BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

TESTING APPLICATIONS USING LINUX CONTAINERS

TESTOVÁNÍ APLIKACÍ S VYUŽITÍM LINUXOVÝCH KONTEJNERŮ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR
AUTOR PRÁCE

Bc. MATÚŠ MARHEFKA

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. ALEŠ SMRČKA, Ph.D.

BRNO 2016

Zadání diplomové práce/18574/2015/xmarhe00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Marhefka Matúš, Bc.**

Obor: Počítačové sítě a komunikace

Téma: **Testování aplikací s využitím Linuxových kontejnerů**

Testing Applications Using Linux Containers

Kategorie: Analýza a testování softwaru

Pokyny:

1. Nastudujte Linuxové kontejnery a technologie s nimi související (jmenné prostory, kontrolní skupiny, SELinux, apod.). Nastudujte technologii Docker pro administraci a běh Linuxových kontejnerů. Nastudujte testování softwaru na úrovni integračních a systémových testů.
2. Analyzujte případy užití kontejnerů pro běh různých druhů softwarových systémů (např. webové, databázové a systémové servery) v rámci technologie Docker.
3. Navrhněte metodu testování funkcionality softwaru, která bude využívat výhod Linuxových kontejnerů. Zaměřte se na úroveň systémového a integračního testování.
4. Implementujte testovací framework využívající vámi navrženou metodu s technologií Docker.
5. Demonstrujte testovací framework na testování netriviálního softwarového systému.

Literatura:

- Rosen, R.: Linux Containers and the Future Cloud. Linux Journal. 2014-06-10. URL: http://www.linuxjournal.com/content/linux-containers-and-future-cloud
- Domovská stránka projektu Docker. URL: https://www.docker.com/
- P. Ammann, J. Offutt. *Introduction to Software Testing*, Cambridge University Press, 2008. ISBN 978-0-511-39330-3.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání a část návrhu testování.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT

Konzultant: Beneš Eduard, Ing., RHcz

Datum zadání: 1. listopadu 2015

Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček

vedoucí ústavu

Abstract

This thesis discusses software containers (Docker containers in particular), as a variant of server virtualization. Instead of virtualizing hardware, software containers rest on top of a single operating system instance and are much more efficient than hypervisors in system resource terms. Docker containers make it easy to package and ship applications, and guarantee that applications will always run the same, regardless of the environment they are running in. There is a whole range of use cases of containers, this work examines their usage in the field of software testing. The thesis proposes three main use case categories for running software systems in Docker containers. It introduces aspects for applications running in containers, which should give a better overview about an application setting within containers infrastructure. Subsequently, possible issues with testing software systems running inside Docker containers are discussed and the testing methods which address the presented issues are proposed. One proposed testing method was also used in the implementation of the framework for testing software running in Docker containers which was developed within this work.

Abstrakt

Táto diplomová práca sa zaoberá softwarovými kontajnermi (obzvlášť Docker kontajnermi) ako jednou variantou serverovej virtualizácie. Softwarové kontajnery namiesto virtualizácie hardwaru spočívajú na vrchole jedinej inštancie operačného systému a sú z hľadiska systémových zdrojov oveľa účinnejšie ako hypervisory. Docker kontajnery uľahčujú balenie a distribúciu aplikácií, a zaručujú, že aplikácie budú vždy bežať rovnako, bez ohľadu na prostredie, v ktorom budú spustené. K dispozícii je celý rad prípadov použitia kontajnerov, táto práca skúma ich použitie v oblasti testovania softwaru. Diplomová práca navrhuje tri hlavné kategórie prípadov použitia pre beh softwarových systémov v Docker kontajneroch. Predstavuje aspekty pre aplikácie bežiace v kontajneroch, ktoré by mali poskytnúť lepší prehľad o nastaveniach aplikácie v rámci infraštruktúry kontajnerov. Následne sú diskutované možné problémy s testovaním softwarových systémov bežiacich v Docker kontajneroch a sú navrhnuté testovacie metódy, ktoré predložené problémy riešia. Jedna navrhnutá testovacia metóda bola tiež použitá pri implementácii frameworku na testovanie softwaru bežiaceho v Docker kontajneroch, ktorý bol vyvinutý v rámci tejto práce.

Keywords

software testing, virtualization, containers, Docker, Linux, test environment.

Klíčová slova

testování softwaru, virtualizace, kontejnery, Docker, Linux, testovací prostředí.

Reference

MARHEFKA, Matúš. *Testing applications using Linux containers*. Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Smrčka Aleš.

Testing applications using Linux containers

Declaration

Hereby I declare that this diploma thesis was prepared as an original author's work under the supervision of Mr. Ing. Aleš Smrčka, Ph.D. and the supplementary information was provided by Mr. Ing. Eduard Beneš. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Matúš Marhefka
May 24, 2016

Acknowledgements

I would like to thank my thesis advisor, Ing. Aleš Smrčka, Ph.D. for the consultations and assistance with this work. Special thanks to Ing. Eduard Beneš for his help, advices and support considering mainly technical aspects of this work. Finally, I want to thank my family for their constant support.

Contents

Chapter 1

Introduction

This work discusses field of testing applications running in software containers which are based on the *Docker* technology. Software containers are a type of server virtualization, a method of separating an application from other applications, from the operating system and the physical infrastructure it uses to connect to the network [30]. The container uses the kernel of the operating system it runs on, and virtualizes the instance of the application. It is a type of application sandboxing but with much less overhead than with standard Virtual Machines (VM). Virtualization technology and differences between these technologies are discussed in the Chapter 3.

Docker is one of used solutions for working with software containers. It provides an infrastructure and automation of deployment of applications inside software containers [9]. Docker uses Linux kernel containment features such as *Cgroups*, namespaces and many more to isolate applications from each other but also from the operating system. Docker project also provides the *Docker Hub*, a cloud-based registry service for building and shipping application or service containers [9]. It provides a centralized resource for searching, distributing and managing containerized applications. More details about the Docker platform and Linux kernel containment features it uses can be found in the Chapter 4.

The goal of this work is to analyze Docker containers use cases for running different kinds of software systems (for example web or database servers), and to propose methods of testing these systems on integration and system level, for their different setups in containers. This work should also provide a guidance for developers and testers to determine if it is appropriate for certain software system, or set of tests to be run inside a software container. The targets of testing in this work are applications running either in a single Docker container or in multiple Docker containers on the same host operating system. This is discussed in the Chapter 5 and Chapter 6.

The Chapter 7 is dedicated to the implementation of the framework which was developed within this work. It describes its architecture, configuration, plugins (which extends the framework functionality) and also its command-line interface. The Chapter 8 then demonstrates the framework workflow when testing applications running in Docker containers. Workflow is demonstrated on two use cases: the use case where a tested software is running in a single Docker container, and the use case where a tested software is running in multiple Docker containers.

As different GNU/Linux distributions use various system and service managers, different Docker versions, configurations and standard paths, this work will focus on the following technologies:

- *Fedora* 22 or later,
- *Docker Engine* 1.10.0 or later,
- *Docker Compose* 1.6.0 or later.

Chapter 2

Overview of Software Testing

Software testing, as defined in the "Guide to the Software Engineering Body of Knowledge – SWEBOK" [25, p. 82], "*consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain*". Testing always implies executing the program on selected inputs and is conducted on a subset of all possible tests (as a complete set of tests can generally be considered infinite in practice). The subset of selected tests is usually determined by risk and prioritization criteria. It must be possible to decide whether the observed outcomes of testing are acceptable or not, otherwise, the testing effort is useless. To verify if the observed behavior of the tested program is expected, this behavior is checked either against user needs (commonly referred to as testing for validation), or against a specification (testing for verification). The fundamental, theoretical limitation of testing is that the problem of finding all failures in a program is undecidable, and thus testing can show only the presence of failures, not their absence. Software testing should be pervasive throughout the entire development and maintenance life cycle of software. Therefore, planning for software testing should start with the early stages of the software requirements process and test plans and procedures should be systematically and continuously developed as software development proceeds to help to highlight potential weaknesses of software. [25, 22]

Information in this chapter is from books "Guide to the Software Engineering Body of Knowledge – SWEBOK" [25] and "Introduction to Software Testing" [22].

2.1 Testing Levels

Software testing is usually performed at different levels throughout the software development and maintenance processes. Tests are grouped by where they are added in this development process. The main levels during the development process which do not imply any process model are:

- *unit testing*—target of the test is a single module;
- *integration testing*—target of the test is a group of modules (related by purpose, use, behavior, or structure);
- *system testing*—target of the test is an entire system;
- *acceptance testing*—validates that the finished product satisfies software requirements.

2.1.1 Unit Testing

Unit testing is designed to assess the units produced by the implementation phase and is the lowest level of testing. Unit testing verifies the functionality of a specific sections of code that are separately testable, usually at the function level. Depending on the context, these could be the individual subprograms or the larger components made of highly cohesive units. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools. Unit testing is mainly the responsibility of the programmer.

2.1.2 Integration Testing

Integration testing is the process of verifying the interactions among software components. It is the phase in software testing in which individual software components are combined and tested as a group. Integration testing assumes that individual components work correctly. It takes components that have been unit tested as its input, groups them in larger subsystems, and assesses whether the interfaces between grouped components in a given subsystem communicate correctly.

2.1.3 System Testing

System testing takes, as its input, all of the software components that have passed integration testing and also the software system itself. It is concerned with testing the behavior of an entire system. System testing is designed to determine whether the assembled system meets its specifications and assumes that the pieces work individually, and asks if the system works as a whole.

System level of testing usually looks for design and specification problems. Effective unit and integration testing will have identified many of the software defects. System testing usually evaluates the nonfunctional system requirements, such as security, performance, accuracy, reliability, or external interfaces to other applications, utilities, hardware devices, or the operating environments.

2.1.4 Acceptance Testing

Acceptance testing should determine if the requirements of a specification are met. It is done after the system testing. Acceptance testing is designed to determine whether the completed software meets the customer’s needs based on the requirements analysis phase of software development. Acceptance testing always involve users, and in contrast with the system testing it verifies that the solution works for the user.

Chapter 3

Introduction to Virtualization Technology

Virtualization is based on the concept that its technology represents an abstraction from physical resources [5]. The two most common types of virtualization used in terms of computing are server virtualization and storage virtualization [5]. Each of these main types of virtualization have more different approaches on how they virtualize physical resources. Every approach has its benefits and drawbacks and when choosing which one to use multiple aspects must be considered as efficiency, utilization of hardware resources, power consumption, maintenance or security.

Source of information in this chapter was book “Virtualization for Dummies” [5] and article “Types of Server Virtualization” [13] when not stated otherwise.

3.1 Server Virtualization

As dedicating a whole server hardware to a single application or task is in most cases uneffective, virtualization started to be used on servers. In server virtualization, software is used to divide the physical server into multiple virtual environments so the resources of the server itself are hidden, or masked from users. These virtual environments are called virtual servers or virtual machines. Each virtual machine runs its own operating system known as “guest operating system” which is typically running onto another operating system called “host operating system” with the help of virtualization software. It is the way of maximizing server resources and effectiveness while improving application service delivery which goals are performance, availability, responsiveness and security.

Server virtualization can be implemented as a standalone software, or in hardware or assisted by hardware, as a component of an operating system or as an application running on an operating system. Three main types of server virtualization are:

- hardware emulation,
- paravirtualization,
- virtualization on operating system level (or software containers).

3.1.1 Hardware Emulation

In hardware emulation, a thin software layer called virtual machine monitor (VMM), or hypervisor, must be used to create a virtual machine (VM) by emulating all the physical hardware required for running an operating system. Each virtual machine has unlimited access to this emulated hardware. Software executed on virtual machines is separated from the underlying hardware resources and runs on top of the guest operating system. Hypervisor catches and identifies traps and responds to protected or privileged CPU instructions called by each virtual machine. It also handles queuing, assigning and returning the results of hardware requests made by virtual machines. Guest operating systems do not need to be modified to run on hypervisors.

Hypervisors are classified to two types [26] (cf. Figure 3.1):

- *type-1* – native or bare-metal hypervisors, run directly on the host's hardware (typically as part of the host's kernel) to control the hardware and to manage guest operating systems. A virtual machine wtih a guest operating system runs as a process on the host.
- *type-2* – hosted hypervisors, run on a conventional operating system just as other processes.

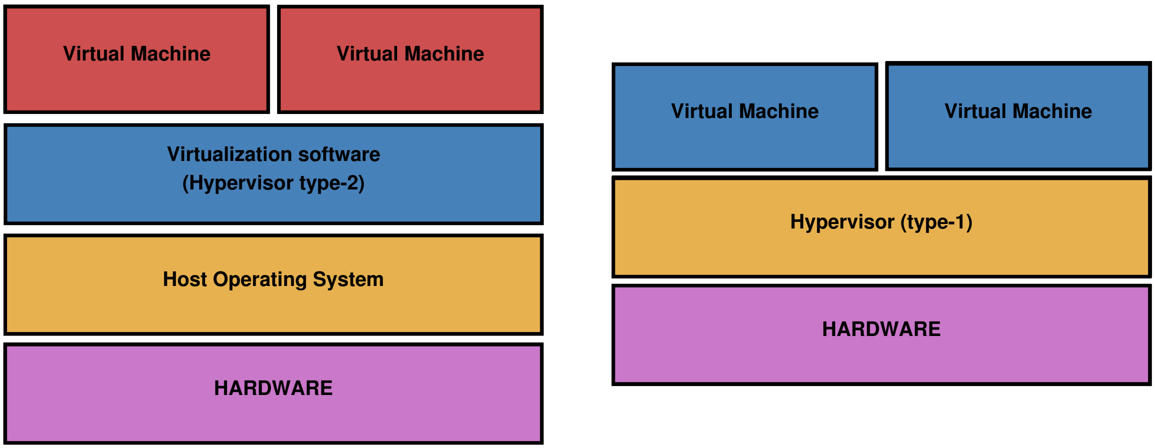

Figure 3.1: Types of hypervisor for hardware emulation virtualization [13].

Another aspect to look at is whether hypervisor software uses *hardware-assisted virtualization* or only *software-based virtualization* [21].

Hardware-assisted virtualization is a virtualization approach using help from physical hardware, primarily from the host system processors. It helps to intercept potentially dangerous operations that a guest operating system may be attempting and also makes it easier to present virtual hardware to a virtual machine. These hardware virtualization features differ between Intel and AMD processors. Intel named its technology *VT-x*, AMD named theirs *AMD-V*. Both technologies offer x86 virtualization instructions for a hypervisor to control ring 0 hardware access (rings are privilege levels provided by hardware) by creating a new "ring -1", so a guest operating system can run ring 0 operations natively without affecting other guests or the underlying host operating system. The CPU flag for VT-x capability is “vmx”, for AMD-V it is “svm” [21]. On the Linux kernel, processor support for the hardware-assisted virtualization can be checked via `/proc/cpuinfo`:

```
$ grep -E "(vmx|svm)" /proc/cpuinfo
```

Software-based virtualization is mainly used in the absence of hardware-assisted virtualization (for example on the older processors). The main problem behind software-based virtualization is that VM’s guest operating system contains many privileged instructions which cannot run natively in ring 0 hardware privilege level and so must be replaced. One of the solutions for this problem in software can be replacing privileged instructions with a jump to a VM-safe equivalent compiled code fragment in hypervisor memory [21]. On the other hand, the guest user-mode code, running in ring 3, can generally run directly on the host hardware in ring 3.

Examples of hardware emulation technologies may include *KVM*, *QEMU* or *Oracle VM VirtualBox*.

