ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA (DISI)

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Mobile Systems

Fast Docker container deployment in Fog computing infrastructures

CANDIDATO

Lorenzo Civolani

RELATORE

Prof. Paolo Bellavista

CORRELATORE

Prof. Guillaume Pierre

Anno Accademico 2017/2018

Sessione III

Ringraziamenti

Desidero esprimere la mia gratitudine al professor Paolo Bellavista per il prezioso contributo alla mia formazione. Ringrazio Guillaume Pierre e tutta l'equipe Myriads del centro INRIA Rennes – Bretagne Atlantique per avermi accolto con entusiasmo e aiutato con pazienza durante la preparazione di questo lavoro.

Sono grato ai compagni di Università di Bologna, Mittuniversitetet di Sundsvall e Université de Rennes 1 per il raro valore della loro amicizia, e ad Elisabetta per il supporto che non è mai mancato.

Grazie alla mia famiglia per avermi permesso di intraprendere questo percorso di studi, per la condivisione dei momenti felici e il sostegno nei periodi difficili.

Grazie, infine, a me stesso, per la fatica, le rinunce e l'impegno.

Contents

Abstract

I contenitori software, meglio noti come *container*, realizzano ambienti virtuali in cui molteplici applicazioni possono eseguire senza il rischio di interferire fra di loro. L'efficienza e la semplicità dell'approccio hanno contribuito al forte incremento della popolarità dei contaier, e, tra le varie implementazioni disponibili, Docker è di gran lunga quella più diffusa. Sfortunatamente, a causa delle loro grandi dimensioni, il processo di *deployment* di un container da un registro remoto verso una macchina in locale tende a richiedere tempi lunghi. La lentezza di questa operazione è particolarmente svantaggiosa in un'architettura Fog computing, dove i servizi devono muoversi da un nodo all'altro in risposta alla mobilità degli utenti. Tra l'altro, l'impiego di server a basse prestazioni tipico di tale paradigma rischia di aggravare ulteriormente i ritardi. Questa tesi presenta FogDocker, un sistema che propone un approccio originale all'operazione di download delle immagini Docker con l'obiettivo di ridurre il tempo necessario per avviare un container. L'idea centrale del lavoro è di scaricare soltanto il contenuto essenziale per l'esecuzione del container e procedere immediatamente con l'avvio; poi, in un secondo momento, mentre l'applicazione è già al lavoro, il sistema può proseguire col recupero della restante parte dell'immagine. I risultati sperimentali confermano come FogDocker sia in grado di raggiungere una riduzione notevole del tempo necessario per avviare un container. Tale ottimizzazione si rivela essere particolarmente marcata quando applicata in un contesto a risorse computazionali limitate. I risultati ottenuti dal nostro sistema promettono di agevolare l'adozione dei software container nelle architetture di Fog computing, dove la rapidità di deployment è un fattore di vitale importanza.

Introduction

Software containers provide isolated virtual environments in which applications can execute without interfering with each other. Instead of simulating a full hardware platform, as virtual machines do, containers realize a virtualization at the operating-system level, which effectively reduces the overhead cost. As a consequence of the efficiency and simplicity of its approach, this technology has been getting increasingly popular in the last few years [7] and, among the different implementations of software containers, Docker [15] is without any doubt the most widespread. In Docker, a container represents the running instance of an image, which in turn can be seen as a snapshot file holding all the information about a container's content and configuration. When the user asks for an image which is not present in the local cache, the deployment process takes place: the image is downloaded from the public repository, decompressed and then run as a container.

Unfortunately, because of the growing size of Docker images, the deployment phase can become considerably long. The issue of slow deployment is particularly relevant in a Fog computing architecture, where the usage of resource-constrained devices and the slow public Internet network exacerbate the delays. In such context, for example, a popular containerized web server can take more than 60 seconds to be ready to start. However, speed is crucial in the Fog computing paradigm, where we must be able to deploy containers rapidly in order to support user mobility and fulfil the requirements of latency-sensitive applications. Large delays in the procedure may prevent us from starting a new instance of a service before the requiring user has left the vicinity of an access-point. It is therefore evident that the slowness of the container pull is a serious problem that must be addressed.

In this work we present FogDocker, a new software component that enables an original approach to the image pull in order to reduce the overall time needed to

start a container. FogDocker leverages the fact that, in most cases, a container does not need all of its files during execution: we show that, although images tend to be very large in terms of size, it is unlikely that a container will access the totality of its content during the first phases of a typical usage. The key idea of FogDocker is then to deploy the essential files first, let the container start its execution, and then asynchronously retrieve the rest of the image in the background while the container is already at work.

To design the aforementioned solution, many challenges have to be faced. First of all, it is necessary to find a reliable criterion to identify which files are essential for the execution of the container and which, instead, can be retrieved during a second phase. In addition, the overall reorganization of the pull operation implies to insert some changes in the code-base of Docker, which is a hard task given the considerable size of the project. Last but not least, our system must work not just on a typical scenario, but also in case of an unusual one. Assuming that a containerized application tries to access a file whose download has been delayed, the execution should not fail; on the contrary, we have to pause the process transparently until the file is finally retrieved.

The rest of this thesis is organized as follows. Chapter 1 describes the context of our work and formalizes the problem that we want to address. Chapter 2 outlines the main concepts of the technological background and presents the related work in recent literature. In Chapter 3 we proceed to the core of this project by describing the design of the system. Finally, Chapter 4 provides experimental results that prove the utility of our work and suggests some directions for future improvements.

Chapter 1

Overview

Since the early days of the Internet, the number of devices connected to the network has experienced a constant increase. In the last years, in particular, the popularity of Internet of Things (IoT) and mobile devices[1] has seen an unprecedented boost: it is estimated that by 2030 around 500 billion devices will be connected to the Internet [14]. Because of their small and/or portable nature, IoT and mobile devices do not usually have enough resources to afford considerable computational tasks on the spot; as a consequence, the burden is often delegated to external servers which have all the means to handle the process. In addition, this new kind of devices may bring about novel applications that require low-latency interactions with the server in order to offer a real-time experience to the user. For example, bringing artificial intelligence techniques to embedded devices may involve generating a continuous flow of data that should be processed quickly in order to provide prompt reactions.

The goal of this chapter is to illustrate the context in which this work has been carried out. First of all, we show that traditional solutions such as Cloud computing are not able to cope with the demands of novel applications in the IoT context. Then, we introduce Fog computing, an innovative paradigm that vows to fulfill such requirements by bringing computation close to the user. After examining its architecture, we explain the advantages that virtualization can bring about. We consider the two main possibilities for virtualization, i.e. virtual machines and

[1]Examples of IoT devices include various types of sensors, home appliances and industrial machinery that are provided with an Internet connection; a smartphone, on the other hand, may well be considered as the most common instance of a mobile device.

software containers, and we compare their pros and cons. After concluding that the latter is the most appropriate choice in a Fog computing architecture, we show that there is considerable space for possible improvements and we outline the way this work intends to achieve them.

1.1 Cloud computing

Cloud computing is the traditional solution to deliver computing services over the Internet. Thanks to it, a connected device can make use of hardware, storage, database, software and other services that are not available locally. In Cloud computing, remote resources are usually hosted in few datacenters around the world, on high-end machines that are managed by specialized vendors. The Cloud computing paradigm has several advantages. For instance, users are able to concentrate on the business logic rather than on the technical aspects of configuring and maintaining local machinery. In addition, externalizing services is economically convenient, because it allows to pay just for the resources that are actually employed, without bearing the fixed costs of a home server.

However, it has been shown that conventional Cloud computing does not meet the requirements of IoT applications [20]. First of all, it is hard to concentrate in a single geographical position all the storage and processing resources that are necessary in order to deal with billions of IoT devices. Moreover, the servers are generally situated far from the proximity of the end devices, causing an increase in the round-trip delay of the requests. Because of their centralized nature, Cloud datacenters can also experience serious network congestions; the resulting high latency in service delivery and service quality degradation would make it hard to cope with the requirements of real-time and latency-sensitive applications. Another issue relates to network unavailability, as some critical services may be required to run even when the remote cloud server or regional router devices are not functioning. Last but not least, some concerns may arise as far as the security of the data is concerned. Sending personal information through long-distance network connections, and storing it remotely, can potentially increase the possibilities of malicious attacks.

In order to address the technical challenge of integrating IoT and the Cloud, a new concept called Fog computing has been proposed.

1.2 Fog computing

Fog computing was first introduced by Cisco in 2012 as a concept that "extends the Cloud computing paradigm to the edge of the network" [6]. It is important to notice that Fog computing is not an alternative to Cloud computing, but rather a promising extension of it: its main intent is to add distributed storage and computational facilities in order to bring Cloud-based services closer to the end devices. In other words, Fog computing accomplishes a distribution of critical core functionalities such as storage, computation and decision making, by bringing them close to the place where the input data is produced and the output data is consumed.

Different kinds of industries are adopting Fog computing as a central paradigm, since its horizontal architecture is able to support multiple application domains. For example, in the context of smart cities data generated by vehicles could be collected and analyzed locally to dynamically adapt the policies of traffic control systems. Also, entertainment applications running on smartphones may need quick interaction with nearby Fog nodes in order to provide augmented-reality experiences to the user. Last but not least, factories may be able to perform an effective monitoring of their machinery by collecting and managing the continuous flow of data coming from appropriately installed sensors.

Given the considerable success of the architecture, great efforts are being made towards standardization. The Open Fog Consortium [10] is an association that brings together companies and academic institutions into an open environment where heterogeneous participants can stimulate the process of innovation and standardization. The main goal is to create an reference architecture for Fog computing that can enable and encourage a widespread adoption of the paradigm across the most diverse application domains. The reference architecture from the OpenFog Consortium has been adopted as the official standard by IEEE in 2018 [3].

Fog computing can be seen as a three-layer architecture in which a new intermediate layer of nodes interposes itself between the IoT devices and the Cloud datacentres. The intermediate layer is composed of geographically-distributed gateways which are typically positioned across densely populated network localities. Fog nodes are usually composed of traditional network components (such as routers, switches, proxy servers and base stations), which are provided with the appropriate storage and computing resources; the usage of networking devices instead of ad-hoc servers

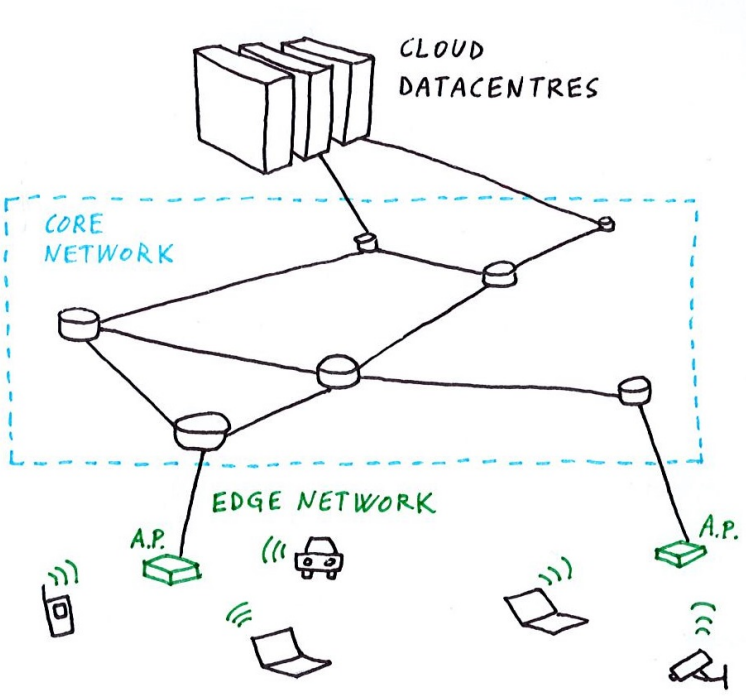


Figure 1.1: Computation domain of Fog computing

greatly benefits the pervasiveness of the environment and makes it possible to achieve the closest proximity to the users. Figure 1.1 represents the organization of a Fog computing environment.

In the following sections we are going to examine the advantages and the challenges of this approach.

1.2.1 Advantages

Fog computing offers several advantages which help meet the requirements of the applications for the IoT and mobile world. Here are some of them:

- The nodes can be placed in close **proximity** to the user devices. For instance, a Fog server might be located in the same router which is providing the wireless connection to a smartphone. As a result, the latency for service delivery can be greatly reduced, with considerable benefits for real-time applications (e.g. augmented-reality); moreover, local computation can be performed for those applications that generate data which is relevant only in the vicinity (e.g. IoT-based data analytics).

- The deployment of the servers can be planned in order to mirror the **geographical distribution** of the users. By placing the nodes in the areas where their services are needed the most, computational resources can be employed more effectively.
- **Location awareness** is facilitated. The pervasiveness of the networking devices which could be turned into a Fog node makes it easier to infer a good estimate of the current position of a user.
- **Scalability** improves. Contrary to a centralized service provider, distributed infrastructure can be easily extended in order to deal with an increase of the demand by the users,

Thanks to the above-mentioned aspects, Fog computing is considered more suitable for IoT and mobile environment, especially if compared with the adoption of Cloud computing alone.

