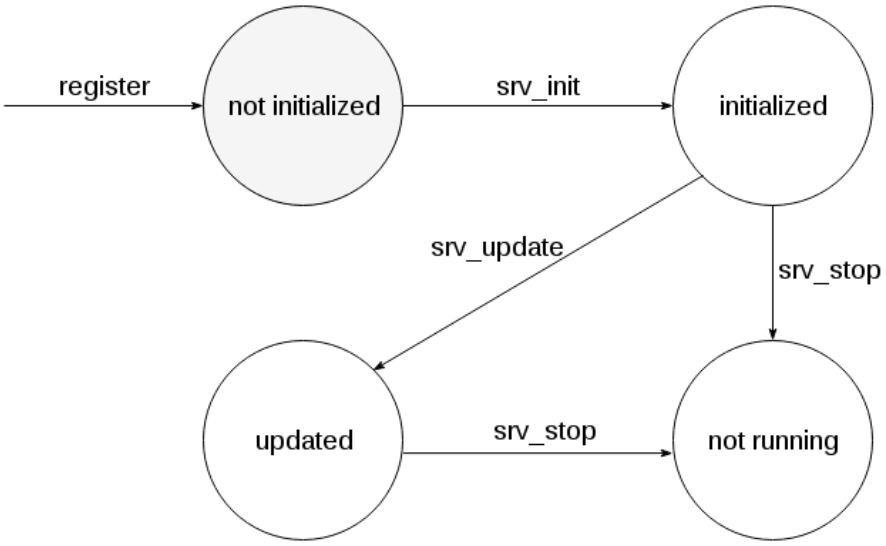

Figure 13: State transitions for service status.

When a container starts, a service is registered to Zookeeper with the service status set to NOT_INITIALIZED. This is the first state. The state encapsulates information about the execution of the main container service which may have started or not. Either way, it does not make any difference.

The only next possible state from NOT_INITIALIZED is INITIALIZED. At some point, the main container process is started. After that, the service performs any required initialization and when it is finally ready to accept connections and does not need any more configuring, the status changes to INITIALIZED.

If the main container process stops, the status changes to NOT_RUNNING, no matter which is the active state.

Finally, if there is an update to the configuration of the service, the status is set to UPDATED.

3.8 SERVICE COORDINATION

Dependent services need coordination to achieve a desired state. Using the *<requires> tag* in application description, we enable service discovery for defined services and synchronization between dependent services. *All the declared service dependencies of the application must form a DAG.*

Service coordination is designed to support the following aspects:

- *start/stop control order of services with respect to dependencies*. Services with no dependencies start first. Dependent services wait for their dependencies to start. On the contrary, on shutdown services which are not required by other services stop first. Controlling the start/stop order of services is an important first step to ensure application and data consistency.

- *initialization and complete shutdown guarantee for required services.* Starting services with respect to dependencies is not enough to ensure consistent application execution. A required service must first initialize (see Section 3.7) in order to be certain that it functions as expected. For example if we start a database and try to connect to it right away, without waiting to be ready to accept connections, our application will fail, except we explicitly specify to retry until a connection is established. Therefore, we guarantee that any service that is required from another service will start *and initialize* before it becomes available to its dependencies. What is more, shutting down the services of an application in random order could lead to data inconsistencies and unpredictable errors. For that reason, services begin to shutdown in reverse order than the one started and are waited to shutdown completely before the services that they depend begin the shutdown process.

- *notifications for state changes and monitoring of required services.* Dependent services need to be aware of changes in the status of services (see Section 3.7) they require. We make sure that services are notified for that. If a service changes multiple times between the moment a notification is sent and until it is inspected, the service client will always see the last one applied up until that moment.

Synchronization of services is accomplished using Zookeeper in association with the Brokers inside the containers. Coordina-

tion is completely decentralized relying solely on inter-container communication through the coordination service. Containers get deployed to the Docker Host and the ones with no declared dependencies start their services and begin initialization. Dependent services query the service registry for their dependencies. If they find them, they start monitoring them and wait until they become available. If they don't, they wait until they are registered. When a service becomes available, its state is processed and an action is determined depending on the current context. Services react on all state changes of a required service.

3.9 CONTAINER CONFIGURATION

Our tool offers a clean design that helps us easily setup the configuration of containers for a distributed application. The design decouples containers from their dependencies (see Section 3.1) enabling the substitution of a container with another. As a result, there is no need to re-configure the whole application.

All the configuration of a deployed application is stored to the Zookeeper configuration store. The application configuration is read, processed and then stored to Zookeeper. When the containers start, communicate with Zookeeper and retrieve the configuration required.

3.9.1 *Container Environment*

Configuring a container for a service requires to setup its environment. The environment empowers the user to decouple configuration information from service instances. This is accomplished by declaring key-value pairs of configuration settings that will substitute the hard-coded information in configuration files.

For every declared key-value pair, an environment variable is created that will become available at runtime to the executed container processes. The environment of a container can be fully customized by declaring any environment variables necessary.

To setup the environment of a container we use *the <env> tag* in the application description. In Listing 4 we see an example of a user declaration.

Listing 4: Example of environment variable declaration.

```
<?xml version="1.0" encoding="UTF-8"?>
<webApp>
    <containers>
        <webContainer>
            ...
            <env>
                <env_var>value</env_var>
            </env>
        </webContainer>
        ...
    </containers>
</webApp>
```

The declaration is done using lowercase letters and the underscore.

If a container has any service dependencies declared, then the environment variables of its dependencies are injected into the container as well. We will see how we can reference and use the environment variables in Section 3.9.3.

The Broker module handles the proper setup of the container's execution environment. It reads the configuration and exports the environment variables making the available at runtime to processes.

3.9.2 *Host Fields*

For every container there are two environment variables that are designed to refer to the container:

- ${HOST_IP}. The HOST_IP environment variable does not appear anywhere in the application description. It can be used to obtain the IP of the container. It cannot be set by the user. It is set and updated automatically.

- ${HOST_PORT}. The HOST_PORT environment variable must be defined in the application description. It is a mandatory field that every container has to specify in its environment. It *specifies the port on which the main container service is running.* If this field is not set correctly, the container will halt.

3.9.3 *Availability And Usage*

The container environment consists of all the environment variables defined in the application description for the container along with all the environment variables of its declared dependencies.

The environment variables become available to the container processes at runtime. We can use environment variables in configuration files instead of hard-coded information and then apply environment variable substitution to the configuration files (see Section 3.14)

The general format to reference an environment variable declared in *the <env> tag* of the container is: ${ENV_VAR}. ONLY the format with brackets is supported.

The general format to reference an environment variable of a declared service dependency is: ${SRV_NAME_ENV_VAR}, where SRV_NAME is the name of a service dependency.

To sum up, a container has access to environment variables that concern:

1. the container. Those environment variables are declared to *the <env> tag* of the container.
2. the dependencies of the container. Those environment variables are declared to *the <env> tag* of the container's dependencies.

An example illustrating the above follows.

```
Let three containers with dependencies as follows:
        data <-- business <-- web

The environment variable declarations for every service is:
        data: db_name
        business: app_name
        web: my_id

The created environment variables in every container are:
        data: ${DB_NAME}
        business: ${APP_NAME}, ${DATA_DB_NAME}
        web: ${MY_ID}, ${BUSINESS_APP_NAME}
```

Finally, we can use environment variables that refer to a dependency as values to environment variable declarations of a container. Example:

```
Containers with dependencies:
        data <-- business

Environment variable declarations
        business: db_used=${DATA_MY_DB}
        data: my_db
```

3.9.4 *Re-configuration*

Due to the fact that all the configuration of a deployed application are managed through Zookeeper, if we stop and restart an application the re-configuration process is handled automatically and transparently by the tool.

3.10 PROCESS EXECUTION MECHANISM

Managing the life-cycle of a service in a container usually requires the execution of multiple processes. We need to be able to control how the service starts and how it stops.

We have designed a mechanism to handle process execution in containers. Processes are arranged in process groups. A process group is a set of processes that are executed together as a unit in order to enable the container to achieve a certain state. We define two process groups:

- *start process group*. The processes defined in this group execute the necessary logic to start and initialize the container service.
- *stop process group*. This process group handles the shutdown procedure of the container service.

Every process group allows for multiple process declarations. The start group takes precedence over the stop group in all scenarios. If the stop group is activated while the start group is running, the stop group processes will be queued for execution after all processes of the start group.

PROCESS We consider a process to be associated with:

- an environment. The environment is consisted of all the necessary environment variables that will be injected into the process in order to support its execution (see Section 3.9.1).

- *a resource*. The resource is an execution description. It contains the execution command, its arguments and its properties (e.g. abortOnFail, see Section 3.10.1).

DEFINITION FORMAT The supported format to define a process to run is:

- executable/script ARG1 ARG2 ... ARGN

Tokens are split using the space delimiter. Tokens surrounded by double or single quotes are parsed as one. Moreover, environment variables from the container's environment or its dependencies may be used as arguments. The declared environment variables are automatically expanded.