Kernel-based Virtual Machine (KVM) is a type-1 hypervisor built into the Linux kernel which can run multiple virtual machines running unmodified operating systems. An associated virtual machines run as user-space processes on top of the Linux kernel and each virtual machine has its private virtualized hardware. KVM requires processor support for hardware-assisted virtualization (Intel VT-x or AMD-V). [20]

QEMU and Oracle VM VirtualBox are a type-2 hypervisors and they can use software-based virtualization, but also supports hardware-assisted virtualization where possible. [20, 21]

3.1.2 Paravirtualization

Paravirtualization is also based on hypervisor virtualization model. It is not emulating a hardware environment in software but rather uses a paravirtualization hypervisor which multiplexes access to the underlying physical hardware environment directly. Paravirtualization hypervisor requires that the guest operating system should be recompiled or modified before installation inside the virtual machine. The modified guest operating system then communicates directly with the hypervisor, and eliminates overheads occurred due emulation and trapping process.

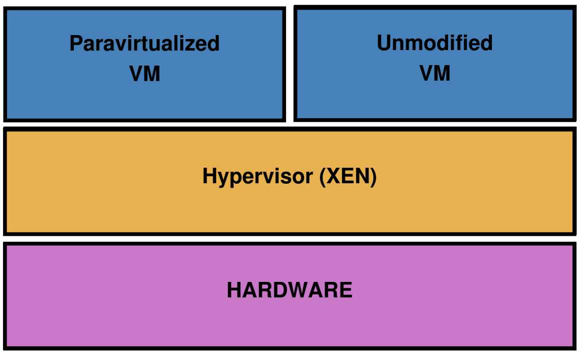

Figure 3.2: Paravirtualization (Xen virtualization stack) [13].

The main open-source representative of paravirtualization software is *Xen*, a type-1 or baremetal hypervisor, which makes it possible to run many instances of an operating system or different operating systems in parallel on a single host machine [33] (Figure 3.2). Xen starts the host in domain 0 (or privileged domain): it is the only domain which, by default, has direct access to hardware and it runs a customized Linux kernel. Once the domain 0 has started, one or more domainU (short for user domains, also called VMs or guests) can be started and controlled from the domain 0 [33]. Processors that support hardware-assisted virtualization make it possible to support unmodified guests in Xen, including proprietary operating systems.

3.1.3 Virtualization on Operating System Level

Hardware emulation and paravirtualization usually provides a high level of isolation and security as all communication between the guest and the host is through the hypervisor. This approach is also usually slower and incurs significant performance overhead due to the need of the hypervisor to access physical hardware.

Another level of virtualization called “operating system level virtualization” or “software containers” or “container virtualization” or “virtual private servers” was introduced to reduce overhead caused by the hypervisor. Virtualization on operating system level uses the operating system’s normal system call interface and do not need to utilize services of the hypervisor software. Because of that a separate kernel and simulating all the hardware is not required for the software containers. This type of virtualization allows running multiple isolated user space instances (instead of just one) on the same operating system kernel (Figure 3.3) and so it is as close as possible to the hardware emulation and the paravirtualization [2].

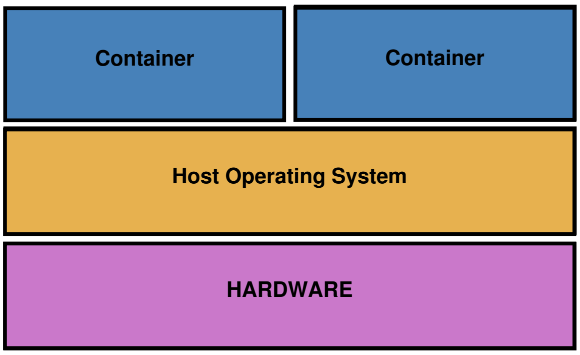

Figure 3.3: Operating system level virtualization [13].

Container virtualization provides an environment that groups and isolates a set of processes or a process and resources such as memory, CPU or disk of these processes from the host operating system and all the other containers running within the operating system. However, sharing the same kernel with the host operating system and also with all the running containers has a set of shortcomings [2]:

- host operating system’s kernel cannot host different guest kernel from itself,
- and the isolation between the host and the container is not as strong as hypervisor-based virtualization (because of sharing the host kernel)

There are many common cases where containers can be used [2]:

- allocating finite hardware resources amongst a large number of users,
- as an operating system or an application packaging mechanism,
- installing, configuring and running different applications or libraries,
- creating testing environments and running tests for different applications or libraries,
- and separating several applications to containers for improved security, application isolation and resource management features.

For Linux, more implementations of operating system level virtualization exist, including Docker, *LXC*, *Linux-VServer* or *OpenVZ*. All can run different Linux distributions in their virtual environments, but all share the same underlying kernel. [34]

3.2 Storage Virtualization

Storage systems provide fast and reliable storage for computing and data processing. In order to do so, they use the physical storage resources (special hardware such as disk drives) aggregated into storage pools and special software for working with these storage pools. The logical storage is then created from storage pools and presented to the application environment. Storage systems can provide either block accessed storage, or file accessed storage.

Storage virtualization is the process of abstracting logical storage from physical storage. It can be implemented within the local storage arrays or at the network level where logical storage is composed of multiple disk arrays from different vendors scattered over the network. This abstraction allows the multiple storage arrays to be managed as if they were a single pool. There are two major types of shared networked storage systems: NAS and SAN systems.

Network-attached storage (NAS) is a storage device which offers storage to multiple clients and servers to share files over the Local Area Network (LAN). NAS is simple to deploy and manage as there is no need to keep track of files spread among multiple machines over the network, all the data is located in one place. Common uses of a network-attached storage are cases like document and backup files, storage with multiple types of data (like media) or e-mail.

A storage area network (SAN) is a storage device which appears as locally attached to the operating system. It usually has its own network of storage devices. Compared to NAS, it does not provide the file accessed storage, only block-level operations. Common uses of a storage area network may include e-mail servers, databases or high usage file servers.

Chapter 4

Docker Platform

Docker is a type of virtualization software used for automation of packaging, deployment, and running applications [9]. It uses the operating system level virtualization (Subsection 3.1.3) features and technologies of the Linux kernel such as Linux kernel namespaces, Cgroups, union capable filesystems and others to allow independent software containers to run within a single Linux instance without the need of starting and maintaining virtual machines.

As Docker provides application portability by encapsulating applications within containers, it started to be used for cloud computing [8]. The environment it provides for an application abstracts it from the underlying platform and this environment is also portable from platform to platform. There are many possibilities how Docker can be used, including load sharing, an application isolation, the ability to provide server reuse for more applications, or creating testing environments [8].

Information in this chapter is mainly from official Docker documentation [9] if not stated otherwise.

4.1 Architecture of Docker Platform

Docker works by combining a lightweight container virtualization with workflows and tooling that helps to manage and deploy applications. Applications run isolated in a container because of the container virtualization platform. The isolation allows it to run many containers simultaneously on a target host system.

Tooling and a platform were developed to help with:

- packaging applications and supporting components into Docker containers,
- and distributing Docker containers to production environments.

Docker consists of:

- Docker Engine: the container virtualization platform.
- Docker Hub: the platform for sharing and managing Docker containers.

Docker uses a client-server architecture. The Docker client talks to the Docker daemon, which does the building, running, and distributing of Docker containers. The Docker client and daemon communicate via sockets and they can run either on the same operating system (Figure 4.1), or they can be connected in such a way that a Docker client connects to a remote Docker daemon.

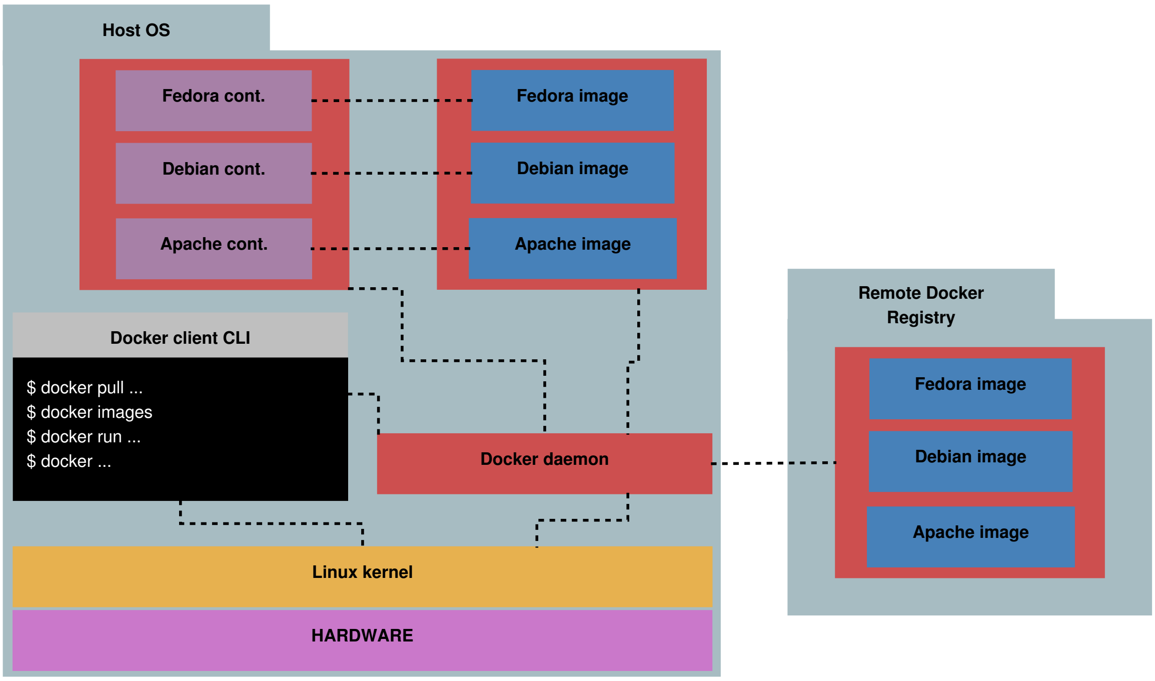

Figure 4.1: Docker architecture with Docker client and daemon on the same OS.

The main three components which Docker works with are:

- Docker image—A static part of Docker platform holding application files.
- Docker registry—A storage for Docker images.
- Docker container—A dynamic part of Docker platform created from a Docker image for running application.

4.1.1 Docker Images

A Docker image is a read-only template. Images are used to create Docker containers. They hold static application files and its supporting components required for running the packaged application.

Each image consists of multiple layers. Docker uses union file systems to combine these layers into a single image. Files and directories of separate file systems (also called branches) can form a single coherent file system with files and directories transparently overlaid. Collisions of files with the same names are resolved by layer age: a file from a newer layer will overlap a file from older layers.

When making changes or updating existing image, a new layer gets built. It is the main reason why distribution of Docker images is lightweight and fast where instead of replacing or rebuilding the whole image, only a new layer is added or updated. It is very similar to revision control systems like *git*, which record changes to the repository in form of snapshots.

Docker provides a simple way to build new images or update existing images, which can be done automatically by reading the instructions from a Dockerfile. The Dockerfile is a text document. It contains all the instructions (or commands) a user could call on the command line to assemble an image. Each instruction creates a new layer in an image. Instructions may include actions like: run a command, add a file or directory, create an environment variable, what process to run when launching a container from the image, or others.

Every image starts from a base image. For example, base image may contain a Fedora operating system and multiple added layers with some web server application on top of it. Docker images are the build component of Docker and they can be built, or downloaded from Docker registries.

4.1.2 Docker Registries

Docker registries are storages for images. These may be public or private stores where images can be uploaded or downloaded from (Figure 4.1).

The Docker Hub is the public Docker registry which serves a collection of existing images for usage. Users can upload their own images or pull existing ones from the Docker Hub, or they can use their own registry. Docker registries are the distribution component of Docker.

4.1.3 Docker Containers

Each Docker container is created from a Docker image and is holding everything that is needed for an application to run. Docker containers are the run component of Docker and they can be run, stopped, started, moved, and deleted. Containers are creating isolated and secure environment for applications.

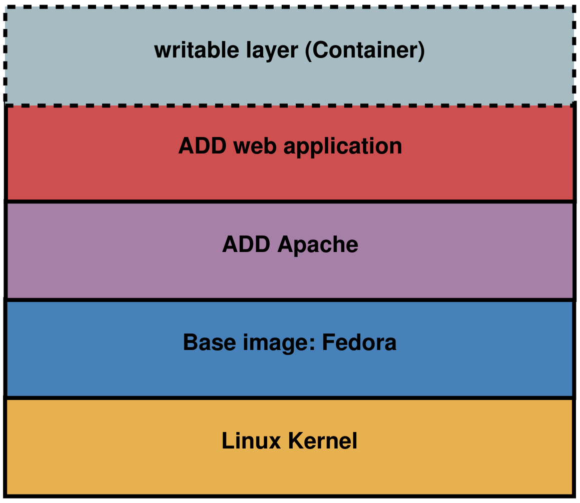

Figure 4.2: Docker container layers.

A container consists of an operating system's kernel, user-added files, and meta-data. Docker image contains information about what process to run when the container is launched and many other configuration data. As Docker image is read-only, when Docker runs a container from an image, it must add a read-write layer on top of that image (using union filesystems) in which an application can then run (Figure 4.2).

4.1.4 Docker Daemon

All interactions with the daemon are not direct but through the Docker client. The Docker daemon is the persistent process that manages containers. It can listen for Docker Remote API requests over three different types of sockets: unix, tcp, and file descriptor.

By default the Docker daemon listens on `unix:///var/run/docker.sock`, requiring either root permission, or docker group membership, which can be acquired the following way:

```
$ sudo groupadd docker
$ sudo chown root:docker /var/run/docker.sock
$ sudo usermod -a -G docker $USER
```

When accessing the Docker daemon remotely, the tcp socket needs to be enabled which by default provides un-encrypted and un-authenticated direct access to the Docker daemon, and should be secured either by the built in HTTPS encrypted socket, or by putting a secure web proxy in front of it.

System managers such as *SysVinit*, *Upstart*, or *systemd* can be used to manage the Docker daemon's start and stop. Configuration of the Docker daemon for systemd process manager can be found and modified in the `/etc/sysconfig/docker` file on the host operating system by specifying values in variables. More on the Docker daemon configuration can be found in *Docker Docs* [9].

4.1.5 Command Line Interface of Docker client

Command line interface (CLI) of Docker client is the primary user interface to Docker. Its main goal is to communicate with a Docker daemon and to accept commands issued by the user. Client can be used to create and remove images, run, pause or stop containers, list containers, inspect containers and many more.

It is the single binary and it is the main tooling for packaging applications into Docker containers, and for deploying and managing these containers in production environments.

4.1.6 Docker Machine

Docker also makes it possible to run Docker containers on the other operating systems like *Microsoft Windows* or *Apple Mac OS X*. Operating system level virtualiation uses operating system's normal system call interface instead of using the hypervisor software. This means that Docker cannot be run natively on these operating systems, as a Docker daemon uses the Linux-specific kernel features. Instead, `docker-machine` command must be used to create and attach to a virtual machine. This machine is a Linux virtual machine that hosts Docker on the target operating system. Information on how to run this virtual machine can be found in Docker Docs [9].

4.1.7 Running a Container

Running a container is done by calling the Docker client with the run option telling it to launch a new container:

```
$ docker run -it fedora /bin/bash
```

To run a container, the Docker client tells the Docker daemon what image to build the container from (in this case it is a base Fedora image) and the command to run inside the container when it is launched (in this case `/bin/bash` to start the *Bash* shell inside the new container).

The Docker daemon does all the work on background which for the above example (“`docker run -it fedora /bin/bash`”) includes the following:

1. Pulls the Fedora image from the Docker Hub, if it doesn’t exists locally on the host system; if the image already exists, then Docker uses it for the new container.
2. Creates a new container from the pulled image.
3. Allocates a file system in which the container is created and mounts a read-write layer (a read-write layer is added to the image).
4. Creates a network interface that allows the Docker container to talk to the local host.
5. Attaches an available IP address from available addresses pool to the container’s new network interface.
6. Executes a process specified on launching the image (in this example `/bin/bash`).
7. Captures and provides an application output: standard input, output and error which can be used to monitor an application running in the container.