1.2.2 Challenges

Even though Fog computing is widely considered a promising evolution towards the needs of the latest technology devices and new types of applications, many challenges are still open. To begin with, a good criterion for the **selection of the nodes** for a Fog computing infrastructure is crucial. Factors such as execution environment, application requirements and patterns of user mobility must be taken into account when evaluating the possible places of deployment.

Supplying network components with **general-purpose computational facilities** could also be a hard task. As we said, traditional network components can be converted into potential nodes for a Fog computing infrastructure. However, these devices are designed for specific tasks, and may lack the required hardware and software support for general purpose computing. Provisioning the means for general purpose computing, while preserving their traditional activities, can be challenging.

Another challenge relates to the fact that the infrastructure will be likely deployed in a **resource-constrained** environment. The large number of nodes of a Fog computing infrastructure imposes the adoption of simple commodity machines, with weak storage and computational resources; its broad geographical distribution suggests

that they will be connected between each other, and to the Cloud servers, through the public Internet network. Developing applications in a resource-restricted environment will be harder compared to conventional datacentres.

Since the infrastructure is designed to rely on conventional networking components, **security issues** may arise. In fact, nodes that were designed for simpler tasks could become vulnerable to malicious attacks. Furthermore, privacy and authentication in such a distributed environment would be hard to ensure without inducing a significant degradation of the performances.

Techniques for **efficient resource provisioning** are of the greatest importance, and time is certainly one of the key factors to be taken into account. Computation time must be reduced as much as possible in order to provide low-latency responses that can support real-time interactions with end devices.

Furthermore, it is necessary that the infrastructure supports the **mobility** of users to a good extent. A fast deployment of the computational resources on the nodes is essential to follow the users as much as possible while they move. The aim is to keep an acceptable latency rate even when a certain device is changing its point of attachment to the network.

1.3 Virtualization for Fog computing

Among the many challenges of Fog computing, supporting mobility stands out as a major point: it is very important that the services be able to move swiftly between the different node of the infrastructure, trying to deliver as much as possible a continuous experience for the user. Also, a Fog infrastructure should support multi-tenancy: it should be easy and efficient to deploy and run multiple services simultaneously on the same point of presence of the network. For these reasons, the use of virtualization techniques is widely considered as an appropriate choice for the organization of the services in a Fog environment.

The term *virtualization* refers to the act of providing a client (in our case, a service) with a view over a certain set of resources (in our case, computational ones) which is different from the actual one. Virtualization is achieved by introducing an indirection level between the logical and the physical view of the platform. The objective

is to decouple the behavior of the resources offered by a system from their specific implementation.

Depending on the level where the abstraction layer is introduced, different kinds of virtualization are possible. Virtualization at process-level realizes a multitasking system by letting multiple processes execute concurrently on a shared platform. Virtualization at memory-level provides each running processes with a virtual address space which is independent from the address on the physical memory. Virtualization at system-level lets a single hardware platform be shared among many virtual machines, each of which is managed by a different operating system.

The rest of this section takes into account the popular approach of **virtual machines** and the more recent trend of **containerization** techniques. The pros and cons of both technologies will be evaluated in relation to the Fog computing paradigm, which is the context of this work.

1.3.1 Virtual machines

Virtual machines are a traditional form of virtualization which belongs to the category of system-level virtualization [21]. The aim of the technology is to let multiple virtual machines execute concurrently even if they share a common hardware platform. This decoupling is realized by a software component called Virtual Machine Monitor (VMM) or hypervisor, which works as a mediator in the interactions between the virtual machines and the underlying machinery. The VMM provides an abstraction of the physical hardware by either simulating its behavior or by performing a translation between the exposed and the real instruction set. Its responsibility is to provide the virtual machines with all the virtual resources needed for operation (e.g. CPU, memory and I/O devices).

The adoption of virtual machines in Fog computing can bring all the advantages of working with virtual environments. First of all, the possibility to deploy a service together with its execution environment greatly simplifies the process of start-up. Libraries, configuration files and system variables are some examples of what can be successfully encapsulated around an application and brought to the node where it is put into execution. Furthermore, as each virtual machines is managed by its own operating system, it is even possible to deploy applications which have been designed

for distinct platforms. Virtual machines provide also excellent isolation properties. Every service can run in a "sandbox" which avoids by design any interference with the other tasks in execution on a given Fog node. In case of failure or cyber attack, for example, the rest of the services would not be affected.

In spite of their benefits, virtual machines show substantial limitations as well. One of the main drawbacks of virtual machines is the degradation of performances. The intermediate layer introduced by the abstraction technique generates a considerable overhead. The impact, which is well noticeable on desktop machines, would be even worse in resource-constrained devices such as Fog nodes. Secondly, the size of a virtual machine is usually quite large. Indeed, a snapshot file, which contains all the information regarding a specific machine, can take up to several gigabytes of disk space. It is likely that the storage capabilities of a simple Point-of-presence would not be able to cope with such requirements. As a consequence of their size, transferring virtual machines between physical hosts can be a very slow operation. Since the nodes of a Fog computing infrastructure have no high-speed dedicated network nor performing hardware, the deployment of a service can suffer unacceptable delays. The issue of slow transfer times has been addressed in the case of live migration by carrying out the majority of the process while the operating system continues to run [9]. However, the problems related with the performance overhead and resource consumption still remain.

Some works propose the usage of VMs for the Fog computing architecture, claiming that the prospect of running different operating systems on the same node will ensure an degree of flexibility otherwise impossible to achieve [23]. However, the drawbacks that have just been outlined make it hard to implement such a technology on simple Points-of-presence without neglecting the demands of novel latency-sensitive applications. All in all, our analysis seems to suggest a more lightweight approach to the topic of virtualization.

1.3.2 Containers

Software containerization is a more recent trend in virtualization technology. A container is a standardized unit of software that contains the code of an application along with all of its dependencies. It is an executable software package that includes the binaries, a runtime, system tools, libraries and settings, configuration files, en-

vironent variables and so forth. As a result, an application can be abstracted from the environment in which it executes.

Along with enumerating its features, it is important to explain how a container internally operates, for the word itself does not hint at anything precise. A container is a set of processes that operate inside an environment which is isolated from the one of the host machine. They use their own file system, have their own hostname and cannot interact with the processes who live outside of the isolated environment. In Linux, in order to realize such an isolated environment around a process, containerization frameworks make use of some features of the kernel: **namespaces** and **control groups**.

Namespaces are a feature that partitions kernel resources so that distinct sets of processes see and use different set of resources. As a result, each process (or group of processes) can have a unique view on the resource. Different kinds of namespaces can be accomplished depending on the resource which is being partitioned: mount namespaces can isolate the set of file system mount points seen by a group of processes; PID namespaces isolate the process ID number space; a network namespace provides a private set of IP addresses, ports, routing table and other network-related resources. Namespaces let us choose what a process can see.

Control groups, on the other hand, are a kernel feature that limits, accounts for and isolates the resource usage of a collection of processes. It is possible to control the usage of CPU, memory, disk, network and so on. In other words, control groups let us decide how many resources a process is allowed to use.

Namespaces and control groups are powerful functionalities offered by the OS kernel. When they are used to isolate a process from the rest, then we call it a “container”.

The popularity of software containers has seen a considerable increase since the introduction of Docker (2013), a software platform that allows to build, run and manage software containers [15]. Other container solutions (such as LXC, rkt, and Windows Containers) exist. However, Docker is the most popular one, and is on its way to become a de facto standard for containerization [7]. For this reason, from now on we will no longer distinguish between the two.

As outlined in Figure 1.2, there are similarities and differences between virtual machines and containers. Like virtual machines, Docker containers are:

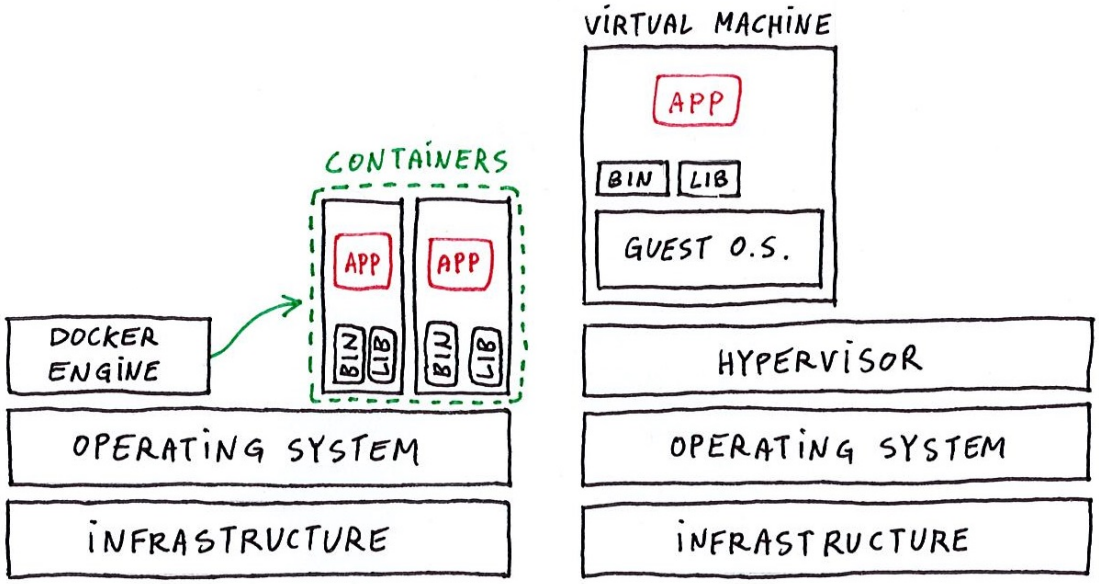


Figure 1.2: Comparison between software containers and virtual machines

- **standalone**, because they incorporate everything needed to run an application, avoiding external dependencies;
- **flexible**, since they can be used to encapsulate the most diverse kinds of applications;
- **portable**, because they allow a seamless deployment of the conainers across different machines; a containerized application will always run the same, regardless of the infrastructure on which it is arranged;

Unlike virtual machines, Docker containers are:

- **faster**, at boot and run time. Virtual machines virtualize the whole hardware architecture and contain, in addition to the application itself and all the libraries it needs, a complete OS; on the contrary, containers virtualize at the operating system level, thus sharing the same Linux kernel at run time. As a result, there is no need to wait for the boot of any OS before launching the application. Furthermore, the overhead of the Docker platform at run-time is negligible. It has been shown that the low computing footprint of containers makes CPU, memory, storage and network usage similar to the one resulting from an execution on the bare-metal operating system [16].

- **easier to create and manage**, as it is not necessary to install and maintain any additional OS.
- possibly **less secure**. A running container could potentially damage the kernel through some harmful operation, with consequences on the rest of the system; on the other hand, the crash of a virtual machine would not influence the others.
- **lighter**. First of all, Docker containers generally use less space than virtual machines: whereas the former can take tens or hundreds of megabytes, the latter can require several gigabytes. Secondly, Docker allows reusability of the data among different containers. The file-system content of every container is organized in layers which can be effectively shared among multiple instances. We refer to Chapter 2 for a thorough exposition of this important Docker feature. Here, it is important to know that it allows a more effective usage of the storage memory.

1.4 The problem of slow pull

Even though containers are more suitable than virtual machines for Fog computing, one big issue still remains: Docker images can reach very large sizes. Table 1.1 reports statistics regarding some of the most popular images from the public Docker repository.

Image	Size (MB)
Apache httpd	130
MySQL	470
Golang	816

Table 1.1: Size of popular Docker images at the time of writing

As a consequence of their size, the pull time of images grows. In fact, it has been shown that such operation is by far the longest one during container start-up [1]. Furthermore, the slowness of the container deployment would be even more acute in a Fog computer environment, where we make use of low-end devices and slow public Internet networks. Such delays are a serious drawback of the Docker system:

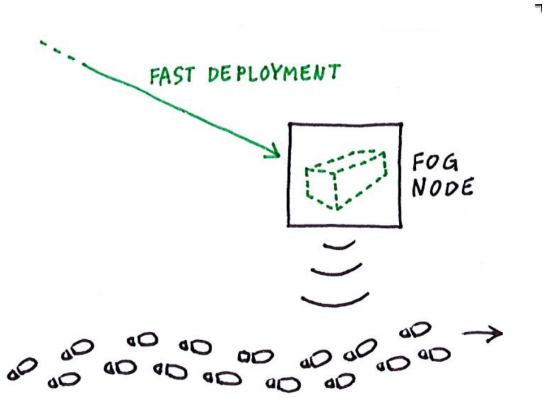


Figure 1.3: Scheme of a deployment as a reaction to user's mobility

pull time must be reduced, otherwise it would be hard to meet the requirements of real-time applications as well as follow the users along their movements, like outlined by the example in Figure 1.3. It is clear that a solution to this problem must be found, and this thesis tries to give a contribution towards the right direction.