3.10.1 *Start Process Group*

The start process group is divided into three sections:

1. *preMain*. In this section we declare any processes that we want to be executed before the container service starts. We can declare as many processes as we want. Declarations are optional.
2. *main*. The main section defines the execution of the main container process which runs the container service. Unlike the other sections, this is an one process section. A main container process must always be defined, so that a service is provided.
3. *postMain*. In this section we declare any processes that we would like to be executed after the container service has started and initialized. We can declare as many processes as we want. Declarations are optional.

3.10.1.1 *Order Of Execution*

Declared container processes in start process group are executed *serially* in the order they were declared. A process is waited to be executed before the next declared process starts. There is a timeout for the execution of every process to avoid waiting for a process indefinitely.

3.10.1.2 *Abort On Fail*

In *preMain and postMain* processes, the user can specify per process to ignore or halt on any errors encountered during execution. There are two main scenarios to consider:

- An error occurs on a process and *we have defined to ignore errors.* In this case, the process execution will continue with the next process in the sequence. This practice is generally discouraged because it may lead to data or service inconsistencies if used unwise. It can be used though to apply an optional feature that will not have any impact in case it fails, such as an update.
- An error occurs on a process and *we have defined to halt*. There are two sub-cases in this scenario:
  - the executing process is *a preMain process*. All remaining processes will be cancelled and the whole start sequence will be aborted. That includes any remaining preMain processes, the main process and any postMain processes.
  - the executing process is *a postMain process*. All remaining processes will be canceled and the whole start sequence will be aborted. But in this case, there may be only postMain processes left to run. The main container process has already started. In order to abort the start sequence the main process will be forced to stop.

3.10.1.3 *Main Section*

The *main* section of the start process group provides additional features.

As we stated earlier, in this section we define a process that will start the container service. A service will run for an indefinite amount of time and will exit only when it is stopped or an error occurs.

When the main process starts the container service, we *wait until the service is ready to accept connections* in order to continue the process execution. We don't wait the process to complete its execution because that would be meaningless. We poll the service for availability for a timeout period. If the timeout is exceeded or an error occurs the whole start sequence is aborted. Otherwise, we continue with the execution of the postMain processes.

The container service is constantly monitored, in order to be able to react in case it stops unexpectedly.

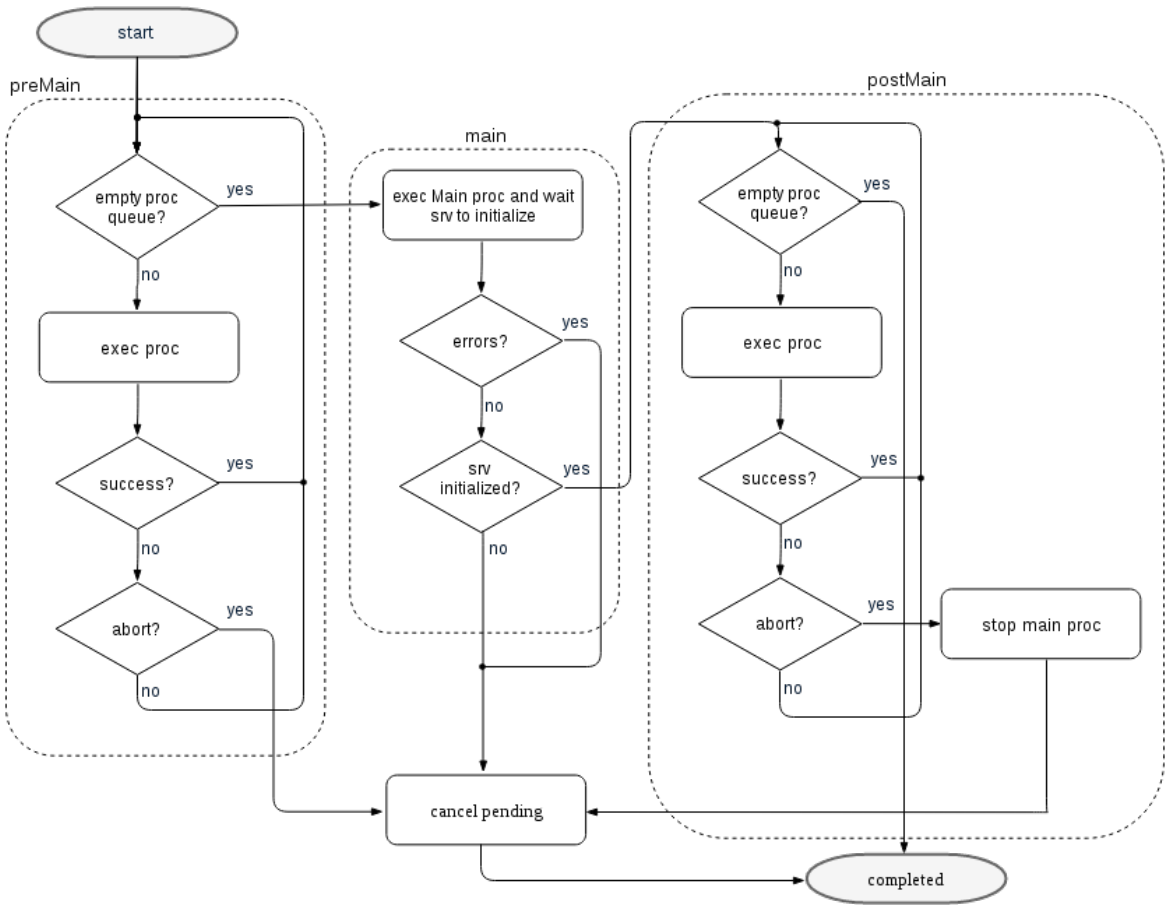

Figure 14: Start process group execution flow.

CONTAINER SERVICE INITIALIZATION When the start process group executes successfully, the container service is considered to be INITIALIZED (see also Section 3.7).

3.10.2 *Stop Process Group*

The stop process group is divided into three sections:

1. *preMain*. In this section we declare any processes that we want to be executed before the container service stops. We can declare as many processes as we want. Declarations are optional.

2. *main*. The main section defines the process that stops the container service. Declaration is mandatory.

3. *postMain*. In this section we declare any processes that we would like to be executed after the container service has stopped. We can declare as many processes as we want. Declarations are optional.

3.10.2.1 *Execution*

Declared container processes in stop process group are executed *serially* in the order they were declared. A process is waited to be executed before the next declared process starts. There is a timeout for the execution of every process to avoid waiting for a process indefinitely.

There is no abort-on-fail option (see Section 3.10.1.2) in stop process group, in contrast with start process group, because when the stop group is being executed the application services need to stop and the containers cannot revert back to any other state.

3.11 CONTAINER LIFE-CYCLE MANAGEMENT

Containers have their life-cycle. A containerized web application requires the user to control the life-cycle of many containers which is very demanding. Our tool provides life-cycle management for the containers of a web application. It fully automates the process of how containers are created, started, stopped and deleted. We achieve that with a two step approach. First, we provide a way to describe how every state is accomplished (see Section 3.4). Second, we introduce an execution mechanism (see Section 3.10) to support the execution of the described functionality.

In Figure 17 we can see the life-cycle of a container managed by the tool.

3.11.1 *Create*

A container is created based on the characteristics that we have defined using *the <docker> tag*. Listing 5 shows the available fields.

Listing 5: The <docker> tag in application description.

```
<docker>
    <image>
<!-- data settings -->
```

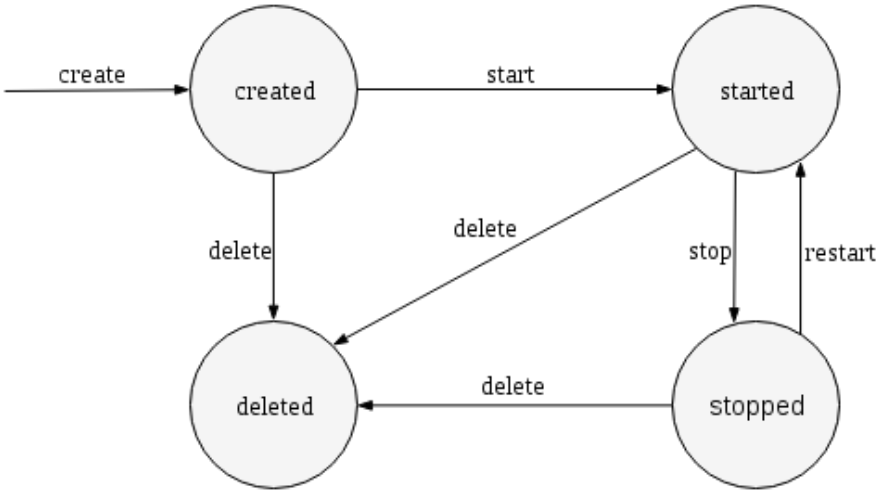

Figure 15: Managed container life-cycle.

```
    <volumes>
    <volumesFrom>
    <bindMnt>
        <hostPath>
        <containerPath>
    </bindMnt>
    <copy withRootDir="true/false">
        <hostPath>
        <containerPath>
    </copy>
<!-- network settings -->
    <publishPort protocol="tcp/udp">
        <hostIp>
        <hostPort>
        <containerPort>
    </publishPort>
    <publishAllPorts>
    <privileged>
</docker>
```

The <image> tag is used to define the base image for the container that holds data, binaries and libraries.