As of now the container is running a specified application (`/bin/bash`) within itself and it is possible to interact with that application, or manage it through the Docker client binary.

4.1.8 Container Links

Link system allows a source container to provide information about itself to a recipient container. For example, the recipient container can be running some web application, and can access information about the source container, in which data storage system or database system may be running. One of the benefits of linking is that the source container doesn’t need to be exposed to the external network at all. To be more precise, the source container will be visible only to containers running on the same host operating system as, by default, all the containers communicate through a private networking interface. Example with a web application and a database can be implemented the following way:

```
$ docker run -d --name db some_db_image
$ docker run -d -p 80:80 --name web --link db some_webapp_image
```

which will run both, the web application and the database containers and will link them with each other, which allows them to communicate and exchange data. Only the web application's container is exposed to the external network.

Linking containers is done by exposing connectivity information of the source container to the recipient container in two ways:

- Environment variables,
- Updating the `/etc/hosts` file.

More details can be found in "Legacy container links" section in Docker Docs [9].

4.1.9 Data Volumes

Data volume is a special directory within one or more containers designed to persist data, independent of the container's life cycle. Therefore, Docker never automatically deletes volumes when containers are removed. Volumes are initialized when containers are created and can be shared and reused among two or more containers. Changes to a data volume are made directly.

Adding a data volume to a container is done using the "`-v`" option of the `docker-create` or the `docker-run` command:

```
$ docker run --rm -dt -v /dir --name fedora_cont fedora /bin/bash
$ docker inspect fedora_cont
    ...
    "Mounts": [
        {
            "Name": "5f87d...68571",
            "Source": "/var/lib/docker/volumes/5f87d...68571/_data",
            "Destination": "/dir",
            "Driver": "local",
            "Mode": "",
            "RW": true
        }
    ],
    ...
```

which will create a new volume inside a container at `/dir`, "Source" is specifying the location on the host and "Destination" is specifying the volume location inside the container.

It is also possible to mount a directory from the host operating system into a container, where path to a container directory must always be an absolute path:

```
$ docker run --rm -it -v /opt:/dir --name fedora_cont fedora /bin/bash
```

This will mount the host directory `/opt` into the container at `/dir`. If the path `/dir` already exists inside the container's image, the `/opt` mount overlays but does not remove the pre-existing content of the `/dir` directory.

Data sharing between containers can also be achieved using a data volume container. This type of container does not run an application, it only creates a new container:

```
$ docker create -v /dir --name fedora_data_cont fedora /bin/true
```

Volume `/dir` can be then mounted in another container which can be then safely removed, while data will stay untouched inside the “`fedora_data_cont`” container:

```
$ docker run --rm -it --volumes-from fedora_data_cont fedora /bin/bash
```

4.1.10 Labels

Adding metadata to Docker images, containers, or daemons is possible via labels. A label is a <key> / <value> pair. Docker stores the label values as strings. It is possible to specify multiple labels but each <key> must be unique or the value will be overwritten. Labels can be used for adding notes or any other information to an image or to a container. The following command will add label with the <key> “*com.example.name*” and value “*test*” to a container:

```
$ docker run --rm -it -l "com.example.name=test" fedora /bin/bash
```

To list all running containers that have the “*com.example.name=test*” label:

```
$ docker ps --filter "label=com.example.name=test"
```

4.2 Containment Features and Technologies of the Linux Kernel

Docker is written in the *Go* programming language which is designed primarily for systems programming, it is a compiled, statically typed language with garbage collection and various safety features [15]. Docker makes use of several Linux kernel features and technologies to deliver the required functionality. This section provides overview of these containment features and technologies which are used to create operating system level virtualization or software containers (Subsection 3.1.3) within the Linux kernel. Used features and technologies have certain capabilities and limitations which apply to applications and tests running inside containers.

4.2.1 Linux Capabilities

Linux capabilities provide control over superuser permissions, allowing use of the root user to be avoided. Minimal set of capabilities can be used to replace the powerful setuid attribute in a system binary. Example of such a binary might be `ping(8)`, which, instead of using the setuid method, is using the Linux capability `CAP_NET_RAW`. This allows `ping(8)` to be run by a normal user, while at the same time limiting the security consequences of a potential vulnerability in `ping` binary. To find out Linux capabilities of a binary, `getcap(8)` can be used:

```
$ getcap /usr/bin/ping
/usr/bin/ping = cap_net_admin,cap_net_raw+ep
```

In Linux, capabilities are a per-thread attribute. [19, `capabilities(7)`]

Docker defines the standard set of Linux capabilities that are set in a container in its “Container Specification – v1” document [10]. It is possible to add or drop certain Linux capabilities for a container by using “`--cap-add`” or “`--cap-drop`” options of the `docker-run` command:

```
$ docker run --cap-add=SYS_TIME --cap-dropp=CAP_NET_RAW ...
```

or it is possible to add all the capabilities and exclude a few ones:

```
$ docker run --cap-add=ALL --cap-drop=SYS_TIME ...
```

4.2.2 Linux Namespaces

A namespace wraps a global system resource in a way that it appear to the processes within the namespace that they have their own isolated instance of the global resource. Changes to the global resource within the namespace are visible only to the processes that are members of the namespace, but are invisible to other processes. Linux makes available *PID*, *network*, *SysV IPC*, *mount*, *user* and *UTS* namespaces. Namespaces can be used to implement containers. [19]

Docker automatically creates a set of namespaces for the container (using the `clone(2)` system call) when it is run. This provides a layer of isolation for each container, so it appears that it is running on a separate operating system (at least regarding the global resources isolation provided by the Linux namespaces). Processes running within a container cannot see or affect processes running in another container, or on the host operating system. Namespaces that Docker uses on Linux are [19, `namespaces(7)`]:

- **PID namespace:** Used for process isolation.
- **Network namespace:** Used for isolating and managing network interfaces, stacks, ports, etc.
- **IPC namespace:** Used for isolating and managing access to IPC resources (System V IPC, POSIX message queues).
- **Mount namespace:** Used for isolating and managing mount points.
- **User namespace:** Used for isolating user and group IDs, the root directory, or Linux capabilities.
- **UTS namespace:** Used for isolating kernel and version identifiers including hostname and NIS domain name.

Example of the network namespace isolation using the tool `unshare` (provided by the RPM package `util-linux`) for working with the namespaces:

```
$ ip a | grep -o "[0-9]\:[[:space:]][a-z].*\:"
1: lo:
2: enp0s25:
3: wlp3s0:
```

```
4: docker0:
5: virbr0:
$ # following command runs /bin/bash in the new network namespace
$ sudo unshare --net /bin/bash
$ ip a | grep -o "[0-9]\:[[:space:]][a-z].*\:"
1: lo:
```

As seen from the example, in the separated network namespace only localhost interface appears to be available. All the other interfaces are hidden and unavailable, because processes in the new network namespace are isolated from the network stack provided by the main namespace.

4.2.3 Control Groups

The control groups or Cgroups in short are a Linux kernel feature that allows to allocate and isolate the resources usage, such as CPU time, system memory, disk I/O, network bandwidth, or combinations of these resources among hierarchically grouped and labeled processes on a system [23]. Control groups can be used in multiple ways:

- by using tools like `cgcreate`, `cgexec` and others from `libcgroup-tools` RPM package,
- by using systemd (with systemctl commands, or by modifying systemd unit files),
- or through other software such as Linux Containers (LXC), or Docker

A Cgroup subsystem (also called a resource controller) represents a single resource, such as CPU time. The Linux kernel provides a range of Cgroup subsystems, that are mounted automatically by systemd. The list of currently mounted Cgroup subsystems on the host operating system can be found in `/proc/cgroups` and includes [23]:

- **blkio:** Sets limits on input/output access to and from block devices.
- **cpu:** Uses the CPU scheduler to provide Cgroup tasks an access to the CPU.
- **cpuacct:** Creates automatic reports on CPU resources used by tasks in a Cgroup.
- **cpuset:** Assigns individual CPUs (on a multicore system) and memory nodes to tasks in a Cgroup.
- **devices:** Allows or denies access to devices for tasks in a Cgroup.
- **freezer:** Suspends or resumes tasks in a Cgroup.
- **memory:** Sets limits on memory use by tasks in a Cgroup, and generates automatic reports on memory resources used by those tasks.
- **net_cls:** Tags network packets with a class identifier (classid) that allows the Linux traffic controller (the tc command) to identify packets originating from a particular Cgroup task.
- **perf_event:** Enables monitoring Cgroups with the perf tool.

- **hugetlb:** Allows to use virtual memory pages of large sizes, and to enforce resource limits on these pages.

Docker makes use of the control groups to share available hardware resources to containers and, if required, set up limits and constraints. It is important to note, that Cgroups cannot be used inside a container by default, and are set for the whole container with the Docker client CLI. Example of using the control groups with the `docker run` command:

```
$ docker run -d --name=c1 --cpuset-cpus=0 --cpu-shares=20 \
   fedora md5sum /dev/urandom
$ docker run -d --name=c2 --cpuset-cpus=0 --cpu-shares=80 \
   fedora md5sum /dev/urandom
$ top -p $(pgrep -d "," md5sum)
```

These commands will run Docker containers `c1` and `c2` which will both compute MD5 message digest from random bytes returned from `/dev/urandom` device until they are stopped. Both of these containers are locked down to the first CPU core (`--cpuset-cpus=0`), but 20% of the CPU will be allocated to the `c1` container, and 80% of the CPU will be allocated to the `c2` container (`--cpu-shares`), thus giving the container `c2` higher priority on the first CPU core than to the container `c1`.

4.2.4 Security-Enhanced Linux

Security-Enhanced Linux (SELinux) is a set of kernel modifications and user space tools for implementation of a mandatory access control mechanism in the Linux kernel. It provides a mechanism for supporting access control security policies, including mandatory access controls (MAC). SELinux can enforce rules on files and processes in a Linux system, and also on their actions, everything based on defined policies. [4]

In SELinux context, files (including directories and devices) are referred to as objects, while processes run by a user are referred to as subjects. Linux uses a *Discretionary Access Control (DAC)* system by default which allows users to control the permissions of files (objects) that they own. DAC controls how subjects interact with objects, and how subjects interact with each other. DAC access is based only on user identity and ownership. Things like the role of the user, the function and trustworthiness of the program, or the sensitivity and integrity of the data are ignored by DAC. On the other hand, SELinux can enforce an administratively set security policy over all processes and files in the system. All processes and files are labeled with a type which defines a domain for processes (domain separates processes from each other), and a type for files. The definition how processes interact with files, as well as how processes interact with each other is called the SELinux policy, and it is used to make access control decisions. SELinux policy rules are always checked after DAC rules and they are not used if DAC rules deny access first. For an operating system running SELinux, standard users are mapped to SELinux users, as SELinux users are part of SELinux policy and are separated from an operating system users. [4]

Processes and files are labeled with a SELinux context (`user:role:type[:level]`) that contains additional information [4]:

- **SELinux user:** The SELinux user identity can be associated to one or more roles that the SELinux user is allowed to use.

- **role:** The SELinux role can be associated to one or more types the SELinux user is allowed to access.

- **type:** When a type is associated with a process, it defines what processes (or domains) the SELinux user (the subject) can access. When a type is associated with an object, it defines what access permissions the SELinux user has to that object.

- **level:** Is an optional attribute, written as lowlevel-highlevel if the levels differ, or lowlevel if the levels are identical (`s0-s0` is the same as `s0`). Each level is a pair of sensitivity-category, with categories being optional. If there are categories, the level is written as "sensitivity:category-set". If there are no categories, it is written as "sensitivity".

When running SELinux, all of this information is used to make access control decisions. Docker interacts with SELinux policy to achieve protection of the host, and protection of containers from one another. It runs each container in a unique SELinux context. This context is permitted to access only the files and mount points required for that specific container.

```
$ docker run --rm -it fedora /bin/bash
# ls -dZ /etc
system_u:object_r:svirt_sandbox_file_t:s0:c546,c986 /etc
# ps -eZ
LABEL                                            PID TTY TIME     CMD
system_u:system_r:svirt_lxc_net_t:s0:c546,c986 1   ?   00:00:00 bash
system_u:system_r:svirt_lxc_net_t:s0:c546,c986 32  ?   00:00:00 ps
```

The above example shows SELinux context of files inside a Docker container. Docker offers two forms of SELinux protection: *type enforcement* and *multi-category security (MCS) separation*.

Type enforcement is a kind of enforcement in which rules are based on process "type" SELinux attribute. The default type for a confined container process is `svirt_lxc_net_t` and is permitted to read and execute all files types under `/usr` and most types under `/etc`, but it is not permitted to read content under other standard directories (like `/var`, `/home`, `/root`, `/mnt`, etc.). Type `svirt_lxc_net_t` is permitted to use the network and write only to files labeled `svirt_sandbox_file_t` and `docker_var_lib_t`. All files in a container are labeled by default as `svirt_sandbox_file_t`. Access to `docker_var_lib_t` is permitted in order to allow the use of docker volumes. [27, 18]

Multi-category security (MCS) separation works by assigning a unique value to the "level" attribute of the SELinux label of each container (also every file that a container writes carries that unique value). By default each container is assigned the MCS level equivalent to the PID of the docker process that starts the container. The MCS level attribute usually looks something like `s0:c1,c2`. Such a label would have access to files labeled `s0`, `s0:c1`, `s0:c2`, `s0:c1,c2`, but would not have access to `s0:c1,c3`. All MCS labels are required to use two categories in the MCS level attribute. If a file is labeled only `svirt_sandbox_file_t`, then by default all containers can read it. Therefore, MCS is used to add isolation between containers by using unique categories in the level attribute of the SELinux label for each container. No two containers can have the same MCS label by default. [27, 18]

One problem caused by running each container with its own SELinux context is sharing filesystems between the host operating system and the container. If SELinux policy is enforced on the host OS, a directory mounted from the host to the container is not writable for the container. The workaround for this problem is assigning the proper SELinux policy type to the host directory:

```
$ chcon -Rt svirt_sandbox_file_t host_dir
```

where `host_dir` is a path to the directory on the host system that is mounted to the container.

Docker added support for SELinux context ("`z`" and "`Z`" as mount modes) for the data volume mounts (see the Subsection 4.1.9), where:

```
$ docker run -v /var/db:/var/db:z fedora /bin/bash
```

will automatically do the "`chcon -Rt svirt_sandbox_file_t /var/db`". More detailed description can be found in `docker-run` manual page under the "`-v`" command line option.

It is important to note that SELinux is not namespaced which means containers cannot have their own separate SELinux policies by default. SELinux will always appear to be disabled in a container, although it is running on the host operating system.

4.2.5 Secure Computing Mode

Secure computing mode (Seccomp) filtering helps with creation of sandboxes for running processes. The initial implementation allows a process to make a one way transition into a secure state where it cannot make any system calls except `exit()`, `sigreturn()`, `read()` and `write()` to already open file descriptors. If it attempt any other system calls, the kernel will terminate the process with *SIGKILL*. [19]

A more recent extension to Seccomp allows to specify a filter for incoming system calls of a process. The filter is expressed as a Berkeley Packet Filter (BPF) program, where the data operated on is system call number and the system call arguments. This allows to define what system calls should be blocked, as well as filter on system call arguments (numeric values only; pointer arguments cannot be dereferenced). Having a reduced set of available system calls reduces the total kernel surface exposed to the application. The filter has ability to raise a signal when process tries to call a disallowed system call, allowing the signal handler to simulate it (which can be useful for debugging purposes). [17]

Docker containers use Seccomp filtering to disable certain system calls which are not required to be available inside containers. These are system calls like loading a new kernel into the memory (`kexec_load()`) or system calls for working with the kernel modules which is not allowed inside containers (complete list of disabled system calls for Docker containers can be found in "Seccomp security profiles for Docker" [11]). To find out if the kernel is configured with `CONFIG_SECCOMP` enabled:

```
$ cat /boot/config-$(uname -r) | grep CONFIG_SECCOMP=
CONFIG_SECCOMP=y
```

One can find out Seccomp status for the process by looking at the Seccomp flag in the process status file:

```
$ grep Seccomp /proc/<pid>/status
```

where possible values are:

```
0: Seccomp is not enabled
1: Seccomp ‘‘strict mode’’ is enabled (initial implementation)
2: Seccomp-bpf is enabled (filtering with a Berkeley Packet Filter)
```

It is also possible to run a container without the default Seccomp profile:

```
$ docker run --rm -it --security-opt seccomp=unconfined ...
```

or with the custom profile specified in the external file in *JSON* format (for more details see [11]).