Several works have concentrated on the evaluation of the usage of Docker containers in a distributed computing environment such as the Fog. Most of them consider this technology as a promising solution, but they do not take into account the issue and implications of potentially slow pulls. Bellavista et al. [4] proved that containerization techniques are suitable for implementing Fog nodes even in the case of machines with limited resource availability. The work is set in a typical distributed scenario in the domain of Smart Connected Vehicles, where Fog nodes placed on public vehicles continuously collect and process data coming from multiple sources. The Fog nodes are implemented on low-end Raspberry Pi 1 machines. First, they compute the overhead introduced by the usage of Docker containers; then, they evaluate the performance effect of executing multiple instances at the same time, which is a likely use case of the Fog computing paradigm. Even though, as expected, native execution outperforms the containerized approach, results show that the overhead is limited. Moreover, the cost grows linearly with the number of running containers, which proves the scalability of the approach. However, the study assumes that the images are already present in the local cache, and does not examine the time needed for the pull operation. Ismail et al. [16], on the contrary, take into account this parameter. They emphasize how the reduced size of containers allows a swift deployment across the distributed infrastructure. However, they do not highlight the necessity of improving the deployment time of Docker images in

order to better meet the requirements of latency-sensitive applications.

All in all, Docker containers offer an environment which reproduces as closely as possible the one of a virtual machine, but without the overhead that comes with running a separate OS and simulating all the hardware. Than being said, there are problems that ought to be addressed in order to make the technology more attractive for the Fog computing systems. The deployment time of containers shall be reduced, with the final objective of delivering a tool which can help meet the requirements of real-time applications. Our goal is to provide a viable solution to this obstacle.

1.5 Objective of the work

The objective of this work is to reduce the duration of the deployment of Docker images from a repository to a worker machine. In this work we present FogDocker, a new software component that enables an original approach to the image pull operation in order to reduce the overall time needed to start a container. FogDocker leverages the fact that, in most cases, a container does not need all of its files during execution: in the paper we show that, although images tend to be very large in terms of size, it is unlikely that a container will access the totality of its content during the first phases of a typical usage. The key idea of FogDocker is then to deploy the essential files first, let the container start its execution, and then asynchronously retrieve the rest of the image at run-time while the container is already at work. The next chapter goes through the technical background which is necessary to understand the design of our solution.

Chapter 2

Technical background

The popularity of software containers has been constantly increasing in the last years. Among the container runtimes on the market at the moment, Docker is without any doubt the leader. In recent studies [7], companies that monitor infrastructures and applications have presented some trends regarding the adoption of the platform: Docker is the preferred container runtime in more than 80% of the hosts, and the number of units running on each machine climbs by 50% each year.

This chapter provides the necessary technical background in order to understand the design of our system. First of all, we introduce Docker by outlining its concepts, architecture and components, along with some usage examples. Then, we present some papers in the literature that aim to address the same problem of this work.

2.1 Docker

Docker is a platform that allows to create, deploy and run software containers. Containers are standardized units of software that tie together an application with everythings it needs for the execution, and let a client treat them in a uniform way. The concept of container is older than Docker itself, and dates back to the introduction of LXC in 2008; however, Docker makes it easy to work with them, and offers a set of tools that provide the user with a simple interface to deploy and manage applications inside containers. For example, a simple command like `docker run -it alpine sh` will download the Alpine Linux distribution, initialize

a file system, initialize the necessary namespaces and control groups for container isolation and return an interactive Unix shell to the user. As a result of its simplicity of usage, it is since Docker's introduction in 2013 that the popularity of software containers has experienced a considerable increase.

By using Docker it is certainly possible to benefit from all the advantages that derive from the adoption of any containerization technology. First of all, containers are **self-contained**, since they contain everything needed for seamless execution. As a consequence, developers can focus on the business logic of the application without worrying about the details of each system that will host it. In addition, containers are **portable**, as they can be easily moved across multiple machines. A system administrator, for example, can carry out the deployment of a system quickly. Containers are also **lightweight**, thanks to their low resource requirements and performance overhead. Therefore, the number of physical machines needed to run a certain set of isolated applications can be effectively reduced.

Besides, choosing Docker brings the advantages related to the selection of a successful and widespread technology. Thanks to the work of Docker's large community, many popular applications and frameworks have been containerized and are available for download through public repositories. Examples that can be found in the official Docker repository are: the Apache web server, Redis, the Java Virtual Machine, MySQL and so forth. Hence, users can afford to download and execute applications in an easy fashion, without having to worry about providing the needed dependencies or setting an appropriate configuration of the system.

The architecture of Docker follows a client-server organization. The Docker client is a command-line tool that allows user interaction with the system; in response to a command typed by the user, the client sends a request to the daemon. The Docker daemon, in turn, is the background component in charge of performing the actual management of the containers; it is constantly listening for incoming requests, and handles them by performing the requested actions and transmitting back the result.

Communication between the two components is organized according to a REST API called Docker API. In a simple scenario, client and daemon would run on the same machine; in this case, requests and responses of the Docker API would transit over Unix sockets. However, a client could interact with a remote daemon as well; communication would then happen over a network interface. Figure 2.1 shows a

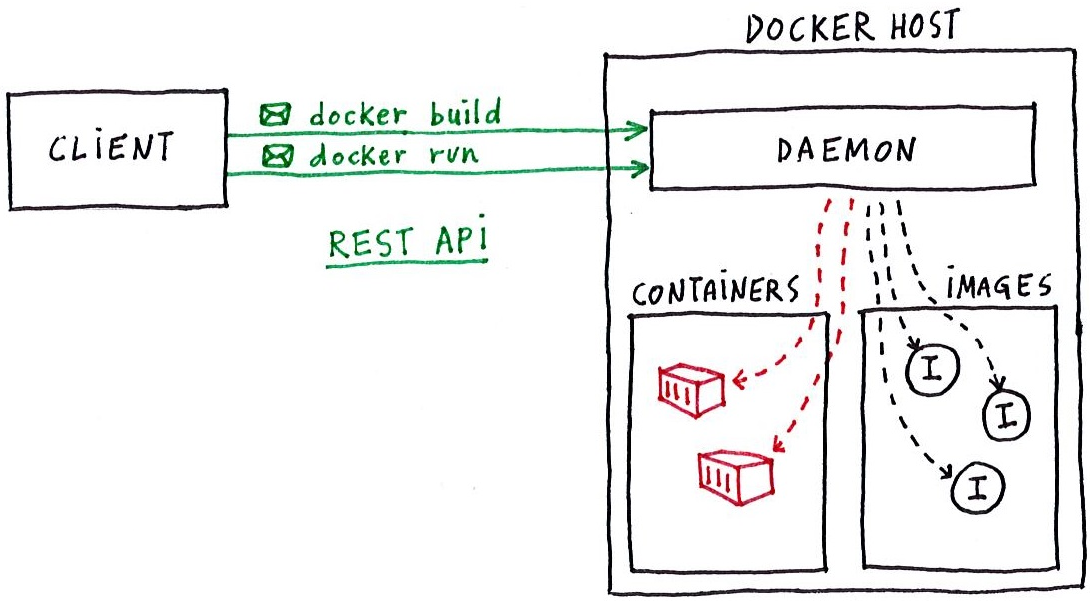


Figure 2.1: Outline of the architecture of the Docker engine

high-level schema of a Docker engine, i.e. the combination of a client and a daemon. All in all, the decoupling the two main components of the architecture provides a good degree of flexibility.

Besides the command-line client and the daemon, the last element of the basic architecture of Docker is the registry. The Docker registry is a component that realizes image sharing: daemons download them locally (pull) or upload them to the remote storage (push). A very common instance is Docker Hub, the public registry that everybody can use; here, for example, any user has access to the official images of popular containerized applications. Even though daemons are preconfigured to pull (and push) images from (and to) the Docker Hub, it is possible to run an independent instance of the registry as well.

2.2 Images

In Docker, an image is a read-only template that can be used as a basis to instantiate containers. It contains everything that is needed to execute the enclosed application, i.e. files, libraries, runtime, environment variables and configuration. Also, an image contains additional information regarding how to build a new container from it and how to start its execution. As an example of the relationship between an image and

a container, we can refer, to some extent, to the one existing between a class and an object in an object-oriented programming language. It is possible to list the images that are currently available on a Docker daemon by issuing the command `docker image ls` on the client. Listing 1 shows the resulting output.

```
$ docker image ls
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
redis               latest    82629e941a38   2 weeks ago      95MB
debian              latest    a0bd3e1c8f9e   2 weeks ago      101MB
golang              1.11.2    df6ac9d1bf64   2 months ago     774MB
lcivolani/glife     latest    d1566859e4c8   4 days ago       103MB
lcivolani/docs      latest    e14f07647839   9 days ago       55.8MB
```

Listing 1: Output of an "image list" command

As it can be seen from the output of the command, each Docker image is identified by means of the following strings:

- the name of a **repository**, which can be seen as a deposit where one or more versions of an image are stored. A Docker registry usually contains multiple repositories. Generally speaking, there are no restrictions on the format of the name; on Docker Hub, however, the following convention is adopted: official images have simple names (e.g. "redis"), whereas user-built ones shall incorporate their author in the opening (e.g. "lcivolani/glife");
- an optional **tag**, which can be used as a discriminator among the different versions of an image. If a tag is not specified, the default value "latest" is used to point to the most recent one.

There are different ways for a daemon to obtain an image, and the simplest one is to retrieve it from a registry. As we said, either the official Docker registry or a self-hosted one can be used for the purpose. The public Docker Hub, which is the one that is used by default, offers plenty of pre-built images that are available for free: they can be downloaded to the local storage by a daemon and then used to run a container. The operation of retrieving an image from a registry is called **pull**, and it involves the download of the data over the network. To pull an image, the command `docker pull NAME[:TAG]` can be issued through the client.

Another way to get an image is to build it locally, and to do so a special file called "Dockerfile" must be prepared.

2.2.1 Dockerfile

A Dockerfile is a text document that provides the instructions on how to build the image by listing one by one the actions that the daemon has to perform during the creation process. We can easily observe that a Dockerfile is written according syntax reported by Listing 2.

```
# Comment
INSTRUCTION arguments
```

Listing 2: Syntax of a Dockerfile

There are many kinds of instructions that can be included on a Dockerfile; the aim is to give the user the means to express the content of an image (usually an application's code and all of its dependencies) and the configuration of the environment (such as system variables, network parameters and so on). Some of the most useful instructions are reported in the following list:

- `FROM <image>` initializes the build procedure by setting the base image on top of which the changes have to be applied. A valid Dockerfile must begin with a `FROM` instruction. The special command `FROM scratch` can be used to indicate that no base image is required.
- `COPY <src>... <dest>` can be used to copy files or directories from the file system of the host at the path `<src>` into the file system of the container at the path `<dest>`.
- `ADD <src>... <dest>` is essentially a more powerful version of `COPY` which is able to handle both URLs and compressed archives as source of the transfer; the former will be downloaded from the network, whereas the latter will be unpacked.
- `RUN <command>` or `RUN ["executable", "arg", ...]` is used to execute an arbitrary command and commit the results into the file system of the con-

tainer. The two forms of this instruction have slightly different outcomes: the first one runs the command in a shell; the second one launches directly the executable. Typical use-cases of this instruction include the installation of software packages from a remote repository or the compilation of an application's source code.

- `ENV <key> <value>` sets the environment variable `<env>` to the value `<value>`. Variable set by this instructions will be available when a container is launched from the image.

- `CMD ["executable", "arg", ...]` is used to indicate a default starting point for the execution of a container, and it is usually the last instruction of a Dockerfile. For example, if we include `CMD ["echo", "hello world"]`, any container that is launched will emit the expected string and then terminate.

Once the Dockerfile has been prepared, the build process can start. As it has been shown, the client-server architecture of Docker prescribes that even if user interaction happens with the client, the actual execution of the operations takes place on the daemon. Since the two components are decoupled, and possibly execute on different machines, the Dockerfile and its context by it must be sent to the daemon; in the context, the daemon will find, for example, all the files and folders referenced by the instructions `COPY` and `RUN`. A common way to invoke the build process is by placing the Dockerfile and all necessary files into a new folder, and then use `docker build -t NAME[:TAG] /path/to/folder` on the command line. The option `-t` lets the user assign a name (and, optionally, a tag) to the image; the daemon will expect the Dockerfile to lie in the context.

A Docker registry allows not only to download images, but also to upload them. Locally built images can be pushed to a Docker registry with the command `docker push NAME[:TAG]`.

2.2.2 Building incremental images

The design of a Dockerfile is influenced by Docker's incremental approach to the task of image creation. Indeed, instead of starting from scratch every time, a new

```
FROM openjdk:7
COPY Main.java /usr/src/myapp
RUN javac /usr/src/myapp/Main.java
CMD ["java", "Main"]
```

Listing 3: A simple Dockerfile

image can be defined based on an existing one, providing only the additional changes that are necessary in order to obtain the new version. As a consequence, the `FROM` instruction must be the first of every Dockerfile.

In order to get the idea of the effectiveness of the approach, we propose the following example. Let us assume that we want to package a Java application with everything it needs to run, so that it can be easily ported across different machines. In this case, we would have to make sure that the file system contains not only the bytecode of the program, but also the Java Virtual Machine runtime that is able to execute it; furthermore, we would have to provide all the files, libraries and settings that a JVM expects to find when running on top of an operating system. In theory, we could proceed by preparing a Dockerfile that contains all the instructions needed to copy the base file system of a given Linux distribution, run the installer of the JVM and finally import the binaries of our application. However, the procedure would be tedious and, most importantly, it would have to be repeated for every Java application we want to containerize. A simpler and more flexible approach for the task is to use the official JVM image as a basis instead than starting from scratch; in this way, our Dockerfile has to specify only the incremental changes to be applied to the original version (i.e. copying the application itself), without reinventing the wheel every time. The following Listing 3 shows an example of Dockerfile that accomplishes the desired result.