As we can see in Listing 5 there is a data section and a network section. The data settings will be explained in Section 3.12 and the network setting in Section 3.13.

The user provides the create description for the containers of the application and the tool handles the process of creating all containers transparently.

3.11.2 *Start*

After a container is successfully created, it is automatically started. Starting a container means to execute a process that is responsible for starting the container service. In order to run a service, a series of actions are usually required. For that reason, we have designed a mechanism (see Section 3.10) to facilitate process execution in containers providing control, flexibility and a clean interface towards that matter.

Using *the <start> tag* in application description as we can see in Listing 6, we can define what processes are going to be executed in order to start the container service and accomplish its desired state. We will explain only in brief Listing 6 as there is an in detail analysis regarding the design and features of the execution mechanism in Section 3.10

Listing 6: The <start> tag in application description.

```
<start>
    <preMain abortOnFail="true/false">
        ...
    <main>
        <postMain abortOnFail="true/false">
        ...
</start>
```

In *the main section* we declare a process that will start the container service. In *the pre/post-main sections* we declare processes to run before (pre) the container service starts and after (post) it becomes available accepting connections. We set *the attribute abortOnFail* if we want to halt on any errors encountered.

Normally, a container is started right after it is created. If any error occurs in-between, it will not change state. In this case, It can be later deleted. The container will maintain the start state until instructed otherwise by the tool.

3.11.3 *Stop*

A successful service execution concludes with a controlled and consistent shutdown. The common practice to stop a service running in a Docker container is to signal the service to stop, which is unreliable, lucks flexibility, control and the ability to guarantee data consistency. Our approach provides for controlled service shutdown. In Listing 7 we can see the interface provided.

Listing 7: The <stop> tag in application description.

```
<stop>
    <preMain>
            ...
    <main>
    <postMain>
            ...
</stop>
```

The main section is mandatory. We define how to stop the running service. We may also define processes to run before or after we initiate the service shutdown.

A stopped container can either be re-started or deleted.

3.11.4 *Delete*

A container is deleted automatically when the deployed application gets deleted. A delete request is always forced in order to apply independently of the container state. As a result, the container will be deleted either it is running or not.

3.12 CONTAINER DATA MANAGEMENT

The ephemeral nature of the containers on one side and the inherent demand of web applications to persist and share data on the other, mandate the need to find strategies to manage container data.

The tool supports four different mechanisms that enable data management in regard to the following aspects:

- persistence.
- sharing.

3.12.1 *Docker Volumes*

Using Docker volumes we can persist and share data. Every volume is a mount point on the container directory tree to a location on the Docker host directory tree [14]. Docker provides two types of volumes and Fidelio supports both:

1. *managed volumes*. We only specify the mount point location on the container directory tree while the location on

the host file system is managed by Docker. Managed volumes have the advantage of being decoupled from specific locations on the host file system. They provide an easy way to persist data beyond the container's scope. We declare managed volumes using *the <volumes> tag* in application description. As we can see in Listing 8, multiple declarations are allowed space or newline delimited.

2. *bind-mount volumes*. The location to the directory tree of the host file system to which the volume points is specified by the user. *We can use this type to easily share data on the host with containers*. In Listing 8, we see the general format for declaring a bind-mount volume. We can mount a volume as read-only/read-write in the container by setting *the accessMode attribute*.

When using bind-mount volumes, we need to keep in mind that *the binds refer to directories on the Docker host not the host where Fidelio is running.* We use bind-mount volumes on a host that we have physical access not a remote host.

Listing 8: Data management section in application description.

```
<docker>
    ...
    <volumes>
        /path1/to/container
        ...
    </volumes>
    <volumesFrom>
        container1
        ...
    </volumesFrom>
    <bindMnt accessMode="rw/r">
        <hostPath>
        <containerPath>
    </bindMnt>
    ...
    <copy withRootDir="true/false">
        <hostPath>
        <containerPath>
    </copy>
    ...
</docker>
```

3.12.2 *Volumes From*

If we have some persistent data stored in volumes that we want to share between containers, or want to use from non-persistent containers, we may copy the volumes from one container to another. For that purpose, we can use *the <volumesFrom> tag* in application description. Copying of volumes is transitive, so if we copy the volumes from a container we will also copy the volumes that it has copied. When using this option we should keep in mind that:

1. re-mapping of mount points is not supported.
2. volume sources that use the same mount point conflict.
3. the write permissions set on a copied volume cannot be changed.

3.12.3 *Copy*

The tool provides a *copy option* so that we can transfer data from a host to a container. This is useful especially when we do not have physical access to the Docker Host such as with remote Docker hosts or in a cluster and as a result we cannot use bind-mount volumes to share data between the host and containers. In Listing 8, we can see the declaration format for using this option in application description. Multiple declarations are allowed. We also provide *the withRootDir attribute* which if set copies the contents of the defined resource with the root dir.

3.13 CONTAINER NETWORKING

There are two ways with which Fidelio supports the networking of containers. The first is through network settings that can be defined per container in application description. The second is by integrating with Docker networks and deploying containers in custom-created per application networks.

3.13.1 *Container network settings*

The available network settings for a container are listed in Listing 9.

Listing 9: Network settings of container in application description.

```
<publishPort protocol="tcp/udp">
    <hostIp>
    <hostPort>
    <containerPort>
</publishPort>
<publishAllPorts>
```

The <publishPort> tag binds a port on the container to the host. We can use this option to make a container service available to inbound network traffic by providing access to the mapped host port. If the host port is not defined, a random high-numbered port will be chosen. This is useful so that we avoid conflicts by binding two container ports to the same host port. The host interface on which to expose the port may also be specified. It allows container services to be contacted through a specific external interface on the host machine. If no host interface is set, the port is exposed to all interfaces.

Docker containers usually have network ports exposed. Exposing a port is a means of documenting which ports provide services. We can automatically bind all those ports to the host using *the <publishAllPorts> tag*. This option uses available random high-numbered ports and avoids conflicts.

3.13.2 *Application Defined Networks*

Fidelio uses Docker networks by default for container networking. For every web application, the tool *creates a new custom network that all the containers join*. The custom network provides isolation for the containers which are not accessible from any interface. If a container service needs to be exposed, we can publish the container port and Docker will make sure to write the required routing rules to allow network traffic.

The custom application network is created using the default Docker driver which differs according to the deployed environment. The available options are:

- *bridge*. This driver is used to create networks on *single-host* environments.
- *overlay*. Used to create networks on *swarm cluster* environments that span multiple hosts.

Fidelio *manages the life-cycle of networks* automatically. A new network is always created in order to deploy a web application and gets removed when the application is deleted.

3.14 TASKS

Tasks provide extra functionality by the tool so that certain actions are facilitated and automated. *A task is a function of some type.* Our design allows for integrating new Tasks using the Application Programming Interface (API). We have designed and integrated one Task with the tool which we will inspect right away.

3.14.1 *Substenv*

This task *performs environment variable substitution in configuration files* so that the user does not have to do it manually. All the variables that constitute the container's environment (declared in container's environment description or its dependencies) are expanded. Environment variable declarations in configuration files must comply to the following format: *${ENV_VAR}*. Substitution is performed *before* any declared processes from the container's start section get executed.

In addition, the task can automatically restore the edited configuration files to their previous form after the container service stops. The action is applied *after* the stop section of the container's description is executed. In Listing 10 we can see the declaration format of the task in application description.

Listing 10: SubstEnv task format in application description.

```
<tasks>
    <substEnv>
        <filePath restoreOnExit="true/false">/path/to/
            container</filePath>
                ...
    </substEnv>
</tasks>
```

3.15 CLUSTER DEPLOYMENT

Fidelio supports deployment of containers in cluster environments. The tool transparently integrates with Docker Swarm to provide *deployment in Swarm clusters*. Docker Swarm is native clustering for Docker hosts. Fidelio does not require any custom settings, other than those used with single Docker hosts, to enable cluster deployment (see Section 3.3). In Figure 16,

we can see a system overview when deploying containers to a swarm cluster.

When deploying on clusters we should use the *data and network settings* for containers that offer support for remote hosts (see Section 3.12, Section 3.13).