4.2.6 Union File System

Union file system is a filesystem which implements a union mount for other file systems by merging several directories into a single unified view. Unioning allows to keep files with the same names separate physically, but to merge them logically into a single unified view. A union is a collection of merged directories, and physical directories are called branches. Branches can be transparently overlaid, forming a single coherent file system. The priority of mounted branches is specified when they are mounted. In situation, when both branches contain a file with the same name, only a file from the one with the higher priority will be visible. [6]

Each Docker image on the system is stored as a layer. Docker uses union file systems to implement layers in Docker images and can make use of several variants listed in the Table 4.1.

Technology	**Storage driver name**
OverlayFS	overlay
AUFS	aufs
btrfs	btrfs
Device Mapper	devicemapper
VFS	vfs
ZFS	zfs

Table 4.1: Union file systems supported by Docker and their driver names.

Each layer of Docker image fully describes how to recreate an action which led to its creation. This way only layer updates need to be propagated which makes propagation of images very lightweight. When Docker boots a container from an image it first mounts the root file system as read only. After that, every time a change to the file system happens, it attaches another file system layer to that container using union mounts to reflect this change.

```
$ docker info | grep -i "storage driver"
Storage Driver: devicemapper
```

Fedora is using the Device Mapper kernel framework that provides technologies for volume management. The devicemapper kernel driver (which uses the Device Mapper kernel framework) stores every image and container on its own copy-on-write virtual device (also called snapshot). Each image layer is a snapshot of the layer below it. The lowest layer of each image is a snapshot of the base device with a filesystem:

```
$ docker info | grep -i "backing filesystem"
 Backing Filesystem: ext4
```

Device Mapper technology works at the block level rather than the file level. [29]

4.3 Interfaces to the Virtualization on Operating System Level

Docker can make use of two interfaces that work with Linux kernel containment features and technologies: LXC and *libcontainer*.

4.3.1 Linux Containers

Linux containers (LXC) is a userspace interface for the Linux kernel containment features. It provides an API (templates, library and language bindings) and simple tools to access the Linux kernel containment features to create and manage software containers. [16]

Linux containers allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) and also isolation of system resources. To do so, LXC uses the following kernel features to contain processes [16]:

- Kernel namespaces (ipc, uts, mount, pid, network and user namespace),
- SELinux profiles,
- Seccomp policies,
- Chroots (using `pivot_root(8)`),
- Linux kernel capabilities,
- and Cgroups (control groups).

The goal of LXC is to create an environment as close as possible to a standard Linux installation but without the need for a separate kernel.

4.3.2 Libcontainer

Libcontainer is a library written in pure Go language which was developed to provide an interface for accessing the kernel's container APIs directly, without any other dependencies. The library makes it possible for Docker to manipulate Linux namespaces, filesystem access controls, network interfaces, firewalling rules, control groups, Linux capabilities, and SELinux profiles, all without depending on LXC or any other userland package. Libcontainer is the default library used in Docker, but LXC is still optional and can be switched to.

Docker uses libcontainer when creating a new container for configuring the Linux kernel containment features in the following way [10]:

1. Creates all namespaces for the container via the `clone(2)` system call.
2. Creates a root filesystem (rootfs) to jail and spawn processes inside the container; any binaries to be executed must be contained within this rootfs. A `pivot_root(8)` is used to change the rootfs for the process, effectively jailing the process inside the rootfs.
3. Mounts certain filesystems (like `/proc`, `/sys`, `/dev/pts`, etc.) that are required to be mounted within the rootfs for a container to execute properly.
4. Populates `/dev` with a set of required device nodes after the container's filesystems are mounted within the newly created mount namespace (like `/dev/null`, `/dev/zero`, etc.).
5. All Cgroups subsystems are used to handle resource allocation for the container (processes running within the container are placed inside the correct Cgroups).
6. Drops certain Linux capabilities.
7. Disables certain Linux system calls (see [17]).
8. Sets SELinux policy for all the container files.
9. Runs a new process inside of the container.

More specific details can be found in the "Container Specification – v1" [10].

4.4 Docker Containers Limitations

By default, Docker containers are running in "unprivileged" mode, which means that they:

- are running with all Linux namespaces unshared from the host,
- have no access to the host filesystem,
- have no access to filesystems of other containers on the host thanks to SELinux MCS separation,
- are not allowed to access or create any devices on the host (except the required set [10]),
- have certain Linux kernel system calls disabled by default, for example, it is not possible to add or remove modules from the Linux kernel (for more details see [11]),
- have certain Linux capabilities disabled by default [10],
- are not allowed to do any SELinux operations, which means that containers cannot have their own separate SELinux policies by default (see the Subsection 4.2.4 for more details).

Another limitation is based on the fact, that Docker containers are using the operating system level virtualization (Subsection 3.1.3), thus all containers running on the same host operating system shares the same kernel and use it directly. Therefore, it is important to understand that breaking kernel in a container means breaking it also for the host operating system and for all the containers running on that host.

4.5 Super Privileged Containers (SPC)

As stated in the previous section (Section 4.4), all Docker containers are running as "unprivileged" by default. But sometimes, it is required from containers to manage the host operating system or other containers. These are referred to as "super privileged containers". Super privileged containers (SPC) have nearly all the same access to the host as processes running outside of a container on the host, and they are also running with the root privileges. To make use of SPC, following adjustments need to be done when running a new container [28]:

1. Run a container with "`--privileged`" option of the `docker-run` command:
   - disables Seccomp/user namespace,
   - disables SELinux separation,
   - disables mounting of filesystems readonly,
   - allows creation of Linux devices.
2. Disable namespace separation:
   - `docker run --net=host`
     - uses the host's network devices directly within a container,
   - `docker run --ipc=host`
     - shares the host's IPC namespace within a container,
   - `docker run --pid=host`
     - a container can see all the host processes.
3. Extend default mount namespace:
   - `docker run -v /run:/run`
     - bind mounts the host's `/run` in a container,
     - makes it possible to manipulate the host's systemd or Docker daemon from within a container,
   - `docker run -v /var/log:/var/log`
     - allows to read and write log files from the host's `/var/log` directory from within a container,
   - `docker run -v /etc/localtime:/etc/localtime`
     - causes the host system's timezone to be used within a container,
   - `docker run -v /dev:/dev`
     - shares the host's device nodes with a container,
   - `docker run -v /:/host -e HOST=/host`
     - gives full access to the host including the ability to "`chroot /host`" and access the whole host from within a container.

The resulting Docker command is as follows:

```
$ docker run --rm -it --name spc_fedora --privileged \
        --ipc=host --net=host --pid=host \
        -v /run:/run -v /var/log:/var/log \
        -v /etc/localtime:/etc/localtime -v /dev:/dev \
        -v /:/host -e HOST=/host fedora /bin/bash
# /host/usr/bin/docker ps -a --format \
      "table {{.Image}}\t{{.Command}}\t{{.Status}}"
IMAGE                COMMAND              STATUS
fedora               "/bin/bash"          Up 5 minutes
```

The super privileged container, for example, can see itself by executing the `docker-ps` command from the host's filesystem, as can be seen from the above commands listing.

Also, Linux capabilities (see the Subsection 4.2.1) or custom Seccomp profiles (see the Subsection 4.2.5) can provide more fine grained controls over containers or SPC.

4.6 Related Technologies

Docker is not the only project providing solutions for working with the software containers. Other projects include *systemd-nspawn* from systemd developers [19], *Rocket (rkt)* from CoreOS developers [7], *LXD* from Canonical [16], or *Drawbridge* from Microsoft [3].

Systemd-nspawn may be used to run a command or OS in a lightweight namespace container. It is more powerful than *chroot* since it fully virtualizes the file system hierarchy, as well as the process tree, the various IPC subsystems and the host and domain name. Other limits from within the container [19, `systemd-nspawn(1)`]:

- it limits access to various kernel interfaces in the container to read-only (such as `/sys`, `/proc/sys` or `/sys/fs/selinux`);
- network interfaces and the system clock may not be changed;
- device nodes may not be created;
- the host system cannot be rebooted;
- kernel modules may not be loaded.

Rocket is a command line interface for running software containers on Linux. The key features and goals of Rocket include integration with init systems (systemd, upstart) and cluster orchestration tools, compatibility with other container software (for example ability to run Docker images), and modular and extensible architecture. [7]

LXD is a tool for container management. LXD uses LXC to create and manage the containers. It is made of three main components:

- A system-wide daemon (LXD)
- A command line client (LXC)

- An OpenStack Nova plugin (nova-compute-lxd)

The command line tool is designed to handle connections to multiple container hosts and to give an overview of all the containers on the network. It is possible to create new containers on the local host system or on the hosts controllable over the network, and even move them around while they are running. The OpenStack plugin makes it possible to run workloads on containers, allowing to use LXD hosts as compute nodes. [16]

Drawbridge is a form of virtualization for application sandboxing. It combines two core technologies [3]:

- a picoprocess: a process-based isolation container with a minimal kernel API surface, and with all traditional OS services removed;
- a library OS: a version of Windows enlightened to run efficiently within a picoprocess.

4.7 Checking Docker Containers

There are many approaches of testing applications using Docker infrastructure. In general, either Docker images or Docker containers can be tested.

Docker images package applications and before actually running them, there are multiple things that can be validated on a Docker image. These may, for example, include:

- test for Dockerfile validity;
- test for labels (metadata in form of a <key> / <value> pairs stored as strings);
- test if image can be built;
- test if number of image layers agrees.

These kind of tests work with static data or an application configuration, usually stored inside a Docker image, but do not require the packaged application inside a Docker image to be run. Therefore, tests for Docker images can be considered a static software testing method.

On the other hand, Docker containers contain and run applications. After creating a read-write layer on top of an image, the packaged application is executed, forming a container. Possible tests for Docker containers may include:

- test that connection with the containerized application can be established;
- test that required port is open;
- test that for the running container, all required environment variables are set;
- test if the application is properly configured inside the container;
- test if the application functions properly based on its outputs or logs;

and many other test cases. Testing Docker containers can be considered a dynamic software testing method, as it involves interaction with the application running inside a Docker container. The focus of this work are the dynamic testing methods of testing applications running in Docker containers and they are discussed in the Chapter 6.

Chapter 5

Use Cases for Running Software Systems in Docker Containers

Moving a software system with all its dependencies and libraries to Docker infrastructure is straightforward. To build an image, Dockerfiles are used. A Dockerfile contains all the commands to assemble an image, layer after layer. Image build process reads a Dockerfile and commits results of each Dockerfile command as a separate layer. Prepared image is then run, forming the container with the running application.

Many examples of Dockerfiles can be found on the git repository for the official Docker images [12], or in the official repository provided by the Docker Hub Registry, both of these repositories are publicly available. The Docker Hub Registry provides already built and prepared official Docker images, which are intended to provide base images for running and developing applications, minimizing repetitive work needed to be done when setting up the environment for an application, or installing and configuring dependencies and supporting software [31]. For example, the official Docker Hub Registry includes packaging for *Java*, *PHP*, *Python*, *MariaDB*, *Apache* HTTP server, and many others including GNU/Linux distributions like Fedora, *Red Hat Enterprise Linux*, or *Debian*. In general, Docker images can be divided in two categories based on the content packaged inside them: *language images* or *application images*. Both categories provide base images. Base image is an image which has no parent and is typically used as a base for an application in a way that the application and all the application files are placed on top of the base image. GNU/Linux distributions are typically used as base images as they provide software collections, which are based upon the Linux kernel, including a package management system.

Language images, also called language stacks, are pre-built stacks with programming languages. They are intended to provide environments for the specific programming language including all required dependencies. Each language stack's official repository has multiple versions of the language with a tag for each version, which makes it easy to pick the specific version needed for a software project.

Application images are pre-built stacks, usually with pre-configured application ready to run. For instance, Apache HTTP server, which can be used as the base image for building some web based applications or hosting web pages. A content with a configuration is committed on top of the base image and after the execution of such an image, HTTP server will handle the provisioning and accessing the content.

Based on analysis of multiple Docker images, including ones in the official repository [12], three main use cases for running software systems in Docker containers were categorized within this work. The categorization is based on distribution of application parts into containers and communication between the distributed parts:

- *Single-container application on single-host system (SCSH)*,
- *Multi-container application on single-host system (MCSH)*,
- *Multi-container application on multi-host system (MCMH).*

5.1 Single-container Application on Single-host System

The SCSH use case is the most straightforward. An application or all application parts are contained only within one container. Container with the application runs on a single host operating system and it does not communicate or interact in any way with other running containers on host. Therefore, only building and running it, eventually exporting some ports, when an application requires them, is needed.

5.1.1 Use Case Example

The example of such a use case might be the following Dockerfile using Python language stack, putting Python application on top of it:

Listing 5.1: **SCSH Dockerfile**

```
FROM python:3-onbuild

RUN mkdir -p /usr/src/app
COPY python-app /usr/src/app
WORKDIR /usr/src/app

COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r requirements.txt

EXPOSE 80

CMD [ "python", "./app.py" ]
```

This Dockerfile will build on top of the python language stack, so commands like “`pip`” are available for usage. Also the HTTP port is exposed and it is possible to connect to the python application running in the container.

5.2 Multi-container Application on Single-host System

This type of use case covers an application or a software system distributed into multiple containers which are running on the same host operating system. It is usually required that distributed parts running in the separate containers need to communicate with each other or access exported interfaces from other containers. For this purpose, Docker uses networking. It also provides the linking system which allows to link multiple containers on a single host together and send connection information from one to another, to be able to exchange data more easily (see the Subsection 4.1.8 for more details).

5.2.1 Docker Compose

For connecting multiple containers it is also possible to use Docker Compose. It is a tool for defining and running multi-container Docker applications on single host systems. Using Compose follows this process [9]:

1. Define an application's environment in a Dockerfile.
2. Define the services that make up the application in "docker-compose.yml" configuration file so they can be run together in an isolated environment.
3. Command Compose to build and start the entire application.

Compose allows to manage the whole application environment, it can build, start, stop, rebuild, or view the status of the running application.

5.2.2 Use Case Example

To demonstrate the MCSH use case, consider the following example:

- **Container A:** Apache HTTP server (httpd) with a simple web page which will use a perl script to print fortunes (`fortune(6)`) for a user.
- **Container B:** Data container for storing `/var/www` directory.