2.2.3 Layered structure

In addition to the possibility of building a new image based on an existing one, the platform has another feature which promotes reusability at a finer level. Rather than being stored in a single file, the content of Docker images is organized in multiple layers; these layers are stacked on top of each other, forming a pile that represents

the initial content of the container's root file-system. The layers of a Docker image are generated by the daemon during the build process; in particular, most times an instruction of the Dockerfile is executed, the produced result is deposited into a new tier which is then piled on top of the stack. For example, let us consider again the previous case of an image holding a Java application and its dependencies: the `FROM openjdk:7` instruction, which establishes the starting point of the build process, will bring in all the layers of the original image; with `COPY Main.java /usr/src/myapp`, a new tier containing the source code of the application is added on top of the existing ones; `RUN javac /usr/src/myapp/Main.java` compiles it and puts the result into an additional layer; `CMD ["java", "Main"]`, on the contrary, does not generate any extra component: after all, it only defines a configuration parameter.

The layered approach has a twofold peculiarity which is able to promote reusability at a finer level. First of all, layers are immutable and read-only; containers that are instantiated from an image cannot alter the content of the underlying components. Second of all, they are referenced using an hash of their content, adopting a method which is called **content addressable storage**. When an image is pulled from a registry, the daemon computes an hash over the data of each layer and assigns the results to each of them respectively. Thus, equivalent layers will have uniform identifiers across multiple Docker machines, and, as a result, they can be easily shared among those images whose structure is alike. For example, let us assume the case in which ten Java applications have to be containerized; all the layers belonging to the base image `openjdk:7` can be effectively shared between them, since the only differences lie in the upper sections containing the code of the application itself. Content addressable storage of layers reduces redundancy to a large extent, and helps saving a considerable amount of disk space.

2.3 Containers

As it has been said, a container is a standard executble component that includes an application and all of its dependencies, such as runtime, system tools, libraries, files and settings. In Docker, a container is a running instance of an image. With the command `docker run IMAGE[:TAG]` the user can instruct the daemon to instantiate a new container and put in into execution. A container shares the kernel with the host machine, but is well isolated from it and from the other instances as well.

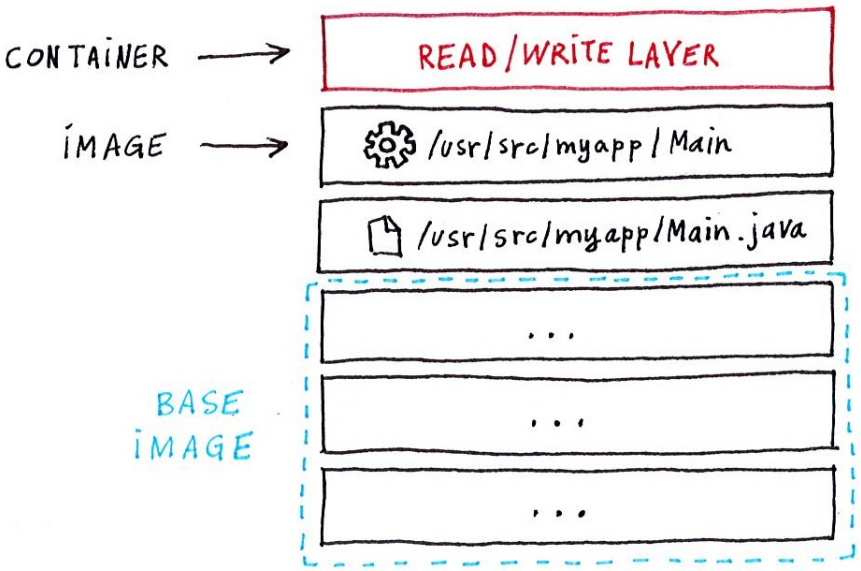


Figure 2.2: Structure of a container encasing a Java application

To fully understand how a Docker container works, it is important to show its internal structure and behavior. The previous section has shown that the file system of an image is composed by a stack of layers, each of which contains the set of differences from the layer beneath. Those layers are immutable: they cannot be altered, because they are indexed according to their content and they may be shared among multiple images. However, a running container which is restricted to a simple read-only access to the data would probably be of questionable utility. For this reason, when the Docker daemon creates a container, a new writable layer is added on top of the stack. The purpose of this layer, which is usually called "container layer", is to keep track of all the changes made by the running process. The running container is therefore provided with a unified view of all the tiers as its root file system: existing files will be accessed directly from the lower layers; creation of new files and modification or deletion of existing ones will be recorded in the upper-level writable component. A visual representation of the layers that compose a container is provided by Figure 2.2.

The implementation details about how the content of the layers is stored and managed on the disk of the host machine are handled by the **storage driver**, a pluggable component of the Docker daemon. Different implementations are available, each of them with its strengths and weaknesses that make it suitable for certain scenarios and inadequate for others; examples are overlay2, aufs, devicemapper, zfs and vfs.

In this work we are going to consider overlay2, since it is the preferred storage driver for modern versions of Docker running on all Linux distributions.

2.4 Overlay file system

Overlay2, which is the default storage driver for Docker, makes use of the Overlay file system (or Overlay FS) in order to realize a concrete implementation of the layered architecture for images and containers. Given that such layered architecture has a key role in the design of our system, it is crucial that an accurate inspection of Overlay FS is provided. The main idea of our solution is to deploy the essential files of an image first, and then let the container start its execution immediately. The presence of layers which are pulled independently by the daemon gives us the opportunity to think in terms of a base tier whose download must be prioritized against the others.

Seeing that the actual management of the layers is of paramount importance, we now delve into finer details over how it is performed by Overlay FS. In particular, two questions arise: first of all, it is not clear how it is possible to provide an unified view of the overall content to the root file system of the container; moreover, there is uncertainty about how a Docker container can operate given that the underlying layers cannot be modified. It turns out that the Overlay FS allows to emulate the union of multiple directory trees by presenting a merged view of them; in other words, the content of a set of folders is combined and the combination is showed into an additional one. Overlay FS is virtual, because it works as an abstraction layer that sits on top of more concrete file systems, such as the traditional ext4. Even though the base directory trees can reside in different underlying file systems, it is quite possible that both of them are in the same.

An instance of Overlay FS is configured with the following parameters:

- a set of read-only directories, called **lowerdir**, containing the content that ought to be joined; they are immutable by design, so that the original content is preserved;
- the **merged** directory, instead, provides the path in which the unified view over the lowerdirs will be mounted;

- finally, an additional writable directory called **upperdir** is provided; in order to support every file operation through the merged view without affecting the underlying immutable layers, the virtual file system needs a place in which to record the activities; this directory does fits the purpose.

Read operations on the merged directory are essentially redirected towards the original content of the appropriate layer. If a folder appears in multiple layers, then a merged version of them is presented; when a lookup is issued in such directory, all the underlying layers are read and the overall result list is returned. If, instead, a given file appears multiple times across the layers, then Overlay FS will reference the upper object.

On the other hand, modifications and deletions of original data are handled with specific techniques. As we have seen, objects in the merged directory are presented from the appropriate lower layer. However, when one of those files is opened in write mode, it is first copied up from the lower into the upper directory; once the operation is completed, the Overlay FS provides access to the new copy of the file in the upper directory. Since writing on a file of a lower layer requires a preliminary copy of it into the upper one, this technique is called **copy-on-write**. Overlay FS also supports remove operations preventing any change to the lower layers. When an existing file or directory is deleted, a **whiteout file** or an **opaque directory** with the same name is respectively created and inserted into the upper layer. We omit to discuss the implementation of these two "special" objects. Their purpose is to hide the original lower objects, as any directory in the lower layers with the same name will be ignored. As a result, Overlay FS is able to give the impression that the file (or folder) has been removed without actually modifying the lower immutable directories. Figure 2.3 represents a possible configuration of an instance of the file system. Elements on the same column are supposed to have the same name; a whiteout files is drawn in red.

This brief overview about the internals of Overlay FS allows us to finally understand how Docker makes use of it to concretely implement the management of the layers. Each layer in Docker corresponds to a directory within the path `/var/lib/docker/overlay2/` which hold all of its content. When a container is created from an image, Docker installs an additional directory in the same path in order to host the data of the new read-write layer. Then, a merged view of the image layers and the container layer is initialized through an appropriate instantiation

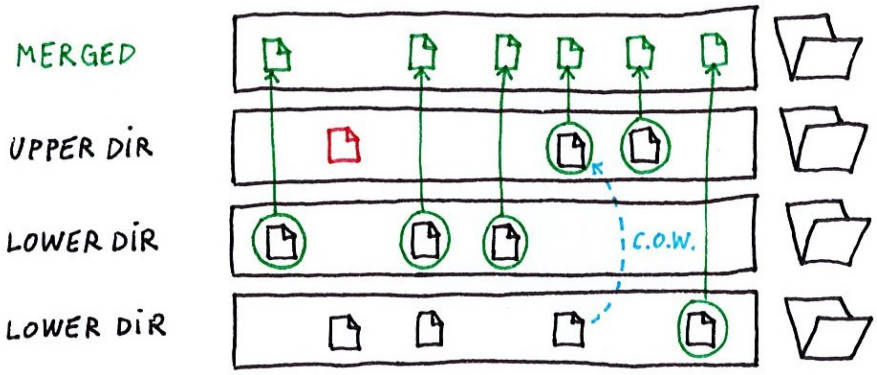


Figure 2.3: Example of an instance of Overlay FS

of Overlay FS: the image layers will be mounted as lower directories, whereas the container layer will be mounted as the upper directory. Finally, the resulting unified view is mounted into a further merged folder, which, in turn, becomes the root of the internal file system of the starting container. As a result, the process executing inside the container will enjoy a unitary perception of the contents of the underlying layers and will be able to modify its content; the virtual file system will transparently make sure that structure of the image is by no means affected, whatever the activity carried out by the application.

2.5 Related work

Since the introduction of Docker in 2013, the popularity of software containers has experienced a constant growth. The lightness of this virtualization technique makes the technology suitable for isolating multiple applications, or parts of it, even on machines with low computational resources. However, as it has been mentioned, there is still space for improvements. In particular, the time required to pull an image from a registry can tremendously grow when its size is large and the network connection is slow. In this regard, a growing body of literature has investigated multiple approaches to deal with the problem. The rest of the chapter shows some of the methods that have been put forward in recent years; after outlining their proposals, we will highlight the positive aspects and point out their limitations in relation with the Fog computing paradigm. This review of past literature aims to attest the importance of the issue that we want to address and to justify the novelty

of our approach.

2.5.1 Registry-side enhancements

Anwar et al. [2] recognize the critical role played by the Docker registry during the phase of container start-up. Since the machines hosting Docker daemons have limited memory, it is impossible to store a large number of images locally. As a consequence, many container starts begin with the retrieval of the appropriate image from a registry. However, given the large number of requests to fulfil and the generally considerable size of the data to be stored, retrieved and transferred, a Docker registry can easily become a performance bottleneck. The Docker Hub, for instance, hosts hundreds of terabytes of data and grows with many new public repositories every day. It has been shown that pulling an image from it can account for more than 70% of container start time. Therefore, it is clear that improving the performance of the registry is of extreme importance.

As a first step of the work, Anwar et al. study the workload of a Docker registry in a real-world scenario. The analysis is carried out on the IBM Cloud Container Registry, which well embodies the characteristics of a public-domain service: on the one hand, it is used by different clients, such as individual users, small and medium companies, enterprise businesses and government institutions; on the other hand, it stores images of the most diverse kind, such as Linux distributions, databases and frameworks. During the analysis, various kinds of log data related to image push and pull is collected. The monitoring phase lasts for 75 days, intercepting over 38 million requests handled by the registry system.

The results of the analysis provide useful insights which are then exploited to derive design implications for Docker Registries. For example, it is observed that the ratio between the number of pull and push requests handled by a registry varies according to its age: older registries experience more than 90% of pull requests; for newer ones, the figure decreases to around 70% of the total. Another interesting information deals with the layers composing the images, and shows that most of them are small in size: 65% of the layers weigh less than 1 MB, and 80% of them are under 10 MB. Last but not least, collected information highlights a solid correlation between the upload of an image and subsequent requests for manifest files and layers.

The paper takes advantage of the generally small size of the layers to introduce a **layer cache** on the registry. As it has been shown in the previous part of this chapter, layers are content addressable. A side effect of this characteristic is that it is not necessary to implement any cache invalidation mechanism. Since a change in a layer leads to a change of its ID, the older versions of the layer will no longer be accessed and will eventually be dismissed from the cache. A second design improvements introduces **layer prefetching**. In this case, the observations regarding the relationship between push and pull operations are exploited in order to predict which layers are most likely to be requested, and retrieve them in advance from the physical memory.

Results show that the peformances of the registry improve, and, consequently, the time needed to launch a container is also reduced. However, in spite of the valid optimizations that this work introduces on the registry, the advantages in a Fog computing scenario, which is the context of this work, is questionable. Indeed, once an image has been rapidly located, the full size of it has still to be transmitted over the network, which is one of the greatest performance bottleneck of a Fog computing infrastructure. In such a scenario, a more radical approach is paramount.