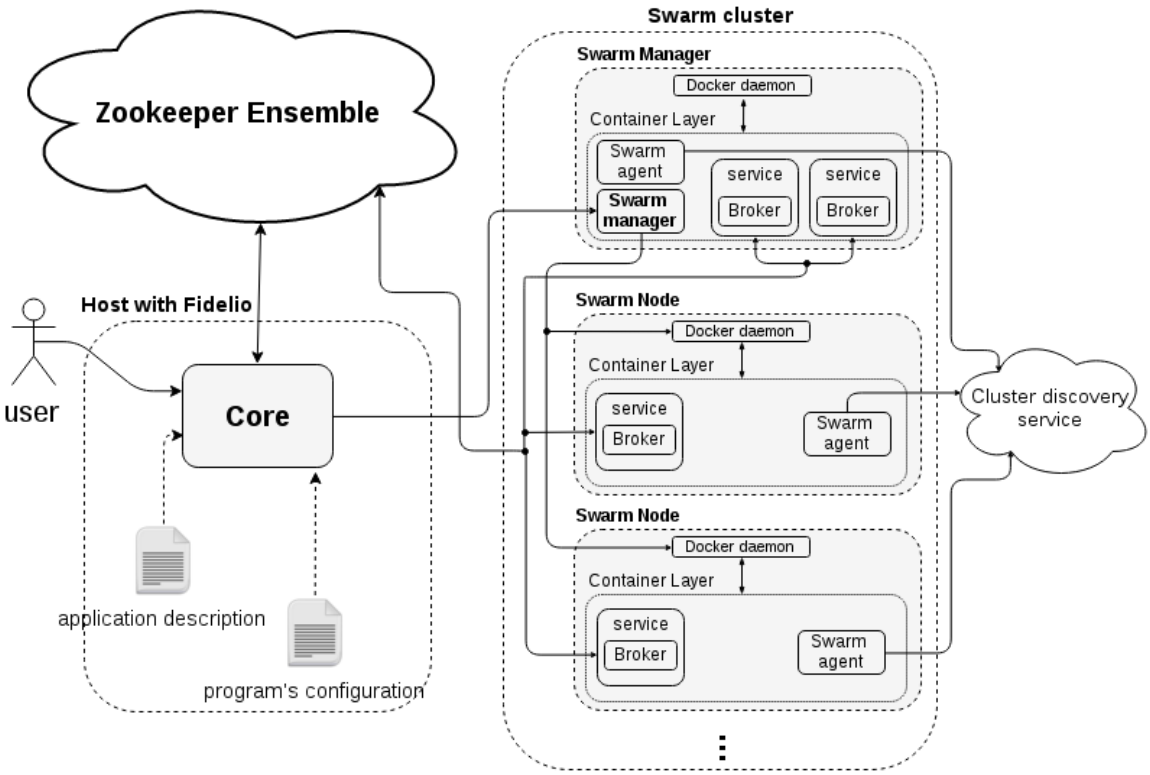

Figure 16: System overview with swarm cluster.

3.16 USER INTERFACE

We have designed a simple user interface that provides basic *commands* to manage the deployment of web applications with Fidelio.

The available commands of the interface are:

- *start*. Starts the deployment of a web application.
- *stop*. Stops a deployed web application.
- *restart*. Restarts a deployed web application.
- *delete*. Deletes a deployed web application.

All the configuration settings for the tool can be set from the command line with command line options. The full description of the user interface with command syntax and available command line options can be found at Section A.2.

3.17 COORDINATION SERVICE

As stated in Section 3.2 our tool transparently integrates with Apache Zookeeper, a distributed coordination service for distributed applications. Fidelio uses Zookeeper as:

- *configuration store*. All the configuration for a deployed web application is stored in Zookeeper. Fidelio manages the configuration of services and enables more advanced operations.
- *service registry*. Services register themselves with Zookeeper in order to be discovered.
- *service for distributed coordination*. Dependent services of an application synchronize their actions through Zookeeper.

We will now give a simple and brief overview of the Zookeeper design.

3.17.1 *Zookeeper Design*

The basic design features of Zookeeper are:

- *simplicity*. Zookeeper offers a shared hierarchical namespace resembling a standard file system. The namespace uses data units called *znodes*. The namespace is stored in memory and as a result each znode holds up to 1MB of data (can be increased but is strongly advised not to).
- *replication*. A set of hosts over which Zookeeper is replicated consist the *Zookeeper ensemble*. The service is available as long as a majority of the servers are available. A client connects to a single server.
- *speed*. Very fast in read-dominant workloads, where reads are more common than writes, at ratios of around 10:1.
- *ordering*. Every update is stamped with a number that indicates the order of all Zookeeper transactions. Based on that, synchronization primitives can be implemented.

3.17.2 *Zookeeper Data Model*

Zookeeper provides a hierarchical namespace where every znode in the namespace is identified by a path, stores data and can have children. Znodes can be ephemeral which means that they exist as long as the session that created the znode is active. Moreover, zooKeeper supports the concept of *watches* for znodes which can be registered by clients. A watch will be triggered and removed when the znode changes.

4

IMPLEMENTATION

In Chapter 3 we introduced the general design of our tool. In this chapter, we will review important aspects of the implementation and inspect the internal structure of the tool.

Fidelio is a tool built to manage containerized web applications. It was developed using the *Java* programming language. The tool has *two modules*, the core module that runs the program and the broker module that runs inside all deployed containers of an application. It is released under the *GNU General Public License v3.0*[1] and it is *free software*.

4.1 LIBRARIES

We will refer in brief to the software used by Fidelio. All libraries are GPL compatible[4].

4.1.1 *Zookeeper*

One of the key features of our tool is that it integrates with Apache Zookeeper. We use the *Java binding* of the *Zookeeper client library*. The library is used to establish communication with the Zookeeper ensemble and issue requests. The provided API enables us to create the required functionality to support distributed synchronization, service discovery and configuration management. The API comes in two versions: a synchronous and an asynchronous [20][7]. We make extended use of the asynchronous version, that does not block, implementing an event-driven design and increasing concurrency. We also make little use of the synchronous version when we want to dispatch a request and process the response right away in order to continue execution. We will see in detail both programming patterns in Section 4.4.

4.1.2 *Docker java client*

A Docker client is required so that the tool can communicate with the Docker daemon. We use the *docker-java-client* for that

purpose. The client uses the remote Docker API to talk to the Docker daemon. It handles all the interactions concerning the Docker platform.

4.1.3 *Sl4j*

This library allows the end user to plug in the desired logging framework at deployment time.

4.1.4 *Log4j*

Log4j is the default logging framework bundled with our software for logging. It requires a configuration file for initialization which makes it easy to use and customize. It is also required from the Zookeeper client library.

4.1.5 *JCommander*

We use this library to parse command line arguments. It has a simple interface and parsing is quite straightforward. It allows to define options and commands for parsing with minimum effort while giving us control over the processing. The library supports the provided user interface.

4.2 CONTRIBUTIONS

In the process of the tool's development we have made contributions to the *docker-java-client* library, which is an important unit for our software. We have added functionality to the library, corrected bugs and implemented unit tests for the contributed code. The code can be found here [9].

4.3 CONCEPTS

4.3.1 *Handlers*

The modules of our tool are concerned with disparate resources. We consider a *resource* to be an entity to which we can apply an action. An *action* defines a way to manipulate a resource towards an end. In order to apply actions we need a provider. Therefore, we introduce the concept of *the handler*. A *handler*

is a provider for actions on resources. It manipulates resources by applying actions and performing operations.

The concept of the handler allows for a very modular design. For every resource or collection of resources we assign a handler. As a result, the domain of resources is partitioned to sub-domains, with each sub-domain providing knowledge that is manipulated by a handler. The actions applied by a handler constitute the API that is exposed to other components.

4.4 PROGRAMMING WITH ZOOKEEPER

The Zookeeper API provides both synchronous and asynchronous methods. Our implementation is mostly event-driven and therefore we rely heavily on the asynchronous programming paradigm. There are cases though, that we use synchronous methods, for example when we want the execution to continue based on a processed response and we do not want any concurrent actions to happen. We are going to present the patterns used for asynchronous and synchronous requests.

4.4.1 *Asynchronous Pattern*

The asynchronous methods of the Zookeeper API have the advantage that they do not block. A method is executed and returns immediately. All the asynchronous methods have void return types. The result of the operation is conveyed via a callback. A callback implementation is passed by the caller. The callback's method is invoked when a response is received from ZooKeeper.

A common pattern we use with the asynchronous methods is:

1. Make an asynchronous call.
2. Implement a callback object and pass it to the asynchronous call.
3. If the operation requires setting a watch, then implement a Watcher object and pass it on to the asynchronous call.

A code sample for this pattern using the *getData* zookeeper method follows:

Listing 11: Pattern example for asynchronous zookeeper getData call.

```
// make synchronous method call
zk.getData("/aZnode",
        getDataWatcher,
        getDataCallback,
        null);

Watcher getDataWatcher = new Watcher() {
        public void process(WatchedEvent e) {
        // process the watch event
        }
}

DataCallback getDataCallback = new DataCallback() {
        @Override
        public void processResult (int rc, String path,
            Object ctx, byte[] data, Stat stat) {
        // process the result of the getData call
        }
};
```

Initially, we use the *getData* method to make an asynchronous call. We implement a *callback* and a *watcher* (if necessary) and pass them to the method call. When a response is received the callback's method gets invoked and its code gets executed. The watcher's method will be invoked when an event happens.