First, create the following files on the host system and place them in the same directory:

Listing 5.2: **index.html** : HTML page with the one form action attribute, which will call the perl script fortune.pl to print a fortune on submit.

```
<html>
<head>
<title>Fortunes for today</title>
</head>

<body>
<form action="/cgi-bin/fortune.pl">
  <input type="submit" value="Give fortune">
</form>
</body>
</html>
```

Listing 5.3: **fortune.pl**: Prints a fortune formatted as HTML text.

```
#!/usr/bin/perl

my $my_say = qx(cowsay -f tux \$(fortune));

print "Content-type: text/html\n\n";
print "<a href=\"/index.html\">Back</a>\n";
print "<h2><xmp>$my_say</xmp></h2>\n";
```

Listing 5.4: **Dockerfile**: Uses a fedora image as the base image, installs required dependencies, adds files with content and tells Docker what command to run and that it exposes HTTP port.

```
FROM fedora

RUN dnf install -y httpd perl cowsay fortune-mod && dnf clean all

ADD index.html /var/www/html/index.html
ADD fortune.pl /var/www/cgi-bin/fortune.pl
RUN chmod 0755 /var/www/cgi-bin/fortune.pl

EXPOSE 80

CMD [ "/usr/sbin/httpd", "-D", "FOREGROUND" ]
```

Listing 5.5: **docker-compose.yml**: Configuration file for Docker Compose.

```
version: "2"
services:
  httpd_fortunes:
    build: .
    image: httpd_fortunes
    container_name: httpd_fortunes
    expose:
      - "80"
    ports:
      - "80:80"
    volumes_from:
      - httpd_fortunes-data

  httpd_fortunes-data:
    image: httpd_fortunes
    command: /bin/true
    container_name: httpd_fortunes-data
    volumes:
      - /var/www
```

Compose configuration file (docker-compose.yml) divides the web application between two containers: `httpd_fortunes` (Container A) and `httpd_fortunes-data` (Container B).

Both containers will be derived from the same image named `httpd_fortunes`, built from the Dockerfile in the current working directory. In this example, containers will not be connected through the link system, only through the mounted volume `/var/www`. This approach can be used for persisting the data in containers (see the Subsection 4.1.9 for more details). When a container is stopped, or removed, or an application inside crashes, all the data produced by the application during the runtime will be lost. That is why a data container is required, but this container must never exit, at least not before backing up the data which it incorporates. The `httpd_fortunes` container will run the httpd daemon as specified in the Dockerfile and will be responsible for serving the requests on the HTTP port 80. The `httpd_fortunes-data` container will not be accessible through the network, it will exit immediately after the start and will only serve as the data volume for the main `httpd_fortunes` container. To build the whole infrastructure run:

```
$ docker-compose build
```

in the directory containing index.html, fortune.pl, Dockerfile and docker-compose.yml files. When build process finishes, starting the whole infrastructure is done by:

```
$ docker-compose up -d && docker inspect \
      httpd_fortunes | grep IPAddress
```

which will start the web application and will also print the IP address of the running httpd. When the main container `httpd_fortunes` needs to be stopped or updated (for example because of the httpd security update), all the data remains untouched inside the data container `httpd_fortunes-data` and the new `httpd_fortunes` image can be started by executing the previous Docker Compose command again. To stop and remove the whole infrastructure run:

```
$ docker-compose down
```

5.3 Multi-container Application on Multi-host System

The previous two use cases were covering only containers running on the same host operating system. The MCMH use case includes situations where an application or a software system is not only distributed into multiple containers, but also these containers are distributed over multiple connected host operating systems. The problem with this type of use case is how to connect multiple containers to each other across different physical or virtual host systems, as Docker itself does not support this by default.

To solve this problem, cluster (connected computers that work together and can be viewed as a single system) orchestration and management tools for Docker can be used as, for example, *Kubernetes*, *Swarm*, *Fleet* or *Apache Mesos*.

Kubernetes is a system for managing containerized applications across multiple hosts in a cluster. Kubernetes provides mechanisms for application deployment, scheduling, updating and maintenance across the multi-host system. The main unit in Kubernetes is a pod, and all containers run inside pods. A pod can host a single container, or multiple cooperating containers. It can contain zero or more volumes, private to a container or shared across containers in a pod. When a pod is introduced into the system, it finds a machine that has sufficient available capacity to run it, and starts up the prepared container(s) there. [14]

Swarm is the native clustering tool for Docker. It uses the standard Docker API, meaning containers can be launched using normal Docker commands and Swarm will take care of selecting an appropriate host to run containers on. Each host runs a Swarm agent and one host runs a Swarm manager which is responsible for the orchestration and scheduling of containers on the hosts. [1]

Fleet builds on top of systemd which provides system and service initialization for a single machine. Fleet extends this to a cluster of machines. Fleet reads systemd unit files, which are then scheduled on a machine or machines in the cluster. Each host runs an engine and an agent. At any time, only one engine is active in the whole cluster, but all agents are constantly running. Systemd unit files (which normally run a container) are submitted to the engine, and engine schedules the job on the least loaded machine. [1]

Apache Mesos is designed to scale to very large clusters involving hundreds or thousands of hosts. It supports all kinds of workloads, not only Docker containers. Mesos can also run several frameworks for container orchestration including Kubernetes and Swarm. It consists of agent nodes which are responsible for running tasks. The master node is responsible for sending tasks to the agents and also maintains a list of available resources. [1]

Chapter 6

Testing Applications Running in Docker Containers

This chapter is dedicated to testing applications running in Docker containers, thus a dynamic testing method. Testing of Docker images is not discussed in this work. Docker allows to create a reproducible runtime environment for a tested application, including all its dependencies, libraries and static data. It is one of the biggest advantages of using Docker for testing software, as tests may be destructive for an application, or can leave it in a non-default state or even corrupted. This can be easily resolved by killing the flawed container and starting a new one with default configuration options. Starting a new container is very fast, as it utilizes only virtualization on the operating system level (Subsection 3.1.3). In contrast with with hardware emulation or with paravirtualization there is no need to start the whole horsepower behind virtual machine over again after each destructive test case.

To sum it up, using Docker to create testing environments and running tests inside containers have many benefits, for instance:

- quick setup of testing environment thanks to Docker images,
- tests reproducible across different machines, as all the configuration and dependencies are packed up with the software inside the container,
- no influence on the host system or other containers running on the host system thanks to operating system level virtualization,
- and ability to run tests in parallel.

On the other hand, it is important not to forget on Docker containers limitations and that operating system level virtualization might not always provide the required isolation for certain applications or testing environments. Therefore, it should be considered first, whether software is suitable for running inside Docker containers.

6.1 Aspects for Applications Running in Docker Containers

When dealing with testing applications running in Docker containers, there are different aspects of application under test which should be taken into account. These aspects are based mainly on the Docker containers limitations (Section 4.4) and the proposed use case categories (Chapter 5). Aspects should give a better overview about an application setting within Docker containers infrastructure, and they can also help to choose a suitable testing method for testing the application. All the proposed aspects are listed in the Table 6.1 and are basically yes/no questions.

1. Requires network and	**a.** acts like a server (provides a network service) **b.** acts like a client (requires a network service)
2. Runs in multiple containers and	**a.** containers run on a single host system **b.** containers run on multiple host systems
3. Shares filesystem data with	**a.** host system **b.** other host systems (through NFS, or sshfs) **c.** container(s) on the same host system **d.** container(s) on other host system(s)
4. Runs as other user than root (“`docker run -u UID ...`”)	
5. Uses SELinux to isolate from the host system and from other containers running on the host system (Subsection 4.2.4)	
6. Uses only allowed Linux kernel system calls (Section 4.4)	
7. Enables/Disables additional Linux kernel system calls (using Seccomp 4.2.5)	
8. Runs in a container(s) with limited system resources (using Cgroups 4.2.3)	
9. Requires to run in a super priviledged container (Section 4.5)	

Table 6.1: Aspects for an application running in a Docker container.

An example on how to use these aspects can be demonstrated on a web application running on top of the base image with Apache HTTP server, which mounts the `/var/www/` directory from the host system into the running container. The following aspects applies for this web application (numbering refers to the Table 6.1):

1a. requires network and provides a network service;

3a. shares filesystem data with the host system;

5. uses SELinux to isolate from the host system and from other containers running on the host system;

6. uses only allowed Linux kernel system calls.

The answer for the rest of the aspects (2, 4, 7, 8, 9) is no. It is clear from these aspects, that this web application:

- provides network service through some interface which can be tested;
- will run under the root privileges;
- can use only Linux kernel system calls enabled in the default Seccomp profile [11], and so operations like loading kernel modules are not allowed from within the container;

- have no limits on system resources (important when doing performance testing);
- have no access to the underlying host system other than to the mounted directory `/var/www/`.

6.2 Tests in Docker Containers

Even if testing applications in Docker containers have many advantages, there are also some issues which need to be addressed. This work addresses possible issues when running tests in Docker containers as part of the following testing tasks:

- creating a testing environment in a container or in an image,
- deploying tests into a container or into an image,
- running tests inside a container,
- monitoring and managing process of testing,
- and gathering results of testing and analyzing potential problems.

Another issue is with the Docker containers limitations. Based on these limitations, it should be considered first, if running the test in a container is appropriate. If certain test needs some more privileges than Docker containers give by default, then it should be evaluated, whether:

- to extend privileges of the container so the test can be executed inside it (maybe even extend privileges so the container becomes super privileged as discussed in the Section 4.5),
- or rather move the test into the fully virtualized environment provided by the hardware emulation or the paravirtualization where the test will not be limited.

The next subsections provide a basic overview of the issues associated with the testing inside Docker containers and suggest possible solutions.

6.2.1 Creating a Testing Environment in a Container or in an Image

To be able to run tests inside a container, it is first needed to prepare a testing environment. Firstly, it should include evaluation if tests require some more privileges than Docker containers give by default (for the Docker containers limitations see the Section 4.4). The evaluation should focus on an analysis of what additional privileges are required for the test to run in a Docker container, and how these privileges influence the isolation between this container and the underlying host system. If it is not desirable for the test to have extended privileges (in case that the test is destructive and extending some privileges may cause a flaw in the host system, or any other case), it is better that the test is moved into the fully virtualized environment. Extending the privileges of a container is discussed in the Section 4.5.

Secondly, it is required to install packages with dependencies needed by tests to run. For an image, this can be done in a Dockerfile by running the required commands, and so creating an additional layer with the test dependencies. For an already running container, tests dependencies can be either copied into the container using `docker-cp` command, or installed or downloaded by running a command in the container using `docker-exec`.

6.2.2 Deploying Tests into a Container or into an Image

By deploying tests it is meant to copy them into a test container and make them available for execution within that container. There are three possibilities how to get tests into a Docker container:

- ADD/COPY commands of Dockerfile,
- `docker-cp` command,
- and volume mounts.

The first possibility is adding all the tests as a separate layer of an image. This can be done by "ADD" or "COPY" commands in a Dockerfile and then building the test image from that Dockerfile. The created test image will run the same application as the original image except that it will have the additional layer which contains the test files.

The second possibility is for an application packed up and already running inside a container. In this case it is needed to copy the files required for testing into a running container using `docker-cp` command, for instance:

```
$ docker cp test.sh CONTAINER:/tmp/
```

will copy the `test.sh` script into the `/tmp` directory of a running container CONTAINER.

Another possibility for deployment of tests into a container can be using a volume mount (see the Subsection 4.1.9), which mounts a volume populated with all the tests into the running container.

6.2.3 Running Tests inside a Container

There are two possibilities how to run tests inside Docker containers:

- as part of the image build process,
- or using `docker-exec` command.

The first possibility makes use of the build process of a Docker image. This is done by modifying a Dockerfile where it is possible to specify commands to copy or to mount required files into the image during its build process. Tests can be then executed as part of the build process using the Dockerfile's "RUN" command. More on this approach is discussed in the Subsection 6.3.1.

Another possibility is running tests in a container using `docker-exec`. This approach can be used only when tests are already deployed inside a container:

```
$ docker exec CONTAINER /tmp/test.sh
```

and will run the `test.sh` script in the `/tmp` directory inside the running container CONTAINER. Results of testing might be printed to standard output/error or written into the log files inside a container.

6.2.4 Monitoring and Managing Process of Testing

Docker provides `docker-ps` command which lists all the containers including their statuses, but this is not sufficient as the complete solution for monitoring and managing the testing process.

To be able to monitor and manage the process of testing in more detail, some kind of manager software would be needed. The main questions here are where and how exactly to run it. Three possible options for running manager software were identified within this work:

- running manager software outside of a container;
- running manager software inside a container;
- or divide manager software in two parts, one part would need to run on the host system, the other one inside a running container.

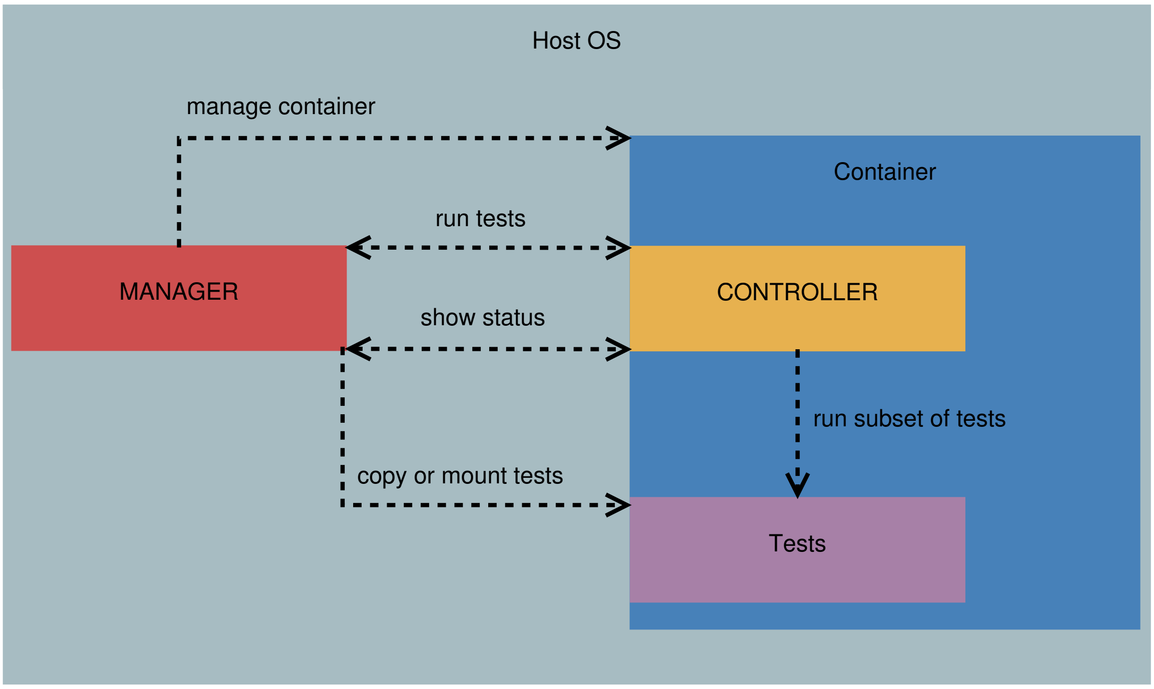

Figure 6.1: Manager software for managing tests in a container, divided in two parts: the part running on the host system (MANAGER), and the part running inside the container (CONTROLLER).

This work uses and discusses only the third option where manager software is divided in two parts. The part of manager running on the host system would be responsible for managing the whole process of testing: distributing individual sets of tests into containers, running containers including the parts of the manager software inside containers, preferably allowing parallel execution. The part of manager software running inside a container (referred to as controller in this work) would be responsible for running the distributed sets of tests, monitoring status and results of tests inside the running containers and provide information to a user (possibly in cooperation with the part of the manager software running on the host system). This variant of the manager software can be seen on the Figure 6.1.

6.2.5 Gathering Results from Testing and Analyzing Potential Problems

Applications running in a container can crash, and if they do, their data will still be on the container filesystem. In such a situation, it is sufficient to restart the failed container and get the required data. However, it might not always be trivial to debug unexpected problems in an application running inside a container. If an application crashes causing a container to exit in a way that it cannot be restarted, getting to the results of tests, the application logs, or other data stored inside the container can be strenuous. One possibility is to make a snapshot of the exited container using the `docker-commit` command, run the newly created image, and copy all the required files from its filesystem. In general, there are two possibilities for gathering files from a container:

- using volume mounts,
- or the `docker-cp` command.