2.5.2 Cooperative Docker registries

Nathan et al. [17] recognize that the download of an image from a remote registry (such as Docker Hub) is a resource-intensive task, because it entails the transfer of a large amount of data over wide area networks. In order to reduce the provisioning time, they propose a new approach for the image management that promotes sharing among Docker hosts in the scenario of microservice-based applications deployment.

In **CoMICon**, each node is provided with a daemon process that realizes the functionalities of the cooperative registry. First of all, the ability to store partial images is implemented. In the first part of the chapter we have seen that the manifest file of an image lists the IDs of all the layers that compose it. CoMICon modifies the structure of the manifest file by adding a flag called “present” that keeps track of the presence (or absence) of a given layer within the current registry. Secondly, registry slaves are provided with the capability of copying single layers between them. This functionality is essential in order to allow a dynaimc redistribution of the layers. When a layer is transferred, the destination registry sets the appropriate “present”

flag in the manifest of the image. Likewise, the possibility of deleting single layers is introduced, for it could be necessary to do so in order to either remove layers belonging to unused images or free additional storage memory. Whenever a layer is removed, its "present" flag is reset. Finally, a new operation called "distributed pull" is introduced.

Once the cooperative registry infrastructure is in place, the CoMICon system configures the distribution of the layers across the different nodes. In particular, the following input parameters are taken into account: the participating nodes, along with their network address and computational resource capacity; the Docker images that ought to be distributed; the microservice applications that should be deployed, i.e. a list of images along with resource requirements. Once provided with the input data, the system is able to make a series of decisions. First, it decides where to place the layers across the nodes of the cooperative registry in order to minimize the time needed for provisioning; redundant copies can be created for the purpose. Then, the nodes on which each container composing an application shall be deployed are selected. The decision is made such that the resource requirements are satisfied and the amount of data to be transferred over the network is minimized. As a result, CoMICon allows a Docker daemon to retrieve the layers composing an image from distinct sources.

Evaluations show that CoMICon is able to reduce the provisioning time by 28%. Indeed, by transferring layers from multiple registries in parallel, it is possible to take advantage of the network bandwidth in a more efficient way. However, the solution appears to be strictly designed for the deployment of microservice-based applications, in which the nodes that are going to run a set of services can be seamlessly interchanged. On the contrary, in a Fog computing architecture nodes are geographically distributed and the decision on which of them shall host a given service is usually imposed by the mobility pattern of the users. For this reason, a less pervasive approach to the problem of reducing pull time of containers is required.

2.5.3 Centralized storage for images and containers

Harter et al. [13] recognize that standard methods for container deployment are very slow. The problem resides mainly in the act of provisioning the file-system. Whereas the initialization of network, computing and memory resources is fast, the

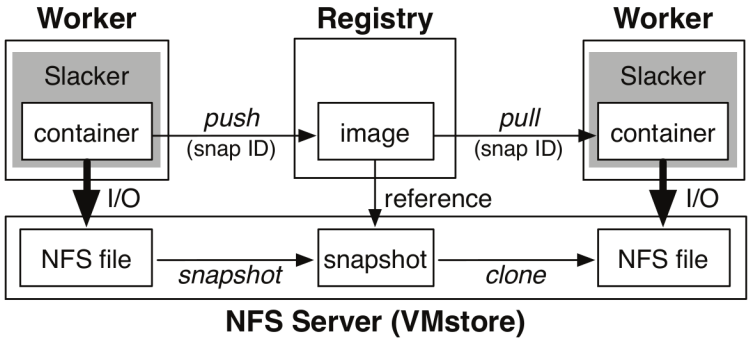


Figure 2.4: Architecture of Slacker

deployment of an entire file system (containing a Linux distribution, the application binaries and all of its dependencies) implies large download and installation overheads. File-system provisioning is a real performance bottleneck which causes undesired network congestion during the download phase and severe disk contention during the installation phase, when the content is decompressed and copied into the file system of the host.

In order to address the declared issue, they first perform an extensive analysis of the process of container start-up. Many different images of many different kinds have been taken into account with the goal of producing reliable results. Some interesting trends have emerged: first off, the analysis confirms that 76% of the time required to launch a Docker container is spent on pulling the image from the registry; moreover, a small portion of the data is actually used by the container once it executes: only 6.4% of it is truly needed.

The main contribution of the paper is the proposal of **Slacker**, a new storage driver for the Docker platform whose design is based on the result of the aforementioned analysis. Figure 2.4 shows the architecture of a Docker system which makes use of Slacker. The scheme is built around a centralized NFS storage which is shared by both the daemons and the registries of a Docker platform. The main idea of the work is to move the content of images and containers onto the shared NFS. Therefore, communications between daemons and registries will only involve the appropriate metadata, whereas the actual data is accessed remotely.

We provide a brief outline of the proposed system. On the network storage, the file system of a Docker container is represented by a single NFS file, whereas a Docker image consists of a snapshot of a certain file. Since the data never leaves the

shared storage, executing containers lazily retrieve the data from the appropriate NFS file through the network; in this way, I/O of useless content is avoided, and the container can start in a shorter time. Also, content is never compressed nor uncompressed, which helps save computational time as well. Another characteristic of Slacker addresses the issue of long pull operations. In the original Docker implementation, the whole data of the image itself flows over the network. With Slacker, only the snapshot IDs (which identify images) are exchanged, as the registries are effectively transformed into name servers. The custom storage driver will take care of instructing the NFS storage to make a snapshot or a clone during the push or pull operation respectively.

Results show that Slacker is able to speed up the deployment cycle of containers by a great extent. The time needed to pull a container is reduced by a factor of 72; this tremendous performance gain is due both to the fact that the data never leaves the shared storage and to the efficient implementation of the cloning operation that it provides. In addition, containers can start their execution immediately, as the lazy retrieval of data allows to avoid any preliminary transfer.

In spite of its good results, some drawbacks have to be considered. First of all, in spite of speeding up the starting time, the lazy approach obviously slows down container operations at run time. Moreover, the representation of container content as NFS files entails the flattening of the its layers; this is a considerable deviation from the standard philosophy of Docker, which promotes the layered organization of data in order to promote reusability and avoid redundancy. Last but not least, the work seems not suitable for an application in a Fog computing architecture. In a Fog computing architecture, which is composed of multiple geo-distributed nodes, the introduction of an additional component providing shared storage services would introduces new challenges regarding the choice of its location and possibly cause moderate but constant network congestion. All in all, the general disadvantages of the system, along with the ones specific to a potential application in a Fog computing environment, suggest that new directions for research be explored.

2.5.4 Docker daemon optimizations

In a recently published paper, Arif Ahmed and Guillaume Pierre [1] face the problem of slow Docker container deployment in the context of Fog computing environ-

ments. The article focuses on Docker daemons running on commodity machines such as Raspberry Pis, since they are likely to act as Fog nodes in next-generation infrastructures. Such low-end platforms are usually characterized by limited storage capacity, which can entail the need for frequent pulls, and low I/O performances, which can make the pull process remarkably slow. The usage of a standard Docker platform on Fog nodes can therefore induce large delays that are unacceptable for real-time applications. This paper aims to address such issue.

The authors perform a monitoring of different kinds of activities performed by the Docker daemon during the pull process in order look for the causes of such bad performances. The paper identifies three sources of inefficiency and proposes as many optimizations to address them.

1. The **parallel download of layers**, which is the default behavior of a Docker daemon, threatens to slow down the whole pull process whenever the network bandwidth is constrained. Since the download of the first layer takes longer to complete, the moment in which the extraction process can start is postponed as well. The paper proposes to replace the concurrent download of layers with a sequential procedure. Results show that the more the bandwidth is reduced, the more this optimization gets effective.

2. Docker layers are shipped over the network in the form of compressed archives and then locally extracted by the daemon. The process is carried out by means of a **single-threaded decompression function**. It turns out that this choice does not exploit at best the computational power of most machines, given that even low-end computers, such as Raspberry Pis, often offer a multi-core architecture. For this reason, Ahmed proposes the usage of an alternative multi-threaded implementation of the standard decompression function, which is able to increase the CPU utilization and therefore reduce the overall deployment time.

3. A third observed inefficiency is the **under-utilization of hardware resources**. Indeed, the standard deployment process begins with a network-intensive phase, during which CPU and disk are not used; then, it alternates between periods of high CPU usage (decompression of the archives) and high disk usage (copy of the extracted files). However, there is no reason to keep the sequential order of download, extraction and decompression. For this rea-

son, the paper proposes to arrange the three activities in separate threads, each of which pumps the resulting data to the next through pipelining. The simultaneous execution of the operations better exploits hardware resources of the host machine, thus significantly reducing their total duration.

Experimental results confirm the validity of the approach and of the proposed solution. The suggested optimizations are able to achieve a significant speedup of the deployment process, whose time is reduced by 60% to 70%, depending on the image that is taken into account.

All things considered, the overall objective of Ahmed's work is substantially similar to the one we aspire to; also, both works share the same constraints deriving from the adoption of the Fog computing paradigm. Our purpose is to continue the work on the same direction. In this thesis we try to find new ways to optimize the deployment of containers in order to deliver a tool which is able to meet the requirements of novel applications running on a Fog computing infrastructure.

Chapter 3

System design

In this chapter we are going to describe the structure and functioning of the solution implemented over the course of this thesis. The work is set in the context of Fog computing, a novel paradigm that brings storage and computing resources close to the user in order to face the growth in the number of connected devices and to meet the requirements of latency-sensitive applications. In a Fog architecture, Docker seems to be a valuable tool to gain all the advantages of deploying virtualized services without compromising the performance of the system. However, the considerable size of Docker images threatens to slow down the deployment of containers. This is a clear obstacle for a widespread adoption of the technology in Fog computing, given that fast deployment of services across different nodes is essential to follow the user along his movements. Moreover, the issue of slow deployment is particularly relevant in this context, where the usage of resource-constrained commodity machines and slow public Internet connections is likely to exacerbate the problem.

The objective of this work is to devise a system that is able to substantially reduce the time spent during the deployment of a container. As said in Section 2.5.3, researchers have shown that in spite of the large size of Docker images, only a fraction of the data (6.4% in average, to be precise) turns out to be actually useful. The key idea of this work is to improve the performances of the pull by deploying those essential files first, letting the container start its execution and the proceed with the download of the rest at run time. We have seen that Docker images are not treated as unitary objects; on the contrary, their content is distributed across different layers. For this reason, it seems reasonable to put the essential files into

an additional layer of the image, which will be called "base layer". This is the component that can be downloaded in advance with respect to the rest.

In order to realize a concrete implementation on the basis of our idea, the following research questions must be addressed:

- how to identify the essential files needed for the execution of a container?
- what to put inside the new base layer?
- how to change the Docker source code so that its behavior mirrors our expectations?
- what happens if the container tries to access a file that is not present because its download has not been completed yet?

The following sections of the chapter organize the exposition of the system design by addressing all of the abovementioned matters.

3.1 Profiling the container execution

It has been shown that Docker containers make use of just a limited part of their file-system content during the initial phase of execution. Harter et al. [13], for example, have carried out an extensive study of the matter. They consider a highly-representative subset of Docker images by choosing 57 samples from the public Docker Hub repository; Linux distributions, databases, programming languages, web servers and frameworks are just some of the involved categories. For each image, a block-level tracing tool is employed in order to measure the exact number of bytes read by the containerized process from the file system. Results indicate that, on average, only 6.4% of the data is accessed. In particular, the highest absolute waste in terms of unused files occurs when dealing with languages and web frameworks; the highest relative waste, instead, is within Linux distribution containers.

The first challenge that has to be addressed during the design of our solution is understanding which files are effectively accessed by a container. There are multiple ways to produce a reliable list of the files that are accessed by a process during execution. The following subsections are going to discuss the approaches that have been

attempted until the one that has been finally chosen; furthermore, some additional possibilities are mentioned as well.

3.1.1 Unix file metadata

In Unix-style file systems, all the metadata related to a file is stored into a data structure called `i-node`. Among the information held by an `i-node`, we can find the following attributes: the size of the file on disk; the pointers to the blocks of memory that host the content of the file itself; the owner and the group to which the current file belongs; the access permissions granted to the owner of the file, to users belonging to the same group and to the rest; a set of timestamps. Unix and Unix-like operating systems offer the possibility to inspect a file's metadata through the command `stat`. Listing 4 shows the output of a sample invocation.

```
    File: file.txt
    Size: 11      Blocks: 0
IO Block: 65536  regular file
  Device: 3ah/58d        Inode: 2977001609  Links: 1
  Access: (0644/-rw-r--r--)
     Uid: (661002/lcivolan)   Gid: (29073/ myriads)
  Access: 2019-02-15 11:56:16.336979000 +0100
  Modify: 2019-02-15 11:56:55.417847000 +0100
  Change: 2019-02-15 11:56:55.417847000 +0100
   Birth: -
```

Listing 4: Sample output of a `stat` invocation

As it can be noticed from the output of the command, there are three different timestamps that the operating system keeps track of for each file:

- the last-change time (`ctime`), which tells when the metadata of the file has last been altered;
- the last-modification time (`mtime`), that records the instant of the last change in the content of the file;

- the last-access time (`atime`), showing the last time in which some data belonging to the file has been read by a process.