We use this pattern extensively in our implementation.

4.4.1.1 *Response Processing*

Zookeeper library has a single dedicated callback thread that processes responses to asynchronous calls. Responses are processed in the order they are received. We do not do intensive operations or blocking operations in a callback because if a callback blocks, it blocks all the callbacks that follow it.

The callback's method that gets invoked has a *return code argument* which is a code that corresponds to a *KeeperException* if not zero. We convert the code to an enum and use switch to handle cases. In Listing 12 we can see the general structure.

Listing 12: Response processing in callback.

```
switch (KeeperException.Code.get(rc)) {
    case CONNECTIONLOSS:
        // make again asynchronous method call
        break;
    case OK:
```

```
        // process result
        break;
    default:
        // default action on other error codes
        LOG.error("Something went wrong: ",
                KeeperException.create(KeeperException.Code.
                    get(rc), path));
}
```

CONNECTIONLOSS The *ConnectionLoss code* is returned when the client becomes disconnected from a ZooKeeper server because of a network partition or a failure of the server. When this happens, the client does not know whether the request issued was processed by the server and the response was lost or if the request was not processed at all. The library handles the connection of future requests but we need to determine whether the request was processed or not in order to reissue it.

We handle connectionLoss by reissuing the request and processing the result accordingly.

4.4.2 *Synchronous Pattern*

For every asynchronous method of the Zookeeper API there is a synchronous equivalent. The synchronous method calls block until a response is returned. A common pattern we use with the synchronous methods is:

1. Make a synchronous call.
2. Implement the required logic to process the response of the synchronous call.
3. If the operation requires setting a watch, then implement a Watcher object and pass it on to the synchronous call.

A code sample for this pattern using the *getData* zookeeper method follows:

Listing 13: Pattern example for synchronous zookeeper getData call.

```
// make synchronous method call
byte[] data = zk.getData(zkPath, getDataWatcher,  null);
// process the result of the getData call

Watcher getDataWatcher = new Watcher() {
```

```
    public void process(WatchedEvent e) {
    // process the watch event
    }
}
```

4.4.2.1 *Exception Handling*

Processing of a response from a synchronous method call happens right after the method call, as a result of blocking and waiting the server.

A synchronous method call throws a *KeeperException* and *InterruptedException* which we handle in a try/catch block.

CONNECTIONLOSS EXCEPTION In order to handle the *ConnectionLoss exception*, we wrap the method call to a while loop, so that we can reissue it and process it accordingly.

INTERRUPTED EXCEPTION Because we use the method call in a while loop, we catch the *InterruptedException* log it, set the interrupt status and exit the loop in order to stop execution.

In Listing 14 we can see how all the above are combined.

Listing 14: Exception handling with synchronous method call.

```
while (true) {
    try {
        // make synchronous method call
        byte[] data = zk.getData(zkPath, getDataWatcher,
            null);
        // process result
        break;
    } catch (InterruptedException ex) {
        // log event
        LOG.warn("Interrupted. Stopping");
        // set interupt flag
        Thread.currentThread().interrupt();
        break;
    } catch (ConnectionLossException ex) {
        LOG.warn("Connection loss was detected! Retrying...")
            ;
    } catch (KeeperException ex) {
        LOG.error("Something went wrong {}", ex.getMessag())
            ;
        break;
    }
```

```
}
```

4.5 XML SCHEMA

In order to be able to deploy and manage a containerized web application, the tool requires a description of the web application. The description must be provided in a document written in Extensible Markup Language (XML) . We use an XML schema in order to express constraints about the XML document. There are several different schema languages. In our approach we use W3C XML Schema Definitions (XSD).

4.6 CORE MODULE

The core module has several sub-systems/packages in order to provide its functionality. It implements a modular design with respect to the separation of concerns [11]. Proceeding with our analysis we will present all the sub-systems of the core module and analyze the provided functionality along with the components that build each unit.

4.6.1 *Boot*

The *boot package* is responsible for booting the tool. It contains *the entrypoint* of the program. We also define one class with *the command line arguments* to be parsed and another one to hold the *program's configuration*.

When the program boots a series of actions happen:

- the command line arguments are parsed.
- the program's configuration is set and checked for initialization.
- if no errors encountered, control is passed to the subsystem handling commands.

In Listing 15 we see the structure of the *boot package*.

Listing 15: Boot package

```
|-- boot
|   |-- cl
|       |-- CliOptions.java
```

```
|-- Main.java
|-- ProgramConf.java
```

The tool loads by default the files with the program's configuration and logging properties found in the currently executing path.

4.6.2 *Cmd*

Here we specify all *the commands* of the tool. Every command must extend the abstract class *Command*. We define a class that *handles the execution of the commands, the CommandHandler*. The handler class implements the Commandable interface which enables the execution of the commands.

In Listing 16 we see the structure of the *cmd package*.

Listing 16: Cmd package

```
|--cmd
   |-- Commandable.java
   |-- CommandHandler.java
   |-- Command.java
   |-- DeleteCmd.java
   |-- RestartCmd.java
   |-- StartCmd.java
   |-- StopCmd.java
```

4.6.3 *Xml*

This is a simple sub-system that does all the *xml processing* for the tool. We define an *XmlProcessor* that has the following responsibilities:

- *validates the xml file* with the application description.
- *validates the xml schema* file.
- *validates the xml file against the xml schema.*
- *un-marshals* the xml file to Java Objects (binding).

4.6.4 *Schema*

In this package we include all the classes that are generated from the xml schema. We use Java Architecture for Xml Binding

(JAXB) for that purpose. The classes are used for marshaling/unmarshaling operations between xml and Java Objects.

4.6.5 *Analyzers*

Although we use the xml schema to express constraints on the application description xml document, we define the *analyzers package* to enable more refined restriction control that cannot be performed using the schema.

An analyzer of some type is a class whose instances are used to check if a condition applies. We define two analyzers:

- *DependencyAnalyzer*. Provides methods to verify that the declared dependencies of a web application form a DAG.
- *ContainerNameAnalyzer*. Provides methods to assure the uniqueness of container names.

Moreover, we define an *Analyzer class* which includes instances of all analyzers. We create an instance of this class which provides access to all analyzers and their methods in order to check the required conditions.

In Listing 17 we see the structure of the *analyzers package*.

Listing 17: Analyzers package

```
|-- analyzers
   |-- Analyzer.java
   |-- ContainerNameAnalyzer.java
   |-- DependencyAnalyzer.java
```

4.6.6 *Handlers*

In Section 4.3.1 we introduced the reader to the concept of the *handler*. For the core module we define in the *handlers package* two handlers:

1. *ContainerHandler*. This is a very important class whose instances provide access to all the defined container descriptions. We use it to extract and manipulate the information stored for containers.
2. *NetworkHandler*. Class whose instances handle the interaction with the networks for the deployed application. We use it to manage the life-cycle of networks.

In Listing 18 we see a code snippet with some method signatures from the API of *ContainerHandler*.

Listing 18: Code snippet with some method signatures from ContainerHandler.

```
/**
 * Lists the available containers.
 * @return the list with the available containers.
 */
List<Container> listContainers()

/**
 * Lists the available containers of WebContainer type.
 * @return the list with the available containers.
 */
List<WebContainer> listWebContainers()

/**
 * Lists the available containers of BusinessContainer
    type.
 * @return the list with the available containers.
 */
List<BusinessContainer> listBusinessContainers()

/**
 * Lists the available containers of DataContainer type.
 * @return the list with the available containers.
 */
List<DataContainer> listDataContainers()
```

4.6.7 *Serializer*

The *serializer package* handles the serialization/de-serialization of objects to/from byte arrays. We use JAXB for all operations.

All data stored to Zookeeper are stored in bytes. As a result, we need to serialize our objects to byte arrays in order to store them and when we retrieve them we need to de-serialize the byte arrays to objects so that we can use them.

We define a class *the JAXBSerializer* which provides the API to handle all serialization/de-serialization operations for our objects.

4.6.8 *Zookeeper*

The *Zookeeper package* contains components necessary for the interaction with Zookeeper.

First of all, we define a class *ConnectionWatcher* that handles the connection to the Zookeeper servers. The class defines two methods:

- *connect*. Establishes a connection with a server of the Zookeeper ensemble.
- *closeSession*. Closes a client session.

Apart from that, we define *the ZkConf* class to:

- hold all the configuration of the application that is stored to Zookeeper.
- store configuration that the tool needs and it is not uploaded to Zookeeper.
- define the application namespace that is created to Zookeeper and initialize its data.
- create *the name id* with which an application is deployed.