Volume mounts solve the situations when a container crashes and cannot be restarted. Required files which should persist the container crash (including the files for testing and logs produced by the tests) could be placed in the separately mounted volumes on the host filesystem, or on another container which would be specifically intended for this purpose. When the container crashes or testing is finished, results from testing and various files for analysis of failed tests must be gathered from within the container filesystem. This is straightforward when volume mounts are used for deployment of tests or as a storage for other required files in the container, as these files are directly accessible from the host filesystem and Docker does not remove volume mounts even after the container exits. Therefore, using volume mounts is the safest method for gathering tests results from a container. Another benefit is that volume mounts practically solve three issues at once: deployment of tests into a container, gathering their results and also analyzing potential problems (more on volume mounts in the Subsection 4.1.9).

On the other hand, when volume mounts are not used and test results are a part of the container filesystem, it is required to copy them to the host filesystem. This can be achieved using `docker-cp` command:

```
$ docker cp CONTAINER:/tmp/results.log ./
```

which will copy the log file with the results (`/tmp/results.log`) from a running container CONTAINER into the current working directory on the host system. However, this method is not resistant against the corruption of the container filesystem.

6.3 Methods of Testing Applications in Docker Containers

The methods proposed in this section should give more specific answers to the issues discussed in the Section 6.2.

6.3.1 Method Using the Build Process

The build process of a Docker image consists of the sequence of commands specified in a Dockerfile. This method of testing has to modify a Dockerfile by providing the commands for deploying and running specified tests which should happen after the application setup phase. Only `docker-build` needs to be used with this method to build layers of an image, the image does not need to be run at all.

Listing 6.1: The example of a Dockerfile for the Apache HTTP server.

```
# setup phase
FROM fedora:latest
RUN dnf update -y && dnf clean all
RUN dnf install -y httpd && dnf clean all

ADD mysite.tar /tmp/
...     # other setup and configuration for httpd application

# testing phase
COPY test_dir /tmp/test_dir
RUN bash /tmp/test_dir/test_all.sh
RUN rm -rf /tmp/test_dir

EXPOSE 80
ENTRYPOINT [ "/usr/sbin/httpd" ]
CMD [ "-D", "FOREGROUND" ]
```

This method is best used for unit testing, or it can be used to test an application configuration before actually running it. Another example which can make use of this testing method might be testing a library before it is used by an application. Results of testing can be seen directly from standard output of the `docker-build` command. The advantage of this method is that it combines the image build process with testing.

6.3.2 Simple Runtime Method

Docker provides commands which are sufficient for deploying and executing tests inside a container, and monitoring their progress. But without further infrastructure built on top of that, including volume mounts, it is hard to gather and debug the testing results if some unexpected problems occur (as indicated in the Subsection 6.2.5). Still, this method can find usage because it is easy to implement. Method assumes that an application is packaged and running inside a Docker container. It is using `docker-cp` command to deploy the tests and to gather their results, and `docker-exec` command to run and monitor the tests:

```
$ docker cp test_dir CONTAINER:/tmp/test_dir
$ docker exec -t CONTAINER /tmp/test_dir/test_all.sh
$ docker exec -t CONTAINER ps -ejH
```

and when testing is finished, to obtain logs:

```
$ docker cp CONTAINER:/tmp/test_dir/results.log ./
```

where the CONTAINER is either a container name or a container ID of the running container which is the target of testing.

6.3.3 Method Using Manager Software and Volume Mounts

This method is also used in the framework's default plugin implementation (discussed in the Subsection 7.3.1). It uses the same architecture of the manager software as described in the Subsection 6.2.4, where one part of the manager software runs on the host system and the other one, called controller, is started inside the container.

The part of the manager software that runs on the host system creates a testing environment, which is created as a separate layer or layers of an image. All created additional layers are placed on top of the base image and the application layers which will be tested. This layers contain all the configuration and dependencies required by tests to execute properly.

Volume mount is used for deployment of tests into a container. It is a directory on the host system which is populated with all the required tests. This directory is then mounted into a container and will persist on the host filesystem even if the container crashes during testing, thus already collected testing results will not be lost in such cases.

Running tests inside a container is done right after starting the tested application and it is the responsibility of the manager software. The part of the manager software running on the host system is responsible for creating testing environments, starting and removing containers, mounting tests into containers and for the whole infrastructure management. On the other hand, the controller software (part of the manager software running inside the container) is responsible for running tests, logging their outputs into the log files placed in the volume mounted inside the container, and reporting the status and progress of testing.

There is no need to do any additional work to gather results of testing as these are placed in the directory on the host filesystem (thanks to volume mounts) and can be accessed directly. When the tested application crashes, its corresponding container will exit too, but most of the time it is possible to get to this container filesystem by executing `docker-start` command. For the situations when restarting the container is not possible, volume mounts are one of the remaining options for analysis of what happened. They make it easy to analyze various crashes of the tested application in the container, as logs from testing or any other required files from the container are preserved on the host filesystem.

Chapter 7

Framework Implementation

This chapter introduces the framework implemented within this work, all the implementation specific details, problems, solutions and the architecture of the implemented framework. By default, the framework is based on the method of testing in Docker containers which uses the manager software and volume mounts, presented in the Subsection 6.3.3. The manager software that runs on the host is part of the framework and it interacts and manages the other parts of the manager software running inside containers.

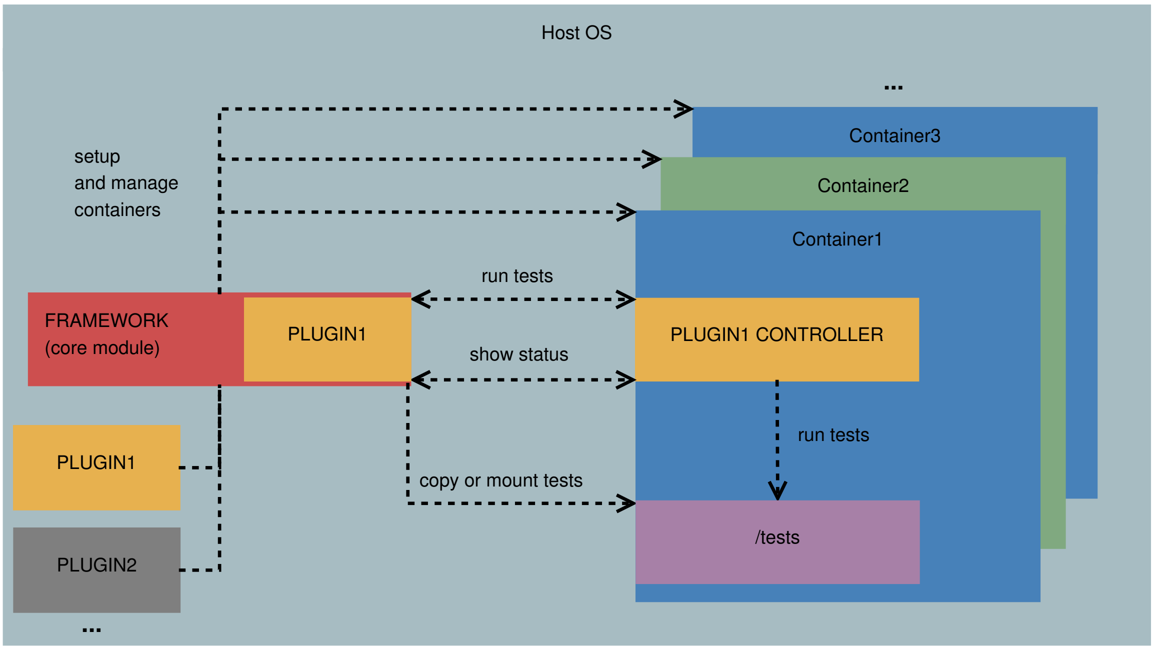

Figure 7.1: Basic framework architecture.

The framework consists of the core module which provides basic functionality and it is designed to be extensible with an additional custom code in form of a plugin. Plugins are used for implementation of custom testing methods and their different variants, for example the one described in the Subsection 6.3.3, or any other methods designed in the future. The basic architecture of the framework can be seen on the Figure 7.1.

The whole framework is implemented in the Bash command language and it is called *whale*. The choice of the Bash language for the implementation is because the framework needs to interact mainly with the Docker CLI, but also it needs to do text processing when parsing configuration files, or to traverse directory structures. Specifically, the following RPM packages are required by the framework: `bash`, `coreutils`, `grep`, `sed`, `gawk`, `nano`, `findutils`, `util-linux`, `procps-ng`, `tar`, `gzip`. Naturally, Docker must be also installed on the host system so the framework can interact with containers through the Docker CLI.

7.1 The Core Module

The core module of the framework is providing helper functions for the plugin code, including functions for creating and parsing configuration files or functions for exclusive access to the main framework configuration file. Exclusive access to this configuration file is to prevent inconsistencies due to possible parallel execution of the whale framework. The main framework configuration file stores the information about configured plugin which framework currently uses and all the test images and their testing recipes (more about the recipe configuration file in the Section 7.2). The configuration file is placed in the `.config/` directory of the current user at `$HOME/.config/whale/config`. The configuration file has the following structure:

Listing 7.1: **$HOME/.config/whale/config**

```
/absolute/path/to/plugin_directory
test_image_name   XMB    /absolute/path/to/recipe_file
base_image_name   YMB    -
...
```

The first line, as mentioned before, contains an absolute path to the directory with a plugin, which must contain at least one file of any name with the ".sh" extension providing the definitions of all the required functions (more details in the Section 7.3). The rest of the lines contain three values: a test image name, its size and an absolute path to the recipe configuration file, or a base image name with an application, its size and "-" character, as base images do not contain testing environments. Test images are then created from the base images.

7.2 Framework Configuration

Except the main framework configuration file, there are also two other types of configuration files used by the framework. These are:

- test configuration file (separate for each test),
- recipe configuration file.

7.2.1 Test Configuration File

Each test must be placed in a separate directory on the host filesystem. This directory must contain at least one executable file with the actual test code, but it also must contain the configuration file: test.ini, which the framework requires. The test configuration file test.ini has the following format:

Listing 7.2: **test.ini**

```
; comment line
[testinfo]
dependencies=bash coreutils
entrypoint=test.sh -v --option "argument"
timeout=20s
```

Each test.ini configuration file must contain the "*testinfo*" section with its own parameters. This section must provide at least the "*entrypoint*" parameter which specifies what file will be executed by the test manager as the entrypoint of the test directory inside a test container. Only one file can be specified, including arguments for that file. The files intended for execution (including files executed from the entrypoint file) should have set the execution file mode bits on the host filesystem. If the test directory contains only one file, it is set as executable and also it is made an entrypoint by default. The other two parameters are:

- *dependencies*: Specifies the list of packages with required dependencies for the test.
- *timeout*: Specifies the timeout for the test (same as `timeout(1)`), when specified time passes test is terminated. When no value is specified, it is set to the default value from the recipe configuration file (more details in the Subsection 7.2.3).

7.2.2 Tests Root Directory Structure

The framework deploys tests to the container from the specified root directory structure located on the host filesystem. It is possible to select all the tests from this directory, or only certain subset of tests. Each test inside the root directory structure must be placed in a separate directory. Test directories inside the root directory can form arbitrary nested structures.

Listing 7.3: Example of the root directory structure with tests.

```
tests_root_dir
|--- test1_dir
|     |--- test.ini
|     |--- file1
|--- test2_dir
|     |--- test.ini
|     |--- file1
|     |--- file2
|--- test3_dir
|     |--- test1_subdir
|     |     |--- test.ini
|     |     |--- file1
|     |     |--- file2
|     |--- test2_subdir
|     |     |--- test.ini
|     |     |--- file1
|     |--- ...
|-- ...
```

The framework can also utilize tests which use the *Beaker* library (*BeakerLib*). BeakerLib is a shell-level integration testing library, providing convenience functions which simplify writing, running and analysis of integration and blackbox tests [24]. Test using BeakerLib must provide Makefile which describes test case compilation, building the test package, but also contains metadata. The metadata includes information about owner of the test, test description, details for scheduling the test, test type, limits for architectures, test time, or test dependencies in form of software packages.

The whale framework uses the available metadata provided by the Beaker test Makefile and extracts the test dependencies and the test time from that metadata. This is done by the `make-conf` framework command, discussed in the Section 7.4.

7.2.3 Recipe Configuration File

Recipe is the main configuration file for setting up the testing environment and deploying tests into a container. It consists of the three sections:

- *global* section
- *environment* section
- *recipe* section

The recipe configuration file should have the following format:

Listing 7.4: **recipe.ini**: Example of the recipe configuration file for the root directory from the Listing 7.3.

```
; Configuration file for whale.
[global]
plugin=/absolute/path/to/plugin_directory
tests_root_dir=/absolute/path/to/tests_directory
docker_run_options=--privileged -v /run:/run ...
default_timeout=5m

pkg_install=dnf install -y
tmp_files_cleanup=dnf clean all
debug_pkgs=strace procps-ng

[environment]
wget -O /etc/yum.repos.d/custom.repo https://example.com/custom.repo
dnf install -y custom_repo_pkg
dnf clean all
...

[recipe]
test1_dir test2_dir
test3_dir/test1_subdir
test3_dir/test2_subdir
...
```

The “*global*” section consists of the following parameters:

- *plugin*: An absolute path to the plugin directory for the whale framework; plugin is responsible for setup, deploying, running, monitoring tests and obtaining testing results from a test container (more in the Section 7.3).
- *tests_root_dir*: An absolute path to the root directory with all the tests as described in the Subsection 7.2.2. Test name is then set as a relative path to the test directory starting from this root directory.
- *docker_run_options*: An additional options for `docker-run` command when launching a test container. It is possible to add various options, including volume mounts, options to create the super privileged container, or to limit resources for the test container. This parameter can be empty.
- *default_timeout*: Specifies the default timeout for each test (same as `timeout(1)`). Tests which do not specify timeout value inside their test.ini configuration file (as indicated in the Subsection 7.2.1) use this default timeout value. When specified time passes test is terminated.
- *pkg_install*: Distribution specific package manager with command for installation and option to answer yes for all the questions.
- *tmp_files_cleanup*: Distribution specific package manager with command for cleanup of temporary files for the currently enabled repositories (significantly reduces sizes of test images).
- *debug_pkgs*: List of packages which should be available inside a test container, mainly for debugging purposes. This parameter can be empty.

The “*environment*” section contains additional commands, which should be executed to extend the testing environment inside a test image or inside a test container, before the actual testing starts. These may include adding additional repositories, installing additional packages, setting environment variables, or any other actions. The format should be one command per line.

The “*recipe*” section contains “the recipe” for running tests inside a test container. Tests must be specified in form of a relative path starting from the “*tests_root_dir*” directory, the parameter of the “*global*” section in the recipe configuration file. Tests specified on the same line will be run together parallelly. The same test cannot be run parallelly multiple times – this means that if one test entry is specified multiple times on the same line, it will be run only once. The same test can be run multiple times only serially, thus it can be specified multiple times, but each entry must be specified on a separate line. In the example of the recipe.ini configuration file from the Listing 7.4, the tests “*test1_dir*” and “*test2_dir*” will run simultaneously in parallel, and after these tests will finish, the “*test3_dir/test1_subdir*” test will be run serially followed by the next serial test “*test3_dir/test2_subdir*”.

7.3 Framework Plugins

Plugins provide code, which can be plugged into the core framework module, extending its functionality. Plugins cover the features of testing methods and their different variants, for example the one described in the Subsection 6.3.3.

Plugins must be implemented in the Bash command language. Every plugin should provide the following Bash functions with the following functionality:

- `plugin_print_help()`: Prints plugin specific help.
- `plugin_setup()`: Prepares and pre-configures the testing environment with all dependencies for tests. Optionally also deploys tests into the prepared testing environment. This may include building a test image, or starting the containers required for testing.
- `plugin_run()`: If tests have not been deployed before, deploys them into a container, preferably as a volume mount. Starts a test container or containers and also starts the test manager, which is responsible for running the deployed tests.
- `plugin_status()`: Prints the progress and the status of testing. The function should document and return specific codes to indicate if testing is still in progress, or if it finished successfully or with an error.