The first method that we tried in order to detect the files accessed by a container involves the monitoring of the **last-access time**. The idea itself is straightforward. First, we launch the container's execution while keeping track of the moment in which the appropriate command is issued. Then, once the initialization has terminated, we can simulate an ordinary execution. What we do at this point depends, of course, on the kind of application that has been containerized. For example, in the case of a web server, we would probably request HTML pages and multimedia content; in the case of a database, instead, it would be natural to send SQL requests of various kinds. The goal of such simulation is to induce a usage of the resources that mirrors as much as possible a standard pattern in a real-world scenario. Finally, when the simulation phase is over, we can scan the whole file system of the container and look for those files whose last-access time has freshly changed.

Unfortunately, this approach, albeit simple and reasonable, does not work. According to its definition, the access time of an element must be updated every time a process performs just any operation on it: every read operation would therefore entail an additional write. As a consequence, this feature is prone to generate a considerable impact on the performances of a system, especially when dealing with large I/O loads and slow traditional hard disks. In order to reduce the overhead, modern Unix and Unix-like operating systems optimize the usage of the `atime` feature by limiting its updates or disabling them completely. For this reason, the mentioned timestamp cannot be relied on in order to identify the files that are handled by a container, and a new approach must be tried.

3.1.2 File-access notification API

Recent versions of Linux offer the **fanotify API** as a new functionality which implements file-access monitoring techniques. The interface provides a system of interception and notification for events that occur on the file system. Supported events include the opening of a file, a read operation, a write operation and a close. Such functionality is commonly used during the design of anti-virus software, where a system-wide scan of file-access operations is usually performed in order to identify

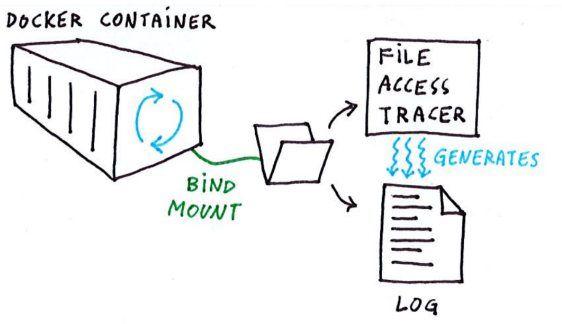


Figure 3.1: Identification of the essential files

suspicious patterns of the activity of a process. The API could also satisfy our purposes, as it offers a way to detect the objects that a Docker container makes use of during its execution.

The kernel delivers file-access monitoring through a set of system calls. First of all, the `fanotify_init` shall be used to create a new notification group. The notification group is an object that lives in the kernel and holds the list of the objects (i.e. files, directories and mount points) that have to be monitored. A second system call, `fanotify_mark`, can therefore be used to add new objects to the group while, for each of them, specifying the events that should be reported. Once the notification group has been created and initialized, processes that are interested in receiving and handling notification events can perform a read on the file descriptor that the initial `fanotify_init` returns. It is a blocking call, meaning that the execution of the listener process will stop until an event is fired. When this happens, the read unlocks and a series of one or more data structures of type `fanotify_event_metadata` flows into the buffer. Each of the received structures represent an event that has occurred on the file system and contains multiple descriptive fields. In particular, among the included information we can find the ID of the process that caused the event and the file descriptor for the object being accessed.

As a second approach to monitor the activity of a containerized process, we made use of the fanotify API. Yet, instead of building a program and use the system calls directly, we tried to exploit a tool called **fatrace** (file-access trace). Fatrace [18] is an open-source program which is freely available on the Internet. By using the fanotify API and the data in the `/proc` folder, fatrace is able to produce a log of events that report which files are being accessed by any running process.

The monitoring of the container through the host file system has proven to be unsuccessful; after all, the documentation of the tool warns that events on virtual and kernel file systems will be ingnored. For this reason, we have decided to perform the profiling of the container within the container itself. In order to be able to launch fatrace inside the container, the executable must be included into its file system. One possibility is to include the binaries in the image by means of a new Dockerfile. Another possibility is to prepare an appropriate folder on the host and make its content accessible from within the isolated environment by means of a **bind mound**; the goal is to provide the isolated process with the access to the access to the fatrace executable. Figure 3.1 reports a scheme of the profiling mechanism. Listings 5 and 6 show respectively how we made use of it to start the container with the execution of fatrace and launch the application that ought to be profiled.

```
$ docker run --name httpd-profiler
  -v "$PWD/profiling":"/profiling"
  --cap-add SYS_ADMIN -it httpd /profiling/fatrace -c -t
```

Listing 5: Initialization of the profiling environment for Apache httpd

```
$ docker exec httpd-profiler httpd-foreground
```

Listing 6: Launch of the Apache httpd web server

During the simulation phase we train the container by stimulating an activity that mirrors as much as possible a typical use-case. As we said in the previous section, the kind of operations performed in this stage depend on the application under scrutiny; therefore, there is a large degree of freedom that can be explored. Once the simulation is over, we stop the execution and we retrieve the log produced by the tracing mechanism. Listing 7 provides a sample of it.

Each line of the log corresponds to an event occurred on the file system. In order, from left to right, we can find:

- a timestamp indicating when the event was intercepted;
- name and ID of the process that caused the event;

```
17:03:26.646211 runc:[2:INIT](10): O /etc/passwd
17:03:26.646211 runc:[2:INIT](10): O /etc/group
17:03:26.646466 runc:[2:INIT](10): RC /etc/passwd
17:03:26.646466 runc:[2:INIT](10): RC /etc/group
17:03:26.646466 runc:[2:INIT](10): RO /usr/local/bin/docker-ent...
17:03:26.646466 runc:[2:INIT](10): RO /bin/dash
17:03:26.646466 runc:[2:INIT](10): RO /lib/x86_64-linux-gnu/ld-...
17:03:26.647142 docker-entrypoi(10): C /usr/local/bin/docker-en...
17:03:26.647190 docker-entrypoi(10): O /etc/ld.so.cache
```

Listing 7: Extract from a log produced by fatrace

- type of the event (`O` for open, `R` for read, `W` for write, `C` for close or a combination of them);
- path of the affected file.

The last step is the filtering of the log data. As we said, we are interested in finding the name of those file that are accessed, regardless of the time, the process nor the type of the operation. The shell command in Listing 8 shows how our requirements can be translated into code.

```
$ cat output.log | cut -d ' ' -f 4 | sort | uniq
```

Listing 8: Processing the generated log data

The result proves that the approach is suitable for our purposes.

3.1.3 Further possibilities

Even though for our system we adopted the technique described in the previous section, other possibilities could be explored. One of them involves the usage of **strace** [8], a debugging utility for Unix-like operating systems that is used to trace the execution of a process. The tool intercepts and reports the system calls performed by a process and the signals received by it. Strace can be of great utility

for developers and system administrators since it allows effective bug isolation when the source code of an application is not available. In our case, this program could be exploited to detect the usage of the system calls that testify an access to a file.

3.2 Building the base layer

Thanks to the preliminary phase of profiling, we can retrieve the list of files that are accessed by a containerized application. The objective is to make use of this information to create a minimal Docker layer that supports the execution of the application, at the very least when it does not deviate from what has been performed during the simulation.

The first necessary step is to get a copy of the whole file system of the original container. The command `docker export` fits our case, as it exports the container's file system to an archive. First off, we launch a sample container from the original image; then, we export its file system into an archive by passing its name to the mentioned command; finally, we extract the archive into an empty folder. Listing 9 shows the sequence of commands that has been issued to extract the file system of an Apache httpd Docker container.

```
$ docker run --name httpd-sample-run httpd
$ docker export httpd-sample-run > base/fs.tar
$ sudo tar -x -f base/fs.tar -C base/fs
```

Listing 9: Extracting the file system tree of an Apache httpd web server

The extraction of the archive generated by `docker export` is a crucial transaction, since it provides the source for the selective copy that is going to be performed based on the result of the profiling activity. It is therefore necessary to protect the overall integrity of the files, not just regarding the content, but also their metadata. In particular, information about ownership, user group and access permissions has to be preserved, since any alteration that modifies the execution environment of the application can potentially prevent it from executing normally. For this reason, the extraction command is issued with superuser privileges.

Once the container file system has been retrieved and appropriately extracted, we

are ready to isolate the content that will belong to the new base layer. By taking advantage of the information deriving from the profiling activity, a part of the content is copied into an initially empty directory. The latter will represent, once the operation is over, a snapshot of the minimal file system that is needed for the container to function. It turns out that the set of elements that have to be copied into the base layer includes more than simply the files listed by the output of a file-access tracing tool. Indeed, it is necessary to include not just the elements needed to support the execution of the container, but also the content whose utility will be clear only further ahead in the thesis. The following subsections investigate into the details of what has to be considered.

3.2.1 Files

For sure, the files that have proven to be necessary for execution must be present in the base layer. As it has been shown in the previous section, the list of files is the result of an appropriate processing of the log produced by `fatrace`. Listing 10 reports the core of the shell script that has been used to copy a list of files (`$LST`) from a source folder (`$SRC`), representing the container's original file system, to a destination one (`$DST`), representing the minimal version of it.

```
cat $LST | while read FILE
do
    sudo rsync --archive --relative "$SRC/./$FILE" "$DST"
    echo "cloned $FILE"
done
```

Listing 10: Selective clone of files

The operation of copying files, albeit simple, hides some caveats that the script must take into account in order to ensure the correctness of the result. First of all, the paths indicated by the tracing tool are relative to the internal perspective of the container, whereas the copy operation, which happens outside of it, needs to be instructed with paths valid from the host's perspective. For this reason, the parameter of the element to be copied at each iteration of the loop is prefixed with the path of the source folder on the host. Secondly, directory branch must be preserved along with each file. In other words, all the parent directories of a file

have to be cloned so that the full paths can be preserved. By using the `--relative` parameter and inserting a current-directory symbol appropriately into the source path we are able to achieve the desired result. Last but not least, metadata must be preserved at this stage as well; the `--archive` option carries out the task.

Once the minimal file system has been prepared, we test the result to check if the containerized application is able to work. To do so, we prepare a specific Docker image consisting of the base layer only and we run a container from it.

Unfortunately, the results of our test prove that copying just the files is not enough for the thin container to work. In most operating systems, and Linux is no exception, files can be accessed either directly or through the usage of links. However, the fanotify API, and therefore the tracing tool that makes use of it, is designed to report just the path of the real element that is being accessed, ignoring the possible links that led the process to it. Indeed, whenever the monitored application opens a file through a link, only the file itself is reported by fatrace and therefore copied into the base layer. As a result, the application inside the thin version of its container is unable to find what it needs to run. It is therefore clear that taking links into account is of vital importance for the success of our work.

3.2.2 Links

It has been pointed out that links must be considered when building the base layer of an original image. The tracing tools based on the fanotify API do not report the name of the used link in case a file is not accessed directly; therefore, to be on the safe side, all the possible paths must be taken into account. More precisely, for each file that is cloned into the minimal file system, it is necessary find all the links that recursively point to it and copy them as well. Listing 11 shows an extract from the shell script that performs the search for links. The relationship between a link and its target cannot be read backwards, because no information of it is recorded in the corresponding `i-node`; for this reason, a brute-force approach is the only way to proceed. The script takes as input a list of files and generates as output the list of links pointing to them.

Seeking links in the file system of the container presents some challenges. When the file system of a container is exported into a directory of the host, some of the links

```
cat $FILES | while read FILE
do
    find / -type l | while read LINK
    do
        TARG=$(readlink -f $LINK)
        if test "$TARG" = "$FILE"
        then
            echo "$LINK"
        fi
    done
done
```

Listing 11: Search for links to a list of files

can break down. In particular, the symbolic links whose target is specified with a relative path keep working on the file system of the host; on the contrary, those which contain an absolute path are interpreted according to the new perspective and therefore cannot be detected. A way to deal with the problem of broken links is to perform the search from the perspective of the container rather than the one of the host. To do so, the command `chroot` can be used. On Unix operating systems, a `chroot` operation consists in changing the root directory of the current process. By setting the directory containing the extracted file system as root, we re-enter the perspective of the container: the correct interpretation of links is restored and we are able to detect the desired ones correctly.

To verify the validity of our approach to a greater extent, we have performed tests with some of the most popular images from the Docker Hub: Apache httpd, redis, MySQL and a Go executable. For each of them, an initial profiling has been executed; then, we used the resulting information to fill the base layer with the appropriate files and links; finally, we prepared a one-layer testing image containing only the minimal data and tried to run a container from it. The result is encouraging: the thin Docker container is finally able to run.

3.2.3 Directory structure

By copying the accessed files and all the links recursively pointing to them we are able to build a layer that contains nothing less and nothing more than what is needed to support the execution of the container. However, it turns out that another element must be included in the base layer: the complete directory structure of the original file system of the container. At this point of the chapter it is hard to explain why this further passage is needed because the motivation will emerge more ahead in the work. In Section 3.4.4, the reader will find the answer to this legitimate question.

The copy of the whole directory tree into the base layer can be realized by making use of the `rsync` command. Listing 12 shows the instruction that achieves the desired result. With the `--archive` option we are able to preserve the metadata of the folders during the copy process. The `--include` and `--exclude` parameters let us respectively include all the folders and exclude all the files from the procedure. The two remaining pieces of information passed to the command correspond to the source directory, containing the overall exported file system of the container, and the destination one, where the content of the base layer is stored.

```
rsync --archive --include "*/" --exclude "*" base/fs/ lean/fs
```

Listing 12: Shell command used to clone the directory structure into the base layer

Once the command has completed its execution, our base layer has reached its definitive form. In particular, it contains the whole directory tree of the container's file system which, however, is populated only with the essential files as well as the links targeting them. The next step will consider the problem of inserting this additional component into the original image.