Moreover, we use an instance of *the ZkMaster* class for all our interactions with Zookeeper. The ZkMaster is responsible for:

- creating the application namespace to Zookeeper.
- cleaning the application namespace from Zookeeper.
- listing and monitoring the running services of an application.
- providing an interface to issue requests to Zookeeper servers for other objects.

In this package there is also the *ZkNamingService* class that provides methods to facilitate the interaction with the naming service for the application. All data for a service are stored to a *ZkNamingServiceNode*.

In Listing 19 we see the structure of the *zookeeper package*.

Listing 19: Zookeeper package

```
|-- zookeeper
   |-- ZkConf.java
```

```
   |-- ZkConnectionWatcher.java
   |-- ZkMaster.java
   |-- ZkNamingService.java
   |-- ZkNamingServiceNode.java
   |-- ZkNode.java
   |-- ZkSrvConf.java
```

4.6.9 *Docker*

The *Docker package* defines a *DockerInitializer* that creates and initializes a Docker client that is used from the tool for communication with the Docker daemon.

4.6.10 *Broker*

The *broker package* is not to be confused with the Broker module of the tool. We define a *core Broker* class that is used to interact with a container. We use a *BrokerInit* class to initialize, start, manage the Brokers and invoke the appropriate methods depending on the user command.

BrokerInit uses an *executorService* that runs operations of Brokers on different threads. A Broker operation is wrapped to a *Future* object. We wait for all operations to succeed in order to continue execution. There is a timeout period that each task must be executed, so that we don't wait for a task forever. If an unrecoverable error occurs pending operations are cancelled.

All the Brokers contact the Docker daemon to issue requests about containers. They handle the life-cycle of the containers. There is a Broker definition for every container type.

In Listing 20 we see the structure of the *broker package*.

Listing 20: Broker package

```
|-- broker
   |-- BrokerInit.java
   |-- Broker.java
   |-- BusinessBroker.java
   |-- ContainerLifecycle.java
   |-- DataBroker.java
   |-- WebBroker.java
```

4.7 BROKER MODULE

The Broker module is comprised of several sub-systems/packages which will be described in the following sections.

4.7.1 *Boot*

The *boot package* of the Broker module contains the entrypoint for the module. It creates an instance of a *Broker* class which takes control and runs the program.

4.7.2 *Env*

The *env package* is concerned with the *container environment*. The container environment consists of all the environment variables defined in the application description for the container along with all the environment variables of its declared dependencies. The package defines three components/classes that support the required functionality for the environment according to the design.

We define an *EnvironmentMapper* class that will map the schema resource containing the container environment information to a resource that is used by the tool. This mapping provides flexibility in development in case there are changes in the schema resource. As a result, the code does not break and the components are not dependent from the low level details of the schema resources.

We define an *Environment* class that represents a resource for the tool to which the schema resource is mapped. This class encapsulates all the information about the container environment.

Finally, we use a *EnvironmentHandler* that acts on the *Environment* resource. The handler creates the environment that is injected to containers. Because the user customizes the declared container environment at will, we do not know beforehand the fields declared. For that reason, we use *reflection* to get the declared name fields and then we create all the key-value pairs for the environment variables.

In Listing 21 we see the structure of the *env package*.

Listing 21: Env package of Broker module.

```
|-- env
   |-- EnvironmentHandler.java
   |-- Environment.java
```

```
|-- EnvironmentMapper.java
```

In Listing 21 we see the UML diagram of the *EnvironmentHandler*.

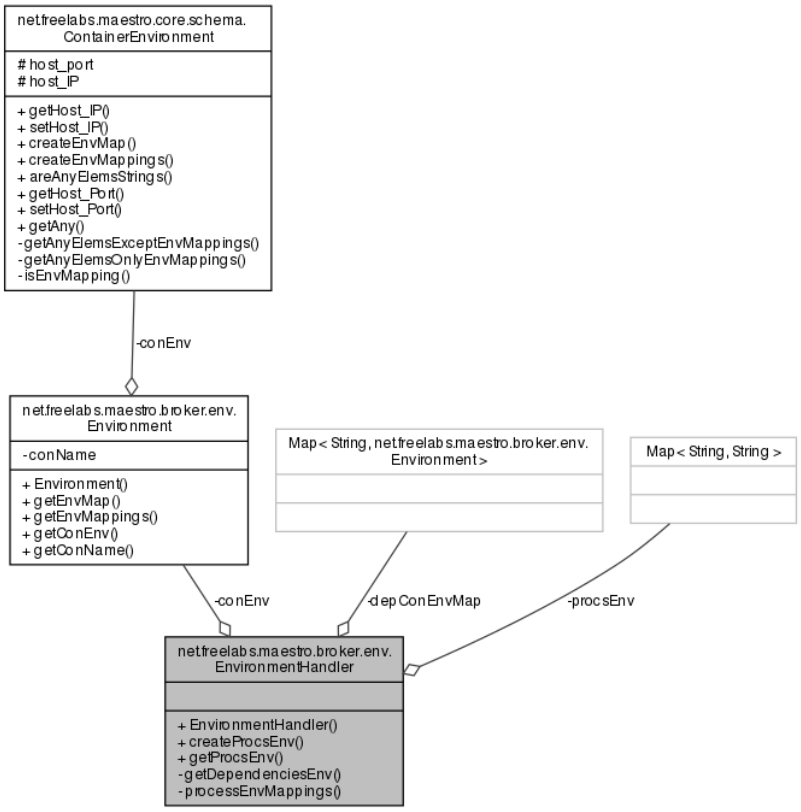

Figure 17: UML diagram of EnvironmentHandler in Broker module.

4.7.3 *Process*

The *process package* defines all the necessary components to support the process execution mechanism of the tool. We will present the mechanism using a bottom-up analysis.

We start by defining the class *ProcessData*. This class holds all the data of a process. The data include *the environment*, all the environment variables that are injected at runtime and *the execution resource*, the cmd/script to execute, its arguments and any properties.

We define the *Executable interface* that is used as a container for code to be executed. It is a functional interface used to pass lambda expressions.

We go on and define a *ProcessHandler*. This is an abstract template class that provides methods to manage the life-cycle of

a process. The handler provides the API to handle a process and its execution. A ProcessHandler is associated always with a ProcessData instance and two objects of type Executable that their code is executed upon success or failure of the process. We provide a concrete implementation of a ProcessHandler *the DefaultProcessHandler* and a more advanced for the main container process *the MainProcessHandler*.

Moreover, according to the design we saw in Section 3.10 we need two process groups. For that purpose, we define an abstract template class *the GroupProcessHandler* which describes how a group of processes are executed. We provide two concrete implementations of this class which will inspect in the sections below.

4.7.3.1 *Start*

The process execution mechanism has a start group, that executes all the required processes to start and initialize the container service. The *start package* contains the classes that are necessary to support the functionality of the start group. For that purpose, we define *the StartGroupProcessHandler* which is a subclass of the GroupProcessHandler. It handles the execution of the start process group.

The ProcessData class is also sub-classed by the *MainProcessData* in order to define additional information necessary to the main container process.

Apart from that, we define *the MainProcMon* class that provides methods to monitor the main process at all states (initialization, start, stop).

4.7.3.2 *Stop*

In the *stop package* we define *the StopGroupProcessHandler* to enable the execution of the stop group processes. This section does not require any custom logic for the main process.

4.7.3.3 *Process Manager*

On top of all these components is *the ProcessManager* class, which is the top-level component of the process execution mechanism, enabling the process execution of all groups. In Figure 18 we can see the UML diagram of the ProcessManager that shows its internal structure and the connections between its components.