The framework core module provides some useful functions for the plugin code, mainly for working with the configuration files, these may include [1]:

- `whale_getall_tests_for_dir()`: Prints all the tests inside the provided root directory with tests in form of relative paths starting from the root directory.
- `whale_recipe_export_global_conf()`: For the provided recipe configuration file: makes all the parameters of the "*global*" section available for the plugin code as global variables (the global variables will have the same names as their corresponding parameters, except that they are uppercase, i.e. the parameter "*default_timeout*" from the recipe configuration file will correspond to the "`$DEFAULT_TIMEOUT`" global variable, etc.).
- `whale_recipe_getall_environment_commands()`: For the provided recipe configuration file: prints all the commands of the "*environment*" section formatted as the "*cmd1; cmd2; ...*" string.
- `whale_recipe_getall_recipes()`: For the provided recipe configuration file: prints all the recipes of the "*recipe*" section including the new line characters.

The framework also provides the locking mechanism for exclusive access to its main configuration file `$HOME/.config/whale/config` (as described in the Section 7.1). Everytime a new image is built, record for that image needs to be created in this configuration file so the framework can track and manage its images and their corresponding containers. For this purpose, or for any exclusive actions needed to be done by the plugin code, functions `lock()` and `unlock()` should be used to prevent more instances of the framework from accessing the shared resource at once:

[1] the "`_whale`" prefix indicates that the function belongs to the core module

```
lock

; critical section - only one instance from all
; the whale processes can be here at the same time

unlock
```

Plugin can also redefine its own trap cleanup function for handling signals like *SIGHUP*, *SIGINT*, *SIGTERM*, or others. Otherwise, the core module trap cleanup function is called, which, by default, does the following actions:

- unlocks the exclusive lock before exiting, so the other instances of the whale framework can continue;
- removes the images and the containers left from the unsuccessful builds or from the interrupted builds;
- restores the tty flags;
- exits with the return code of the failed command.

Plugin trap cleanup function must include at least the functionality of the core module trap cleanup function.

To identify its containers, the framework uses labels (more details on labels in the Subsection 4.1.10). Every container started by the plugin code must be given two labels:

- `$WHALE_LABEL`: The general label used by the framework to recognize its containers.
- `$WHALE_IMG_LABEL=IMAGE_NAME`: The image label used to identify the image "*IMAGE_NAME*", which the container was started from.

It is then possible to find out, which containers belong to the whale framework, which image they were started from, and what recipe configuration file they are using, all based on these labels and based on the main configuration file `$HOME/.config/whale/config`.

7.3.1 Default Plugin

Default plugin provides the specific implementation of the method using the manager software and volume mounts (described in the Subsection 6.3.3). It is intended for testing the single-container application on single-host system use cases, as described in the Section 5.1. Firstly, the plugin prepares the testing environment by building a test image. Secondly, the built test image is started, forming a test container and tests are deployed into that container by the volume mount. Then, the manager software prepared inside the test container (also referred to as the controller) is executed, it runs the pre-configured tests and it can report the progress and status of testing when requested by the framework core module running on the host system. Tests are named as a relative path starting from their root directory (the "*tests_root_dir*" parameter in the recipe configuration file). Images created by this plugin are adding the "`w_default/`" prefix before the base image name, and for the image names to be unique, the "`/date_time`" postfix is appended after the base image name. Example of such a naming convention can be “`w_default/fedora/20160501_094115`” image created by the default plugin from the Fedora base image.

Function plugin_setup

In preparation of the testing environment, the default plugin builds a test image. It is based on the base image which contains the application or the software which will be the target of testing, adding the image layers with test dependencies and a configuration. The test controller software is also added into the test image and it consists of the following files:

- `/usr/bin/controller.sh`: The Bash script for starting sets of tests inside a test container.

- `/etc/controller.conf`: A configuration file for the controller script containing sets of tests. Each set of tests is specified on a separate line. The controller script starts set after set as specified in this configuration file from top to bottom. The next test set is started only if all the tests from the previous set finished. Tests within a set are run in parallel. Example of the configuration file for the controller script can be seen on the Listing 7.5 – basically, it has the same form as the “*recipe*” section of the recipe configuration file described in the Subsection 7.2.3.

Listing 7.5: **controller.conf:** A configuration file for the `controller.sh` script.

```
test1_dir test2_dir
test3_dir/test1_subdir
test3_dir/test2_subdir
```

The controller script uses the *Supervisor*: a client/server system that allows to control a number of processes on UNIX-like operating systems. Specifically, the following parts of the Supervisor system are used by the default plugin [32]:

- `supervisord`: The server piece of the Supervisor. It is responsible for starting child processes at its own invocation, responding to commands from clients, restarting its crashed or exited child subprocesses and logging their stdout and stderr outputs.

- `supervisorctl`: The command-line client piece of the Supervisor. It provides a shell-like interface to the features provided by the `supervisord`. From `supervisorctl`, it is possible to connect to different `supervisord` processes, get their status, stop and start subprocesses of the `supervisord`, and get lists of running processes of the `supervisord`.

- supervisord.conf: A configuration file used by the server process (`supervisord`).

Every built test image binds to the one specific recipe configuration file, which means that when the recipe file is changed, the old image should be removed and the new one should be built to reflect the changes. Each test from the “*recipe*” section of the recipe configuration file is added to the supervisord.conf file as a separate entry, including the controller script. Test has the same names in both, the supervisord.conf, and the controller.conf files. In the example from the Listing 7.6, the test `$test` (can be equal to, for instance, the “*test3_dir/test1_subdir*”) is the test name and also the relative path to the test starting from the root directory with all the tests on the host filesystem (the “*tests_root_dir*” parameter in the recipe configuration file). The `$timeout` and the `$entrypoint` are the specific values taken from the configuration files.

Listing 7.6: **supervisord.conf:** A configuration file for the `supervisord` daemon.

```
; Global settings for the supervisord.
[supervisord]
nodaemon=true
logfile=/tests/supervisord.log
loglevel=info

...

; The controller script is also started by the supervisord.
[program:CONTROLLER]
command=/usr/bin/controller.sh
directory=/tests
autostart=true
autorestart=false
startretries=0
startsecs=0
exitcodes=0
stderr_logfile=/tests/controller.log
stdout_logfile=/tests/controller.log

...

; Configuration for the test "$test".
[program:$test]
command=/usr/bin/timeout $timeout /tests/$test/$entrypoint
directory=/tests/$test
autostart=false
autorestart=false
startretries=0
startsecs=0
exitcodes=0
stderr_logfile=/tests/$test/$(basename $test).log
stdout_logfile=/tests/$test/$(basename $test).log

...
```

Lastly, the commands of the "*environment*" section of the recipe configuration file are executed, forming the last layer of the test image. The test image should now contain the tested application with the prepared testing environment.

Function plugin_run

The `plugin_setup()` function of the default plugin builds test images. The `plugin_run()` function takes a test image and runs it. This will run the application or the software intended for testing inside a new container. Selected tests specified in the "*recipe*" section of the recipe configuration file bound to the test image are copied from the root directory with tests (specified as the "*tests_root_dir*" in the recipe configuration file) into a directory selected by a user, preserving file modes and relative paths from that root directory. This newly created directory structure is then mounted in the new container into the `/tests` directory, and then:

1. the Supervisor is started,
2. the `supervisord` starts the `controller.sh` script,
3. the `controller.sh` script starts individual test sets.

The controller script is started as the child process of the `supervisord` and after it is started, it runs tests as provided in the `/etc/controller.conf` file using the `supervisorctl` client process. By default, each test will produce a log file made from its entrypoint executable file stdout and stderr outputs. Log from each test is placed into its corresponding directory located in the `/tests` directory mounted from the host filesystem. This way, even if a test container crashes or is cleaned up, test results will persist on the host filesystem. When test is run serially multiple times, log file from its entrypoint file will contain concatenated logs from each test run.

Function plugin_status

The default plugin status function prints the progress, and returns the status of testing. It returns these specific codes to indicate the status of testing:

- 2: testing is still in progress;
- 0: testing finished – all the tests exited with 0 code;
- 124: testing finished – one or more tests exited with non 0 code;
- 1: internal error.

7.3.2 Network Plugin

Network plugin also provides the specific implementation of the method using the manager software and volume mounts (discussed in the Subsection 6.3.3). It is intended for testing the multi-container application on single-host system use cases, as described in the Section 5.2. The `plugin_setup()` and the `plugin_status()` functions are similar to the default plugin implementation. The `plugin_run()` function of the network plugin differs from the default plugin function in the following:

- the test image built in the setup phase is run in a way that it provides the `docker` binary from the host filesystem to the test container, meaning that from the test container it is possible to access other containers running on the host system;
- an application container which will be the target of testing needs to be provided (it can be part of the multi-container application): the provided application container needs to be already running;

- the test container provides environment variables AC_ADDR and AC_PORT referring to the application container;

Testing is executed on the test container and tests can access the tested application container:

- using a network (on the address AC_ADDR and the port AC_PORT);
- or using the docker-exec or various other docker commands.

As tests and the tested application are not running in the same container when using the network plugin, the target of testing should be the applications which provide some services accessible from a network.

Similarly to the default plugin, images created by the network plugin are adding the prefix "w_network" before the base image name, and for the image names to be unique, the "/date_time" postfix is appended after the base image name. Example of such a naming convention can be "w_network/fedora/20160501_094115" image created by the network plugin from the Fedora base image.

7.4 Framework CLI

The framework command-line interface provides commands for creating configuration files, loading plugins, printing information about test images and containers, entering containers and analyzing their filesystems, moving images and containers from the local Docker registry into compressed archives and vice versa, or removing images and the running containers. The command-line interface of the whale framework (with the default plugin) provides the following commands:

- **make-conf**
  - Chooses plugin.
  - Creates the recipe configuration file.
  - Creates test.ini configuration files for tests.
  - Use "--beaker" option when making configuration for the tests using the Beaker library.
- **info**
  - Lists all built Docker images, their sizes and their recipe configuration files.
  - Lists all the started containers and the test images that containers are based on.
  - Prints available information about a test image/container.
- **setup** (plugin)
  - Either creates an image from a tested application Dockerfile or uses an existing image.
  - Adds the test controller, dependencies and configuration for all the tests based on the specified recipe configuration file as a new image layer.

- **run** (plugin)
  - Runs a test image (which includes the tested application and the test controller). Also applies command-line options and limitations for a test container from the recipe configuration file.
  - Copies tests (based on the recipe configuration file) into the chosen directory on the host filesystem and mounts them into a test container (logs will be available inside this directory on the host filesystem).
  - Runs the test controller inside a test container.
- **status** (plugin)
  - Shows progress and status of testing.
- **analyze**
  - Enters the specified container and gives a shell.
- **diff**
  - Shows changed files in a container compared to its image.
- **save**
  - Saves the specified test image from the local Docker repository and its recipe configuration file to the compressed archive.
- **load**
  - Loads an image from a compressed archive into the local Docker repository and also extracts its recipe configuration file from that archive into the specified location.
- **rm[-all]**
  - Cleans the selected test images and test containers.

Chapter 8

Framework Workflow and Testing

This chapter describes the workflow of using the framework CLI for testing applications running inside Docker containers. Simplified framework commands workflow can be seen on the Figure 8.1. Placement and configuration of tests is also discussed, and the workflow is demonstrated on one use case for each plugin.

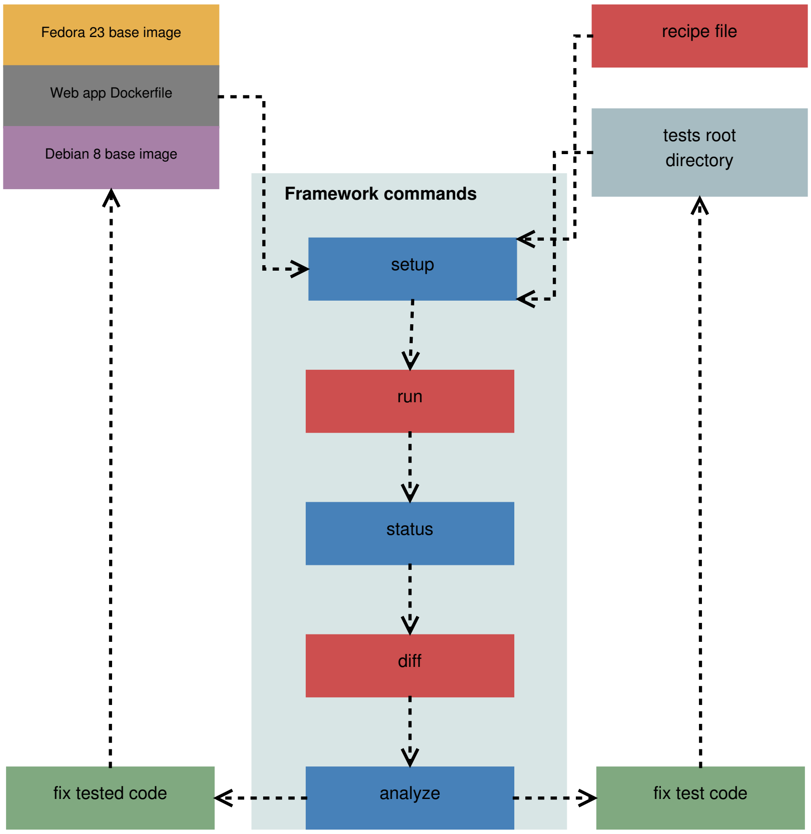

Figure 8.1: Framework commands workflow.

All the files, including code of the whale framework, Dockerfiles and code of tests can be found either on the included CD, or in the *GitHub* repository [1]. The files and paths

[1] `https://github.com/matusmarhefka/whale`

presented in the following text will be referencing these sources. When working with the included CD, copy the `whale/` directory from the CD to the host filesystem and change the current working directory of the host to the the `whale/` directory copied from the CD. When working with the Git repository:

```
$ git clone https://github.com/matusmarhefka/whale
$ cd whale/
$ ./whale help
```

Before using the framework, the following packages need to be installed:

```
$ dnf install -y bash coreutils grep sed gawk nano \
        findutils util-linux procps-ng tar gzip
```

8.1 The SCSH Use Case

Fuzz testing of the systemd D-Bus interfaces was chosen as a representative of the single-container application on single-host system use case. This use case is using the default plugin for testing, and it also demonstrates the adjustments needed to be done to the container to run an init system like systemd.

8.1.1 Setup of Testing Environment

Multiple tests for the default plugin are located at the `examples/tests/plugin-default` location, where the most important ones are:

- `examples/tests/plugin-default/Beaker/dfuzzer_test`—the Beaker test utilizing the tool for fuzz testing processes communicating through D-Bus, *dfuzzer* [2], which is used to fuzz test interfaces that systemd exports on D-Bus;

- `examples/tests/plugin-default/Dummy/dummy_test3`—the Bash test which sleeps for a specified amount of time to demonstrate the framework ability to timeout tests.

First of all, the framework needs to be configured, which includes setting a plugin that will be used for working with Docker containers, and generating test.ini configuration files for each test in the `examples/tests/plugin-default` directory. The `make-conf` command can be used for this purpose:

```
$ # Generates test.ini files for the tests using Beaker library:
$ ./whale make-conf --beaker examples/tests/plugin-default/Beaker \
        plugins/default
$ # Generates test.ini files for the rest of the tests:
$ ./whale make-conf examples/tests/plugin-default plugins/default
```

Beaker tests provide Makefiles which contain information about test timeouts, test dependencies and other metadata (see the Subsection 7.2.2). Therefore, the option "`--beaker`" fills the test.ini configuration files of these tests automatically with the metadata taken

[2] https://github.com/matusmarhefka/dfuzzer

from their Makefiles. All the tests in the `examples/tests/plugin-default` directory already have test.ini files created and filled, so this step can be skipped.