3.3 Preparing the custom version of the image

In the previous sections, thanks to the information collected during the simulation phase, we have been able to prepare the content that composes the base layer; we have also verified that the reduced set of files is able to support the execution of

containers. Here we face the challenge of injecting this additional layer into the original image in order to create a new version which is tailored to our needs. The overall objective of the work, indeed, is to retrieve such base layer first and then, only at run time, proceed by downloading the rest. In this regard, it is worth noting that the layered approach advocated by Docker comes to our help, as it lets us easily define a component which is clearly separated from the rest; such task would have been more complex had the file system been treated as a single unit rather than split over multiple building-blocks.

There are two reasonable possibilities according to which an image can incorporate an additional layer: below the existing tiers or on top of them. The goal of the following subsections is to investigate both options, providing the arguments for and against either and explaining the rationale behind our final choice.

3.3.1 Base layer on the bottom

The first possibility that we discuss regarding how to include the base layer into an original image consists of introducing it below everything else.

As a matter of fact, such idea is probably the first that comes to one's mind because it seems to be intuitive in several respects. First of all, the fundamental elements of any system are usually regarded as a solid platform on top of which the rest of the structure can be built. Our case fits into this view, since the base layer contains the files that are crucial for the container execution. Secondly, Docker images themselves are structured in a way that mirrors the aforementioned view. Indeed, by studying the content of the layers, it can be noticed that the foundational part of an image is often held by its lower parts: in the large majority of cases, indeed, the first layers are the ones hosting the file system of a Linux distribution. The reason for this distribution of the content lies in the fact that Docker images can be defined on top of an existing one, thus avoiding to start from scratch every time and allowing the common components to be shared among different instances. Last but not least, it can be easily noticed that the Docker daemon launches the download and extraction of the layers in sequential order, from the lowest one to the top. By introducing the base layer as a lowest tier, our component would naturally become the first to be retrieved. As a consequence, we could reasonably expect our present choice to facilitate the necessary modification of the Docker source code in the subsequent

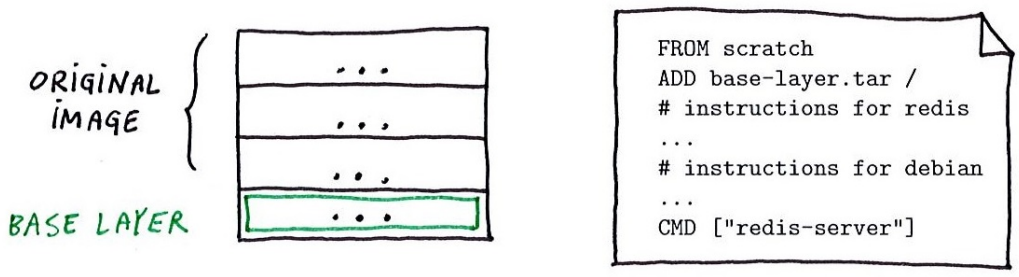


Figure 3.2: Scheme of custom image with base layer on the bottom

part of the work.

However, the approach presents serious drawbacks that threaten to question its actual feasibility. First off, the choice deviates considerably from the Docker philosophy, which regards layers as components that are stackable on top of each other and does not establish other ways to build the pile. As a consequence of not following the conventions, preparing a Dockerfile for our custom version of an image becomes a hard task. As we have seen in Section 2.2.2, every valid Dockerfile must begin with a `FROM` statement that specifies the base image on top of which the new one is defined, making it impossible to insert any other instruction prior to that one. Therefore, the only way to insert a new layer on the bottom is to prepare a Dockerfile that rebuilds the image from scratch. In other words, it is necessary to recursively open the chain of the parent Dockerfiles of the original image and copy their content in order into the new one, just after the first instruction that establishes the base layer. It turns out that this technique is not only inconvenient, but also fragile. Indeed, a complete local re-execution of the build process is involved, even though it has been already performed by the provider of the original image. Copying the content of the parent Dockerfiles into the new one implies that every command must be re-run, which is a potential source of problems since the context in which the process is carried out is by no means guaranteed to be appropriate.

To sum up, the simplicity of this first approach is only apparent. Its severe drawbacks make the method infeasible and can constitute a big obstacle for a potential adoption of our system. It is therefore clear that a different approach should be considered.

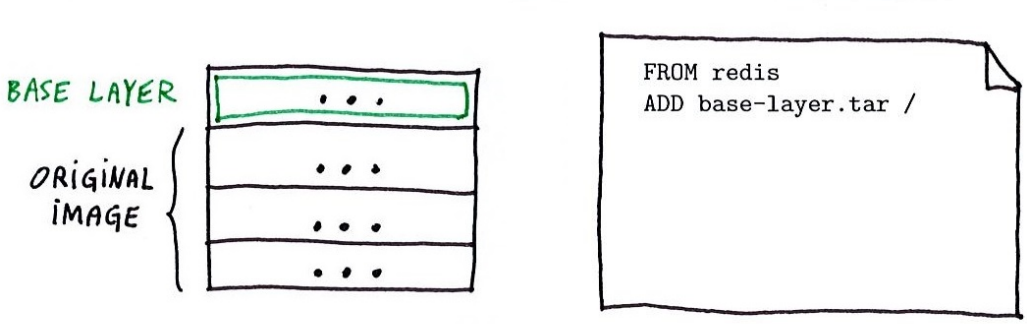


Figure 3.3: Scheme of custom image with base layer on top

3.3.2 Base layer on top

The alternative possibility for the inclusion of the base layer is antithetical to the first one: we insert it on top, rather than on the bottom, of the original image.

The approach appears counter-intuitive for a series of reasons. First of all, from the point of view of common sense, it is curious to position a component which is regarded as fundamental on top, rather than below, the regular ones. Most importantly, a conflict arises with the philosophy of Docker: common practice suggests that basic content should be put in the lower tiers of an image; in this case, the vital elements are positioned on the opposite extremity of the pile. Last but not least, the design decision of placing the base layer on top seems harder to manage when it comes to modifying the source code of the daemon. Indeed, the order in which the layers have to be downloaded is overturned: the last should be retrieved as first.

Nonetheless, a simple analysis of the the effects of the new approach can highlight a major benefit: Dockerfiles become trivial to prepare. The procedure, which consists of taking an original image and adding the new layer on top, boils down to the only instructions `FROM` and `ADD`. The right side of Figure 3.3 shows the Dockerfile needed

Unlike before, there is no need to recursively investigate the parent images and copy the lines of their Dockerfiles. Also, since the image is not built from scratch, it is not necessary to specify again the procedure for the container start-up (with a `CMD` instruction): the information will be automatically inherited from the parent's settings. Most importantly, we do not have to locally re-build the image from scratch and deal with all the problems that this can cause, because a ready package can be pulled directly from a public repository.

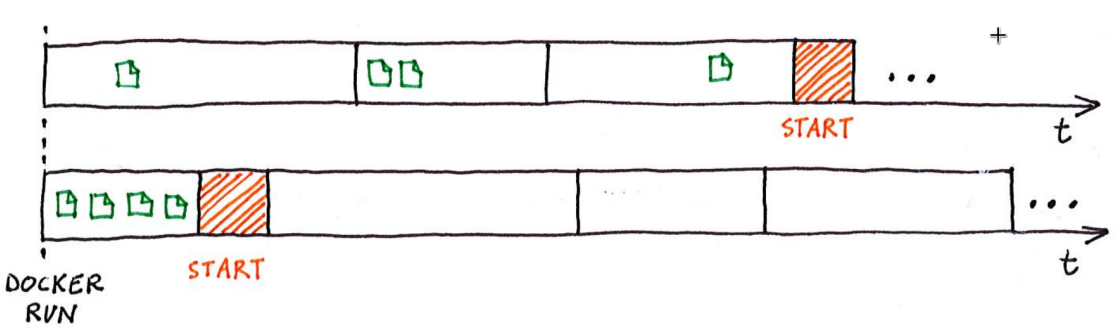


Figure 3.4: Schema of the behavior of the new `pull` operation

All in all, piling the base layer on top of the image seems the most reasonable choice since its advantages outweighs the disadvantages. As a conclusive remark, we can interpret this trade-off as an interesting example of how following the convention of a system can facilitate the work to a large extent, whereas breaking its rules, albeit "unwritten" ones, can cause unexpected complexities to arise.

3.4 The new `pull` operation

In this part we proceed to what can be considered as the core of our system: changing the behaviour of the Docker pull operation. In the previous section we have shown a way to introduce the new base layer into an original image; now it is necessary to introduce some changes in the source code of the daemon so that the procedure will mirror our policy.

Figure 3.4 shows the differences that our system aims to introduce. As already said, the content of Docker images is not transferred as a whole over the network; on the contrary, its content is distributed across multiple layers. The upper part of Figure 3.4 reports the behavior of the standard implementation of the procedure: the daemon retrieves all the layers, and only when the procedure is over it finally lets the execution begin. In the lower part of Figure 3.4 we can find, instead, the new implementation that we are going to propose: after downloading the base layer, which has been appropriately stacked on top of the image, the execution of the container starts immediately; afterwards, in the background, the download of the rest of the image can proceed. The base layer contains the minimal file system that is able to support the execution of the container; essential files are represented in

green in the picture.

Before delving into the details of the implementation, it is necessary to provide an outline of the internal structure of a Docker daemon so that the changes introduced in the code base can be fully comprehended by the reader. Without any doubt, the system is big and relatively complex. Hence, we are not going to provide an extensive description of the entire matter, but just of the parts that are somehow involved by our work. The simplest way to do so is by following the path of a "pull" request and studying the way it is handled by the daemon.

3.4.1 Standard procedure

The path of a pull request begins on the **client**, a Go struct that acts as a client for the Docker Engine API and enables interaction with the server. The code is contained in the package `client`. The library essentially provides 3 things:

- the type `Client`, a struct containing several fields regarding the status of the component;
- some functions to create and initialize new `Client` objects;
- an extensive number of methods for the `Client` type, each of which performs an operation against a docker server.

To establish a communication, a new Client object must be created and then its methods can be appropriately called according to the command that should be issued to the daemon. Listing 13 shows a simple usage of the Client type to pull an image on the Docker daemon; internally, the Client uses the RESTful API to communicate.

On the other endpoint of the communication channel, the Docker daemon receives the request and carries out the task. The HTTP request coming from the client passes through the `Server` first. The server is a sub-component of the daemon containing a long list of routes each of which encapsulates a mapping between an HTTP request and the method that should be called as a reaction. Listing 14 shows the message of the API that flows over the network, from the client to the server, as a consequence of a pull request.

```
// create the client object
cli, err := client.NewEnvClient()

// call a method on it
res, err := cli.ImagePull(context, "redis",
        types.ImagePullOptions{})

// show the outcome of the operation
print(res)
```

Listing 13: Sample usage of the Client type

```
POST /images/create?fromImage=redis&tag=latest
```

Listing 14: Request to pull the latest version of the redis image from a registry, according to the Docker Engine API v1.38

Every well-formed command is accepted by the server, parsed and then routed towards the appropriate handler of the central `Daemon` object. In the case just defined, the server calls the `PullImage` method of the `ImageService`, the internal component of the daemon collecting all the functionalities for image management. Listing 15 reports the signature of the method under discussion. Several arguments have been omitted for the sake of brevity; however, it is worth noticing the presence of the string parameters `image` and `tag`, which identify respectively the name and the version of the image that has to be downloaded.

```
func (i *ImageService) PullImage(..., image, tag string, ...)
        error
```

Listing 15: Partial signature of the function `PullImage`, called by the server to handle a request for image pull

The above-mentioned handler initiates a chain of function calls that gradually discriminate the parameters of the request and perform all the required operations. The initial set of involved functions has not been affected by any change in the course of this work, because it provides those basic functionalities that are needed

in our customized implementation as well. For this reason, we are not going into the details of all the steps, but we will rapidly explain some of the things that they achieve. Among many other things, this preliminary operations: set up a progress printing mechanism to provide the client (and therefore the user) with a feedback regarding the stage of completion of the activities; parse the name of the repository, splitting it up in the two parts of "username" and "repository", in case of a user-defined image, or just "repository", in case of an official one; locate the Docker Registry endpoint from which the image shall be pulled; retrieve and process the manifest file.

The **manifest file** is a JSON document that describes the structure of a Docker image. A Docker daemon downloads and parses it during the preliminary phase of a pull operation in order to know how to retrieve, install and configure the content of the image. Manifest files contain information regarding:

- the layers that compose the root file system of a container; in particular, their DiffIDs, which is the result of an hash computed on the content, is reported;
- the configuration that explains how to run the container, e.g. the path of the command to run, as specified by the `CMD` instruction in the Dockerfile.

Once the manifest has been downloaded and parsed into an appropriate Go object of type `Manifest`, the daemon checks if the image in question is already available locally. If so, the there is no need to pull anything, and the procedure can immediately return to the user; otherwise, it is necessary to proceed with the download.