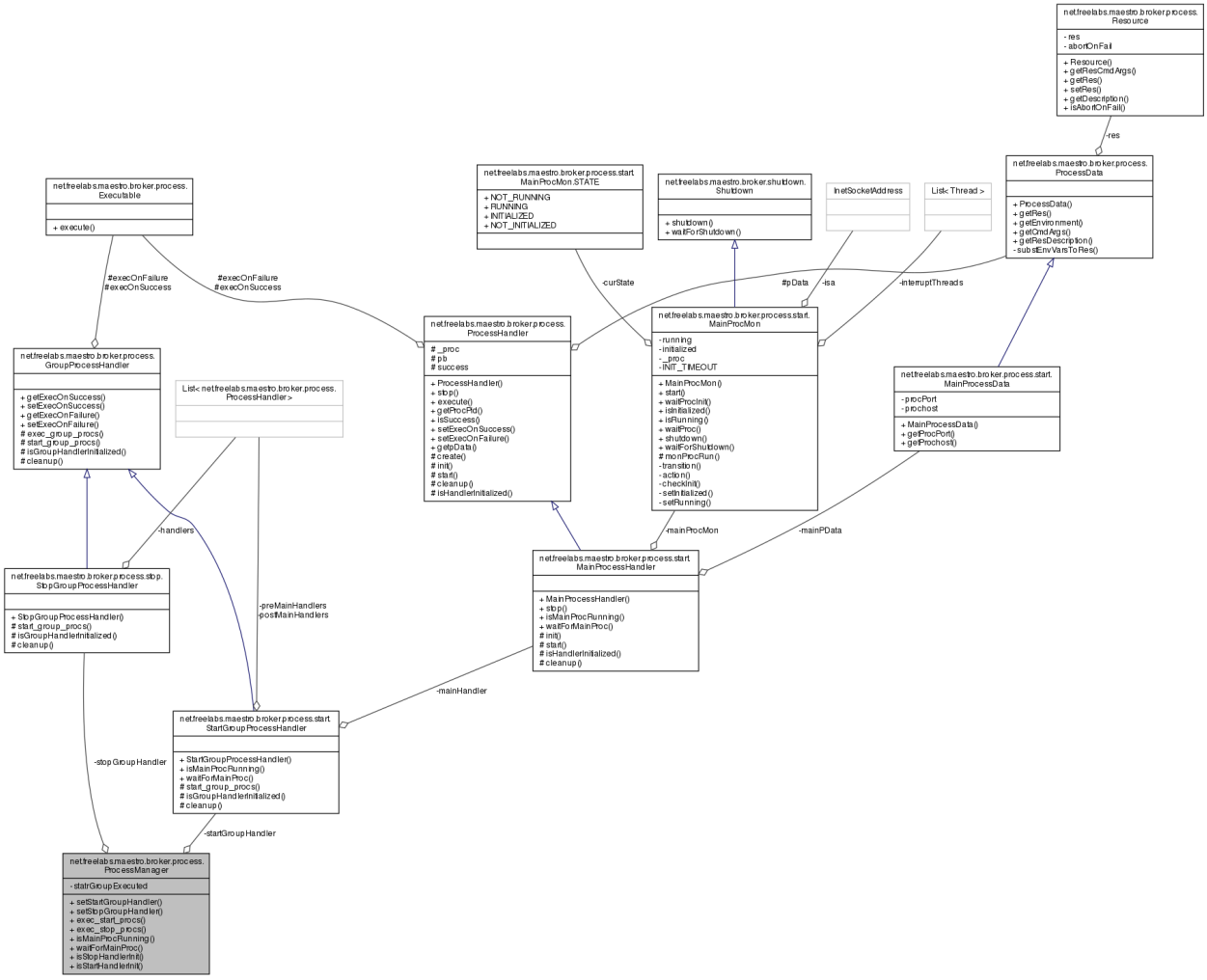

Figure 18: The uml diagram of ProcessManager

In Listing 22 we see the structure of the *process package*.

Listing 22: Process package of Broker module.

```
|-- process
    |-- DefaultProcessHandler.java
    |-- Executable.java
    |-- GroupProcessHandler.java
    |-- ProcessData.java
    |-- ProcessHandler.java
    |-- ProcessManager.java
    |-- Resource.java
    |-- ResourceMapper.java
    |-- start
        |-- MainProcessData.java
        |-- MainProcessHandler.java
        |-- MainProcMon.java
```

```
        |-- StartGroupProcessHandler.java
        |-- StartResMapper.java
    |-- stop
        |-- StopGroupProcessHandler.java
        |-- StopResMapper.java
```

4.7.4 *Services*

The *services package* maintains control over the services required by the container. The container needs the configuration from the services in order to initialize and also to ensure that the required services are initialized (not just running) in order to start the main container process.

We define *the ServiceManager* class that encapsulates all the information about the service-dependencies of the container. The service manager is queried for information about the required services. It maintains a configuration status that indicates if the configuration of a dependency was processed and a service status that indicates if the dependency is initialized.

When all required services are processed and initialized the execution of the start process group of the container is initiated.

In Listing 23 we see the structure of the *services package*.

Listing 23: Services package of Broker module.

```
|-- services
    |-- ServiceManager.java
    |-- ServiceNode.java
```

4.7.5 *Tasks*

The *tasks* package contains the components that provide the functionality for tasks. We support one task that substitutes environment variables to configuration files. Edited files can also be restored to their previous state, if we want to. We define a *TaskHandler* that provides an API to add and execute defined tasks. We provide *the Task interface* that must be implemented from classes whose instances provide a task.

In Listing 24 we see the structure of the *tasks package*.

Listing 24: Tasks package of Broker module.

```
|-- tasks
```

```
    |-- RestoreFilesTask.java
    |-- SubstEnvTask.java
    |-- TaskCreator.java
    |-- TaskHandler.java
    |-- Task.java
    |-- TaskMapper.java
```

4.7.6 *Shutdown*

The *shutdown* package defines an interface that must be implemented from classes that need to perform shutdown operations. The interface defines two methods:

- *shutdown*. Contains the logic to perform all necessary operations (release resources, signal threads to terminate e.t.c.) for normal shutdown.
- *waitForShutdown*. Blocks and waits for shutdown.

Moreover, we define a *ShutdownNotifier* class that is used to give us more control on the shutdown process for multiple components such as when to notify them to initiate shutdown.

In Listing 25 we see the structure of the *shutdown package*.

Listing 25: Shutdown package of Broker module.

```
|-- shutdown
    |-- Shutdown.java
    |-- ShutdownNotifier.java
```

5

CONCLUSIONS / FUTURE WORK

Alone we can do so little;
together we can do so much.

— Helen Keller

5.1 CONCLUSIONS

In this Thesis we designed and implemented a tool that enables the user to manage multi-container web applications in single-host and cluster environments. In Section 1.2 we analyzed thoroughly the complexity of such an approach along with all the services that must be supported in order to provide a complete and robust solution. Existing tools do not address this matter adequately and only provide a portion of the required services leaving the rest to be handled by the user. On the contrary, as we have seen we have managed to offer all the capabilities that provide for a functional solution:

- *service discovery*. Services are able to discover required services, monitor their health and get the required configuration to enable connectivity.
- *service coordination*. Synchronization between dependent services is decentralized and handled transparently, providing high-quality guarantees to support application and data consistency.
- *configuration management*. Configuring containers is facilitated through a container agnostic design. We provide a fully customized environment along with automated tasks to decouple configuration from service instances. In addition, the application configuration is stored in a distributed configuration store which provides control and flexibility in order to implement more advanced operations.
- *integrated process execution mechanism*. The life-cycle of a container service is easily handled with an execution engine which facilitates process execution and provides control at all stages.

- *container life-cycle management*. The life-cycle of all the containers that comprise a containerized web application is automatically managed by our tool, providing simple commands with which the user accomplishes each task.
- *container data management*. Containers can easily store persistent data, share data with other containers or hosts and copy data from hosts.
- *container networking*. Network settings for containers manage the exposure of a container service to the host while containers of an application are deployed on custom networks for isolation.
- *swarm cluster support*. Deploying containers of an application to swarm clusters is handled transparently.

5.2 FUTURE WORK

5.2.1 *Web User Interface*

The tool provides a simple command line user interface in order to manage the containerized web applications. What is more, the containers of a web application that we want to deploy, are described in an xml file. However, users react better on visualized information.

We propose the implementation of a web-based user interface for the tool. First of all, it will facilitate the container declarations for an application. The user will use the interface to select the required fields and set data. Validating data will be performed at the time of input and not at runtime anymore. Furthermore, visualization of information concerning the deployed containers of an application can be easily supported. Finally, managing the life-cycle of many applications can be greatly simplified.

5.2.2 *Dynamic re-configuration*

All services are initialized using configuration information. During their life-cycle, services often need to update their settings. Distributed services though, usually have dependencies from other services. As a result, they need access to the configuration information of the services they depend on in order to be able to connect and communicate or perform more advanced

tasks. When a service updates its configuration, dependent services must be notified, download the new configuration and react accordingly to the change of state based on their current context. This behavior is called dynamic re-configuration.

Fidelio provides configuration management and coordination of distributed services but the containerized services of an application cannot react on the updated state of a service. We propose the further design and implementation of a mechanism that will enable dynamic re-configuration. Some necessary features are already integrated, such as notifications and monitoring of services's state. The description of the web application must be enriched, defining a separate section that will describe the necessary steps the container service must take when a service dependency is updated. In addition, the new section has to be integrated into the process execution mechanism to enable its execution. Concluding, this is a very powerful feature that will enable deployed applications to dynamically adjust on changes in a distributed manner.

5.2.3 *Container settings*

The Docker platform, by leveraging the Linux Kernel technologies, provides many configuration settings for the containers in order to create a fully customized runtime environment. Fidelio encapsulates a small portion of these settings, as it is not its purpose to offer the best possible integration with all the available container settings.

On a future approach, we propose the integration to the tool of more container configuration settings. This can be achieved by customizing the xml schema to support those declarations and enabling their definition in the application description. Due to our clean design the required additions to the code can be easily supported.

5.2.4 *Cluster platforms*

Fidelio transparently supports the deployment of containers in swarm cluster environments. Docker Swarm offers native clustering for Docker hosts. However, other cluster platforms have started to emerge such as Kubernetes and Mesos.

A future task would be to enable support for other cluster platforms. This will increase the tool's usefulness to the end

user and also provide a standard means of managing multi-container web applications on different clusters.