Secondly, the image which runs systemd needs to be built. For this purpose, the Dockerfile is prepared at `examples/Dockerfiles/fedora_systemd/Dockerfile`:

Listing 8.1: **examples/Dockerfiles/fedora_systemd/Dockerfile**

```
FROM fedora
VOLUME [ "/sys/fs/cgroup" ]
CMD [ "/usr/sbin/init" ]
```

The `setup` command provided by the framework is used to create a test image for running tests in a test container. It needs to be provided with:

- an existing image, or with a directory which contains a Dockerfile to build an image from;
- a path to a recipe configuration file: if it does not exist, it will be created and user will be allowed to edit it before the build process starts;
- and a path to a plugin directory.

The `setup` command will have the following form:

```
$ image=$(./whale setup examples/Dockerfiles/fedora_systemd \
        examples/Dockerfiles/fedora_systemd/recipe.ini \
        examples/tests/plugin-default)
```

This will create the `examples/Dockerfiles/fedora_systemd/recipe.ini` recipe configuration file and it will open that file in edit mode for a user to review it. To run systemd in a container, the following needs to be added to the "*docker_run_options*" parameter in the recipe configuration file (a parameter and its value must be specified on the same line in the recipe file – no new line characters are allowed):

```
docker_run_options=--cap-add=SYS_ADMIN --security-opt=seccomp:unconfined
-v /sys/fs/cgroup:/sys/fs/cgroup:ro -e container=docker
```

where:

- `--cap-add=SYS_ADMIN` adds the `CAP_SYS_ADMIN` Linux capability to the test container [19, `capabilities(7)`];
- `--security-opt=seccomp:unconfined` disables the default Seccomp profile, so the system calls like `mount(2)`/`umount(2)` and others required by systemd can be called from within a container;
- `-v /sys/fs/cgroup:/sys/fs/cgroup:ro` mounts the cgroup file system within a container (read-only), as systemd requires it to run properly;
- `-e container=docker` sets the environment variable `container` to let systemd know that it is running within a Docker container.

The last thing before saving the recipe configuration file and continuing with the image build process is preparing the testing environment for the Beaker tests inside that image, as some tests requires it. This is already prepared in the recipe configuration file template under the "*environment*" section and it just needs to be uncommented:

```
; comment line starts with the semicolon character
[environment]
wget -O /etc/yum.repos.d/beaker-client.repo https://beaker-project.org/...
dnf install -y beaker beakerlib
dnf clean all
```

Now, after saving the recipe configuration file, the image build process starts. When build process finishes, it prints the name of the created test image to standard output:

```
...
Test image created: w_default/fedora_systemd/20160501_005026
```

or as in this case to the variable `$image` which can be useful mainly in the automated scripts. This is because the `setup` command prints only the name of the built image to standard output and the rest of the output is redirected to standard error output.

8.1.2 Running and Monitoring Tests

When testing environment is prepared inside the test image, the test image can be run forming the test container:

```
$ cont=$(./whale run $image ./fedora_systemd_logs)
```

Tests specified in the `examples/Dockerfiles/fedora_systemd/recipe.ini` configuration file are copied into the `./fedora_systemd_logs` directory and this directory is then mounted at the `/tests` directory inside the test container. Subsequently, the controller software is executed in the test container, which will run all the specified tests. Logs from testing will be available in the `./fedora_systemd_logs` directory on the host filesystem and this directory will persist on the host filesystem even if the container will be removed. Progress and status of testing can be viewed with the `status` framework command:

```
$ ./whale status $cont
```

which prints the complete list of tests including the list of failed tests to standard output. The return codes of the `status` command are discussed in the Subsection 7.3.1. When testing finished, logs from fuzz testing systemd D-Bus interfaces can be found at:

```
$ less -r fedora_systemd_logs/Beaker/dfuzzer_test/dfuzzer_test.log
```

The framework also provides the `analyze` command for entering the specified container:

```
$ ./whale analyze $cont
```

or the `diff` command for showing changed files in a container compared to its image:

```
$ ./whale diff $cont
```

8.1.3 Other Commands

The `info` framework command can be used to print the information about currently configured plugin, to list all available test images and their containers, or to print information about the specified image or container. Test images and containers can be removed using the framework `rm` or `rm-all` commands.

The built test images and their recipe configuration files can be saved by the framework, which can be useful when moving them to another machine:

```
$ ./whale save $image fedora_systemd_test
$ ./whale rm-all running
$ ./whale rm $image
Untagged: w_default/fedora_systemd/20160501_005026:latest
...
$ ./whale load fedora_systemd_test Dockerfiles/fedora_systemd/recipe.ini
dc1c22b6b5a7: Loading layer 88.38 MB/88.38 MB
...
Dockerfiles/fedora_systemd/recipe.ini already exists, overwrite [y/N]: y
Please update the ‘plugin’ and the ‘tests_root_dir’
parameters of the ‘examples/Dockerfiles/fedora_systemd/recipe.ini’
recipe file if these parameters have changed.
Test image loaded: w_default/fedora_systemd/20160501_005026
```

The framework `load` command loads an image from the specified compressed archive into the local Docker repository and also extracts its recipe configuration file from the archive to the specified location. The `load` command also prints the warning that the "*plugin*" and the "*tests_root_dir*" parameters should be checked, as the loaded image was moved to different filesystem and these parameters require absolute paths.

8.2 The MCSH Use Case

The multi-container application on single-host system use case example was implemented within this work to demonstrate the software system distributed into multiple containers. The software system can be thought of as a backend to a bank system for working with customer accounts. It is running in three containers:

- `xmlrpc_db`: the database container for storing information about customers and their account (for simplicity, the database contains only one table and each customer can have only one account);
- `xmlrpc_db-data`: the data container for storing tables from the database container;
- `xmlrpc_server`: the service container with the XML-RPC interface implemented in Python, which provides the remote procedure call (RPC) interface for accessing and modifying the database container.

The RPC interface of the service container provides the following procedures (with self-explanatory names): `get_all_accounts`, `get_account`, `add_account`, `delete_account`, `get_credit`, `update_credit`. The whole multi-container application can be built and run using the prepared script:

```
$ examples/Dockerfiles/xmlrpc_server/cmd build
$ examples/Dockerfiles/xmlrpc_server/cmd run
```

The target of testing is the RPC interface of the service container. Tests are located in the `examples/tests/plugin-network` directory and they already contain test.ini configuration files. Before preparing the testing environment inside an image with the `setup` command, the framework needs to be configured to use the network plugin:

```
$ ./whale make-conf plugins/network
```

Then the testing environment can be created based on the Fedora image:

```
$ image=$(./whale setup fedora \
        examples/Dockerfiles/xmlrpc_server/recipe.ini \
        examples/tests/plugin-network)
```

There is no need to modify the created recipe configuration file. For the `run` command of the network plugin, an application container which will be tested through the network must be specified. In this case, the `xmlrpc_server` container is used:

```
$ cont=$(./whale run xmlrpc_server $image xmlrpc_server_logs)
```

Tests are run inside the prepared image `$image` and they can access the `xmlrpc_server` container either through the network, or by using the `docker` binary which also makes it possible to obtain logs or enter other containers running on the host system (for more details see the Subsection 7.3.2).
An example can be the `examples/tests/plugin-network/10-show_db` test, which accesses the database container `xmlrpc_db` directly using the `docker` command and not by using the XML-RPC interface provided by the `xmlrpc_server` container.

As with the default plugin, logs from testing can be viewed in the specified directory (`xmlrpc_server_logs`), and status of testing with the `status` framework command:

```
$ ./whale status $cont
```

To cleanup the multi-container application and all the test images and containers from the host system, including the framework configuration, run the following commands:

```
$ examples/Dockerfiles/xmlrpc_server/cmd rm all
$ docker rmi xmlrpc_server
$ ./whale rm-all all
$ rm -rf $HOME/.config/whale
```

Chapter 9

Conclusion

The goal of this work was to analyze Docker containers use cases for running different kinds of software systems, and to propose methods of testing these systems for their different setups. The work introduces reader to the virtualization method called software containers, or operating system virtualization, and discusses its advantages and disadvantages compared to the other types of virtualization. Reader is also familiarized with the Docker platform architecture built on top of the operating system virtualization, and with the underlying containment features and technologies provided by the Linux kernel which are used by the Docker platform.

Three main use cases for running software systems in Docker containers were categorized within this work. The categorization is based on distribution of application parts into containers and communication between the distributed parts.

Another contribution of this work are proposed aspects for the applications running in Docker containers. These aspects are based mainly on the Docker containers limitations and the proposed use case categories for running software systems in Docker containers. After evaluating these aspects it can be easier to decide, if it is reasonable to package an application into Docker infrastructure, or if certain tests are suitable for execution inside Docker containers.

The issues associated with running tests in Docker containers were also presented. These issues are addressed as part of the testing tasks, which include creating a testing environment in a container or in an image, deploying tests into a container or into an image, running tests inside a container, monitoring and managing process of testing, and gathering results of testing and analyzing potential problems. The proposed methods then address the specified testing tasks and they give more detailed answers to the presented issues of running tests in Docker containers. The proposed testing methods focus on applications running either in a single Docker container or in multiple Docker containers on the same host operating system.

Subsequently, the work introduces the framework for testing software running in Docker containers. The framework is implemented in Bash command language, it consists of the core module that provides basic functionality and it is designed to be extensible with an additional custom code in form of a plugin. One of the proposed methods of testing applications in Docker containers was chosen for implementation of two plugins (default and network plugin), which utilize the functionality provided by the framework.

The framework should automate the repetitive work needed to be done when setting up testing environments, copying tests and gathering results of testing from Docker containers. Thanks to its extensibility through plugins, the support for other testing methods and different use cases can be added later to extend the framework functionality. The framework commands workflow is demonstrated on two use cases: the use case where a tested software is running in a single Docker container, and the use case where a tested software is running in multiple Docker containers.

Finally, the framework can be incorporated into larger testing suites targeting software containers validation and testing. In the future, snapshotting of Docker containers, or printing detailed differences of changed files in a container after testing can be added to the framework. Another possible improvement might be the ability to restart tests from the host system without the need to enter the test container.

Bibliography

[1] Adrian Mouat. Swarm v. Fleet v. Kubernetes v. Mesos. URL: https://www.oreilly.com/ideas/swarm-v-fleet-v-kubernetes-v-mesos, 2015 [Accessed: 2016-04-18].

[2] Akshay Karle. Operating System Containers vs. Application Containers. URL: https://blog.risingstack.com/operating-system-containers-vs-application-containers/, 2015 [Accessed: 2016-01-01].

[3] Andrew Baumann and others. Drawbridge. URL: http://research.microsoft.com/en-us/projects/drawbridge/, 2016 [Accessed: 2016-01-08].

[4] Barbora Ančincová and Tomáš Čapek. Red Hat Enterprise Linux 7: SELinux User's and Administrator's Guide. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/SELinux_Users_and_Administrators_Guide/, 2015 [Accessed: 2016-01-08].

[5] Bernard Golden. *Virtualization for Dummies*. Wiley Publishing, Inc., 2011. ISBN 978-0470943311.

[6] Charles P. Wright and Erez Zadok. Unionfs: Bringing Filesystems Together. URL: http://www.linuxjournal.com/article/7714, 2004 [Accessed: 2016-01-08].

[7] CoreOS developers. rkt Documentation. URL: https://coreos.com/rkt/docs/latest/, 2016 [Accessed: 2016-01-08].

[8] David Linthicum. Docker leads the container technology charge in cloud. URL: http://searchcloudcomputing.techtarget.com/feature/Docker-leads-the-container-technology-charge-in-cloud, 2014 [Accessed: 2016-01-04].

[9] Docker Inc. Docker Docs. URL: https://docs.docker.com/, 2015 [Accessed: 2015-12-26].

[10] Docker Inc. Container Specification – v1. URL: https://github.com/docker/libcontainer/blob/master/SPEC.md, 2016 [Accessed: 2016-04-15].

[11] Docker Inc. Seccomp security profiles for Docker. URL: https://github.com/docker/docker/blob/master/docs/security/seccomp.md, 2016 [Accessed: 2016-04-15].

[12] Docker, Inc. and others. docker-library. URL: https://github.com/docker-library, 2016 [Accessed: 2016-01-08].

[13] Geeks Hub. Types of Server Virtualization. URL: http://www.geeks-hub.com/types-of-server-virtualization/, 2016 [Accessed: 2016-04-12].

[14] Google Inc. Kubernetes Documentation. URL: http://kubernetes.io/v1.1/docs/, 2015 [Accessed: 2016-01-11].

[15] Google Inc. The Go programming language documentation. URL: https://golang.org/doc/, 2016 [Accessed: 2016-01-08].

[16] Kernel developers. Linux Containers. URL: https://linuxcontainers.org/, 2015 [Accessed: 2015-12-26].

[17] Linux kernel developers. SECure COMPuting with filters. URL: https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt, 2015 [Accessed: 2016-01-08].

[18] ManKier. docker_selinux man page. URL: https://www.mankier.com/8/docker_selinux, 2016 [Accessed: 2016-04-14].

[19] Michael Kerrisk and others. The Linux man-pages project. URL: https://www.kernel.org/doc/man-pages/, 2016 [Accessed: 2016-01-08].

[20] Open Virtualization Alliance. KVM FAQ. URL: http://www.linux-kvm.org/page/FAQ, 2016 [Accessed: 2016-04-12].

[21] Oracle Corporation. Oracle VM VirtualBox User Manual. URL: https://www.virtualbox.org/manual, 2016 [Accessed: 2016-04-12].

[22] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 978-0521880381.

[23] Peter Ondrejka and Douglas Silas and Martin Prpič and Rüdiger Landmann. Red Hat Enterprise Linux 7: Resource Management Guide. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Resource_Management_Guide/, 2015 [Accessed: 2016-01-08].

[24] Petr Müller et al. BeakerLib. URL: https://fedorahosted.org/beakerlib/, 2016 [Accessed: 2016-05-12].

[25] Pierre Bourque and Richard E. Fairley. *Guide to the Software Engineering Body of Knowledge – SWEBOK*. IEEE Press, 2014. ISBN 978-0769551661.

[26] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. URL: http://doi.acm.org/10.1145/361011.361073, 1974 [Accessed: 2016-04-12].

[27] Project Atomic. Docker and SELinux. URL: http://www.projectatomic.io/docs/docker-and-selinux/, 2016 [Accessed: 2016-04-14].

[28] Red Hat Atomic Host Documentation Team. Running super privileged containers. URL: https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/7/getting-started-with-containers/chapter-9-running-super-privileged-containers, 2016 [Accessed: 2016-04-17].

[29] Red Hat, Inc. and others. Red Hat Enterprise Linux 7: Logical Volume Manager Administration. URL: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Logical_Volume_Manager_Administration/device_mapper.html, 2016 [Accessed: 2016-01-08].

[30] Scott Hogg. Software Containers: Used More Frequently than Most Realize. URL: http://www.networkworld.com/article/2226996/cisco-subnet/software-containers--used-more-frequently-than-most-realize.html, 2014 [Accessed: 2015-12-26].

[31] Scott Johnston. Docker Hub Official Repos: Announcing Language Stacks. URL: https://blog.docker.com/2014/09/docker-hub-official-repos-announcing-language-stacks/, 2014 [Accessed: 2016-01-08].

[32] supervisord.org. Supervisor: A Process Control System. URL: http://supervisord.org/, 2016 [Accessed: 2016-04-27].

[33] The Xen Project. Xen Project Software Overview. URL: http://wiki.xen.org/wiki/Xen_Overview, 2015 [Accessed: 2015-12-29].

[34] Wikipedia. Operating-system-level virtualization — Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/wiki/Operating-system-level_virtualization#Implementations, 2016 [Accessed: 2016-04-13].