5.2.5 *Network customization*

Docker user defined networks is a relatively new addition to the container platform. Docker offers two drivers for user defined networks and Fidelio supports both. Network plugins have already been developed to support custom features for container networking.

As a future addition to the tool, we propose the support of network plugins for containers. It will enhance the network capabilities of applications and provide users with more flexibility on creating the desired network topologies and environments for their applications.

5.2.6 *Data management*

Fidelio supports container data management in respect to data persistence and sharing with Docker volumes. Data management is a very crucial aspect when deploying containerized distributed applications.

We propose to further integrate the tool with volume plugins. This addition will account for greater flexibility on how application data are persisted, shared and accessed providing custom solutions to the user when required.

Part III

APPENDIX

A

APPENDIX

A.1 FIDELIO CONFIGURATION FILE

The general structure of the configuration file, along with the fields per section, is:

1. *Zookeeper*: This section contains the necessary information to establish communication with the Zookeeper ensemble. The fields are:
   - *zk.hosts* the server list for the Zookeeper servers, in the format *ip:port,ip:port,...*
   - *zk.session.timeout* the client session timeout in ms.
2. *Xml Schema*: Here we specify an xml schema file that will validate the xml application description file. The field is:
   - *xml.schema.path*.
3. *Docker*: All initialization information concerning the Docker platform is specified here. Available fields are:
   - *docker.host* the Docker Host URL e.g. tcp://ip:port or unix:///socket.
   - *docker.tls.verify* (optional -default value is false if not set) enable/disable TLS verification (switch between http and https protocol).
   - *docker.cert.path* (optional -required if TLS enabled) path to the certificates needed for TLS verification.
   - *docker.config* (optional) path for additional docker configuration files (like .dockercfg).
   - *docker.api.version* (optional) the API version e.g. 1.21.
   - *docker.registry.url* (optional) the url of a private image registry.
   - *docker.registry.username* (optional) the username for the registry (required to push containers).
   - *docker.registry.password* (optional) the password for the registry.
   - *docker.registry.mail* (optional) the registry mail.

4. *Logging*: The tool uses the log4j framework for logging, so a file defining a log4j configuration is expected. The field is:
    - *log4j.properties.path*.

An example of the program's configuration is shown at Listing 26

Listing 26: Fidelio.properties file example.

```
# ----------------- Mandatory Fidelio Conf -----------------
# ZOOKEEPER CONF
zk.hosts=192.168.1.6:2181
zk.session.timeout=5000

# XML SCHEMA
xml.schema.path=/home/fidelio/schema.xsd

# DOCKER CONF
docker.host=tcp://192.168.1.6:4376

# LOG CONF
log4j.properties.path=/home/fidelio/log4j.properties

# ----------------- Optional Fidelio Conf -----------------
# enable/disable TLS verification
docker.tls.verify=true
# Path to the certificates needed for TLS verification
docker.cert.path=/home/fidelio/certs
# path to additional docker conf files (like .dockercfg)
docker.config=
# The API version, e.g. 1.21.
docker.api.version=1.22
# Your registry's address.
docker.registry.url=http://192.168.1.6:5000
# Your registry username (required to push containers).
docker.registry.username=user
# Your registry password.
docker.registry.password=pass
# Your registry email.
docker.registry.email=mail
```

A.2 FIDELIO COMMAND LINE INTERFACE

Below we present the command line interface of Fidelio:

Listing 27: Fidelio Command Line Interface.

```
Usage: fidelio [options] [command] [command options]
  Options:
    -c, --conf
       <program conf> Path to .properties file with program
           configuration.
    -d, --dockerOpts[]
       Docker options [host=<tcp://ip or unix:///socket>,tls
           =<true to enable
       https>,cert=<path to certs for tls>,config=<path to
           config like .dockercfg>,api=<api
       version>,regUrl=<registry url>,regUser=<regisrty
           username>,regPass=<registry
       password>,regEmail=<registry email>]
    -h, --help
       Show this help message :).
    -l, --log
       <log4j properties> Path to log4j.properties file with
            log configuration.
    -v, --version
       Show program version.
    -z, --zkOpts[]
       Zookeeper options [hosts=<host1:port1,host2:port
           2...>,timeout=<zk client
       session timeout>]
  Commands:
    start      Start application deployment.
      Usage: start [options]
        Options:
          -h, --help
             Help for start command.
          -x, --xmlFile
             <app xml file> Path to application description
                 xml file.
          -s, --xmlSchema
             <schema file> Path to xml schema file.

    stop      Stop deployed application.
      Usage: stop [options] <app id> The id of the deployed
          application to stop.
        Options:
          -h, --help
             Help for stop command.

    restart      Restart deployed application.
```

```
      Usage: restart [options] <app id> The id of the
          deployed application to restart.
        Options:
          -h, --help
             Help for restart command.

    delete      Delete deployed application (zookeeper
        namespace and docker containers).
      Usage: delete [options] <app id> The id of the
          deployed application to delete.
        Options:
          -h, --help
             Help for delete command.
```

A.3 CONTAINER DESCRIPTION FORMAT

A container description follows the following format.

Listing 28: Container description.

```
<ContainerType>
    <serviceName>
    <requires>
    <docker>
        <image>
        <volumes>
        <volumesFrom>
        <bindMnt accessMode="rw/r">
            <hostPath>
            <containerPath>
        </bindMnt>
        <copy withRootDir="true/false">
            <hostPath>
            <containerPath>
        </copy>
        <publishPort protocol="tcp/udp">
            <hostIp>
            <hostPort>
            <containerPort>
        </publishPort>
        <publishAllPorts>
        <privileged>
    </docker>
    <start>
        <preMain abortOnFail="true/false">
```

```
        <main>
        <postMain abortOnFail="true/false">
    </start>
    <stop>
        <preMain>
        <main>
        <postMain>
    </stop>
    <tasks>
        <substEnv>
            <filePath restoreOnExit="true/false">
        </substEnv>
    </tasks>
    <env>
        <host_port>
        <env_declaration>
    </env>
</ContainerType>
```

BIBLIOGRAPHY

[1] Gnu general public license, version 3. URL http://www.gnu.org/licenses/gpl.html.

[2] Jonathan Corbet. Seccomp and sandboxing, May 2009. URL https://lwn.net/Articles/332974/.

[3] Jendrock Eric, Cervera-Navarro Ricardo, Evans Ian, Haase Kim, and Markito William. Java Platform, Enterprise Edition: The Java EE Tutorial, Release 7, September 2014. URL https://docs.oracle.com/javaee/7/tutorial/overview003.htm.

[4] Free Software Foundation. What does it mean to say that two licenses are "compatible"? URL https://www.gnu.org/licenses/gpl-faq.html#WhatDoesCompatMean.

[5] Antonio Goncalves. *Beginning Java EE 7*. Books for professionals by professionals. Apress, 2013. ISBN 9781430246268.

[6] Matt Helsley. LXC: Linux container tools, February 2009. URL http://www.ibm.com/developerworks/library/l-lxc-containers.

[7] F. Junqueira and B. Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013. ISBN 9781449361266.

[8] M. Kerrisk. *The Linux Programming Interface*. No Starch Press Series. No Starch Press, 2010. ISBN 9781593272203.

[9] Dionysis Lappas. Code constributions. URL https://github.com/denlap007/docker-java/commits?author=denlap007.

[10] The Linux man-pages project. Namespaces - overview of linux namespaces, March 2016. URL http://man7.org/linux/man-pages/man7/namespaces.7.html.

[11] R.C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 2008. ISBN 9780136083252.

[12] K. Morris. *Infrastructure As Code: Managing Servers in the Cloud*. Oreilly & Associates Incorporated, 2016. ISBN 9781491924358.

[13] Adrian Mouat. *Using Docker*. O'Reilly Media, 2016. ISBN 9781491915769.

[14] J. Nickoloff. *Docker in Action*. Manning Publications Company, 2016. ISBN 9781633430235.

[15] Ondrejka Peter, Majoršinová Eva, Prpič Martin, Landmann Rüdiger, and Silas Douglas. Resource management guide, 2016. URL https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/index.html.

[16] A. Puder, K. Römer, and F. Pilhofer. *Distributed Systems Architecture: A Middleware Approach*. The MK/OMG Press. Elsevier Science, 2011. ISBN 9780080454702.

[17] Flavien Quesnel. *Scheduling of Large-scale Virtualized Infrastructures: Toward Cooperative Management*. FOCUS Series. Wiley, 2014. ISBN 9781118790106.

[18] Chris Richardson. Service Discovery in a Microservices Architecture, October 2015. URL http://www.ibm.com/developerworks/library/l-lxc-containers. [Online; accessed 02-June-2016].

[19] J. Sebastian. *The Art of XSD: SQL Server XML Schema Collections*. High performance SQL server. Red Gate Books, 2009. ISBN 9781906434175.

[20] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2015. ISBN 9781491901700.