The Docker daemon delegates the download of the layers of an image to a specific object called `LayerDownloadManager`. The responsibility of this component is to determine which of the layers actually need to be retrieved, and then download and register them on the local system. The layer download manager exposes a method called `Download` that realizes the above-mentioned functionality. `Download` is a **blocking function** that ensures that all the requested layers are present in the local layer store. Listing 16 reports a partial signature of it, where non-relevant parameters have been purposely omitted.

Among the parameters of the function, `layers` stands out for its importance: it is a slice (i.e. an extensible array in Go) containing a number of objects of type

```
func (ldm *LayerDownloadManager) Download(
        ...
        layers []DownloadDescriptor,
        progressOutput progress.Output
        ) (image.RootFS, func(), error)
```

Listing 16: Simplified signature of the `Download` method

`DownloadDescriptor`. A download descriptor is an entity referencing a layer that may (or may not) need to be downloaded. Among the several fields stored by it, we can find one that is meant to host the `DiffID` computed on the content of the layer; such field can be empty when unknown, for example in case the layer has never been downloaded before. Each download descriptor is also equipped with its own method that can be called to perform the download of the data from the remote repository.

The core part of `Download` consists of a loop that iterates over the descriptors of the layers that ought to be retrieved. For each of the layers, the function checks if a copy of it already exists in the local cache: if that is the case, the loop jumps to the following layer; otherwise, the download procedure is set up. It turns out that the layer download manager does not take care of the network transfers directly; on the contrary, it delegates such operations to an internal sub-component of type `TransferManager`, which is in charge of making scheduling and concurrency decisions in order to ensure an effective usage of the network resources.

The transfer manager is able to handle a general-purpose type of function, called `DoFunc`, that can encapsulate the logic on any transfer (be it download or upload) operation. Listing 17 shows its signature.

```
type DoFunc func(progressChan chan<- progress.Progress,
    start <-chan struct{}, inactive chan<- struct{}) Transfer
```

Listing 17: Signature of the `DoFunc` function type

The following objects must be passed to a `DoFunc` at the time of its invocation:

- an input channel, called `start`, that can be used from the outside to trigger the execution;

- an output channel, called `progressChan` through which information regarding the progress of the transfer can be submitted;
- another output channel, called `inactive` with which the function itself can notify to the outside world (i.e. to the manager) the fact that the job is no longer actively moving data over the network.

Since the transfer manager is only able to handle such a type of functions, every download descriptor that needs to be deployed is first packed inside a new `DoFunc` and then passed to the transfer manager itself. The `DoFunc` prepared by the layer download manager does the following things:

1. waits for the signal to start, which comes through the dedicated channel `start`;
2. downloads the tar archive containing the layer's content to a temporary file in the folder `/var/lib/docker/tmp`;
3. initializes an archive extractor around the downloaded tar;
4. extracts the content into the appropriate folder on the host machine, by making use of the storage driver pluggable component (see Section 2.4);
5. computes the hash over the content of the layer and stores its result into the appropriate `Layer` object, internally to the daemon.

The call of the blocking function `Download` on the layer download manager is a crucial step of the pulling procedure, because its conclusion signals that the content of the image has been correctly retrieved, extracted and installed in the local cache. Once all the `DoFunc` have terminated, and therefore all the layers have been installed in the local memory, `Download` can return. At this point, the daemon performs an **integrity check** to make sure that the content has been correctly deployed. In particular, every hash computed on the downloaded data is compared to its expected counterpart, as specified by the manifest file; if a mismatch is found, an error is thrown and the pull operation fails.

It is here, in fact, that the customized code of our system begins to appear. In the following part we are going to describe the changes that our system introduces

in the code of the Docker daemon. The content is distributed over several subsections, each of which tries to group together the changes that are meant to achieve a common, albeit partial, objective.

3.4.2 Topmost layer download only

The first objective that we tried to achieve has been to instruct the daemon so that only the upper layer of an image is retrieved, whereas the others are kept empty. Thanks to our reorganization of the image in the previous phases, the topmost tier corresponds to our base layer, and therefore contains the minimal file system that is able to support execution. By deploying that component only, we expect to reduce the amount of data that is transferred, extracted and installed, thus cutting the time that such operation takes.

First of all, we introduce a new global boolean flag `lastLayer` that indicates whether the layer under consideration is, or is not, the topmost one. The value of the new variable is set by the `Download` method of the layer download manager as the first step of the loop that iterates over the layer descriptors. The relevant lines of the code are shown in Listing 18.

```
for i, descriptor := range layers {
        lastLayer = (i == len(layers)-1)
        ...
}
```

Listing 18: Setting the flag indicating whether we are considering the last layer of the image

The `lastLayer` variable is set at the very beginning of each step of the iteration which happens, let us recall, on the layer download manager. Its value is actually read later on in the loop, and used by the manager to determine the desired behavior of each `DoFunc` built around every layer descriptor. When a `DoFunc` function, after receiving the start signal, comes to the critical point of downloading the archive containing the layer's content, it checks the value of the flag. If the value is `true`, it means that the last layer (i.e. the topmost one) is being considered; since, in our system, such layer represents the one containing the essential files for execution,

its download procedure should obviously start. On the contrary, a `false` value means that the `DoFunc` is dealing with an "ordinary" layer, whose download will be launched later on at run time; here it is necessary to avoid the transfer.

```
if lastLayer {
    // regular download of base layer
    downloadReader, size, err = descriptor.Download(
            d.Transfer.Context(), progressOutput)
} else {
    // no download of other layers
    tmpFile, _ := ioutil.TempFile("", "Empty*")
    downloadReader = ioutils.NewReadCloserWrapper(tmpFile,
            func() error {
        tmpFile.Close()
        err := opsys.RemoveAll(tmpFile.Name())
        if err != nil {
            logrus.Errorf("Failed to remove temp file")
        }
        return err
    })
    size = 0
    err = nil
}
```

Listing 19: Selective download of the base layer only; the code is inside the `DoFunc` used to retrieve the content of a layer

Listing 19 reports the code that implements the mentioned behavior. The result of the download of a layer is a `ReadCloser` object built around the retrieved archive. Such object offers two methods: one to read the data, one byte at a time; another to deallocate the internal resource, to be called once the reading is over. If the layer should not be retrieved, then we create our reader around an empty temporary file specifically created for the purpose. As a consequence, the rest of the operations of the `DoFunc` will be executes as if the layer were empty.

3.4.3 Disabling integrity control

Unfortunately, the plain deactivation of the retrieval of a layer's content is prone to generate errors at run time. As we have seen in Section ..., the daemon compares the computed content hashes against the expected ones before completing the pull. Apart from the base one, which is retrieved normally, the rest of the layers get their content replaced by empty archives. As a consequence, in those cases the computed hash will not correspond to the original one, and the resulting mismatch threatens to cause the pull operation to fail.

In order to solve the issue, we devised a method to disable such integrity check. The procedure is structured in two main phases: first, before the download of the image takes place, we retrieve the expected `DiffID` for each layer and we keep track of them; then, during the registration of each layer, we override the invalid hash computed on the empty archive and inject the expected one in its place.

Regarding the retrieval of the **expected hashes**, the operation is accomplished by the daemon with the code shown in the upper part of Listing 20. The daemon in its original version performs such operation after the download of the layers; in our case, it has to be anticipated before it, because the expected configuration has to be available during the download phase itself. The configuration is retrieved in the form of an object of type `RootFS`, which includes every layer's ID. Then, our system keeps track of the expected `DiffID`s by saving them inside the layer descriptor of the respective layer. Such choice is convenient because the layer descriptor is available inside the `DoFunc`, the function that takes care of the download and the registration.

The second phase of the procedure takes care of the actual **overwrite of the hash** result. To do so, we position ourselves inside the `Register`, a method called by a `DoFunc` to install the content on the local file system once the download is complete. Thanks to a newly-created global variable called `DesiredDiffID`, the `DoFunc` is able to set the hash that should result from the download of the layer. Then, from within `Register`, after the content has been installed by means of a call to the storage driver, we override the generated hash with the expected one. Listings 21 and 22 respectively report an extract of the code from the `DoFunc` and one from the `Register` function.

Thanks to the mechanism described in this section, the presence of empty layers

```
// retrieve the configuration
configJSON, configRootFS, _, err = receiveConfig(
        p.config.ImageStore, configChan, configErrChan)

// save the expected DiffIDs inside the layer descriptors
for i := range descriptors {
        descriptors[i].(*v2LayerDescriptor).diffID =
                configRootFS.DiffIDs[i]
}
```

Listing 20: Retrieval and tracking of the hashes of the layers; the operation takes place before the actual download

```
ddid, _ := descriptor.DiffID()
layer.DesiredDiffID = ddid
d.layer, err = d.layerStore.Register(layerData, parentLayer)
```

Listing 21: Setting the expected hash in the `DoFunc` before registration

does not generate errors anymore. It is possible to complete the pull when just the minimal part of the image, i.e. the base layer, has been retrieved. Once the image has been pulled, the user is able to issue the subsequent command to run a container from it. Figure 3.5 shows a simplified visual schema of what has just been explained. The fact that the base layer contains nothing less than what is needed allows the container to start and execute normally; the fact that it contains nothing more than that, allows the size of the minimal image to be as small as possible, and therefore reduces the time spent by a great extent.

3.4.4 Cloning the directory structure

Thanks to the modifications that we implemented in the previous steps, it is now possible to start a container after the deployment of the only base layer. Hence, the next major step would be to fill the remaining layers with the respective content at run time. At this point, however, a question regarding the feasibility of the operation arises. Section 2.4 has explained how a Docker layer essentially corresponds to a

```
var DesiredDiffID DiffID
func (ls *layerStore) Register(ts io.Reader, parent ChainID)
        (Layer, error) {
    ...
    if err = ls.applyTar(tx, ts, pid, layer); err != nil {
        return nil, err
    }
    // overriding generated diffID with the expected one
    if DesiredDiffID != "" {
        layer.diffID = DesiredDiffID
        DesiredDiffID = ""
    }
    ...
}
```

Listing 22: Overriding the generated `DiffID` with the expected one

folder on the file system of the host machine. The layers (i.e. the folders) are mounded together with a virtual file system, called Overlay FS, which then provides a unified view of the overall content to the container.

According to the specification, Overlay FS will not support run-time changes of the underlying layers. In particular, "changes to the underlying filesystems while part of a mounted overlay filesystem are not allowed. If the underlying filesystem is changed, the behavior of the overlay is undefined, though it will not result in a crash or deadlock" [22]. Our system, on the contrary, wants to add new content when the mount is already in place.

In spite of what the documentation says, after performing an extensive set of experiments on the Overlay FS we found a different answer. Our tests reveal that the virtual file system supports live changes of the layers under the following circumstances:

- files are just added into the layers (and not, for example, renamed, modified or deleted);
- the complete directory structure is present across all the layers;

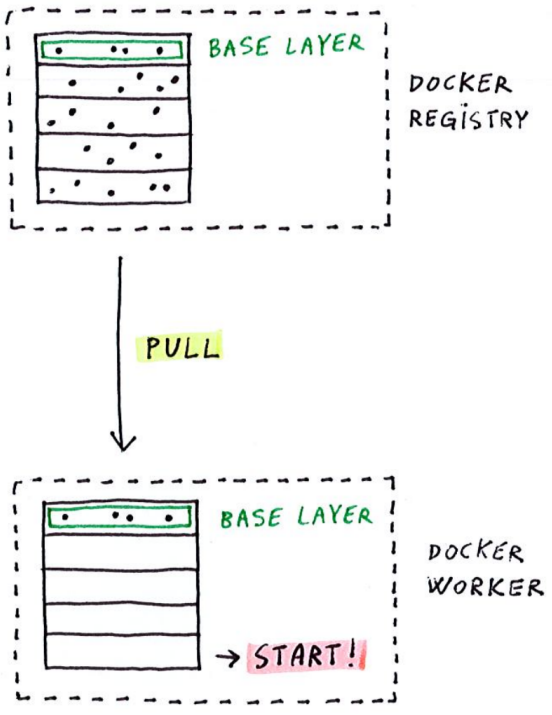


Figure 3.5: Schema of the customized behavior of an image pull

- we drop the `dentry` cache of the underlying file system.

To the best of our knowledge, this behavior of the Overlay FS is related to the way that directories with the same name are merged together across the different layers in order to provide a unified view of their content.

When building the base layer in Section 3.2 we could not explain yet the reason why the whole directory tree was needed inside of it. At this stage, it is clear: we need to copy it across all the empty layers before the virtual file system is mounted by the daemon. The operation is done in the final part of the `Download` function of the layer download manager. Such function would normally terminate once all the `DoFunc`s have finished the execution; in our case, however, we must first provide the skeleton of the file system across all the layers. Let us remember that the complete directory structure of the container has been included into the top-most base layer for this very purpose. Therefore, we are now able to retrieve and copy it into the empty layers below. Listing 23 reports the code that accomplishes what has just been described.

The initial lines of code contained by the loop serve to retrieve the path of the directory corresponding to the currently considered layer. The core of the operation,

```
topLayerMeta, _ := topDownload.layer.Metadata()
topLayerFolder := topLayerMeta["UpperDir"]
// for every delayed layer...
for i := 0; i < len(layers)-1; i++ {
    delLayerDesc := layers[i]
    delLayerDownTrans := downloadsByKey[delLayerDesc.Key()]
    delLayer := delLayerDownTrans.layer
    delLayerMeta, _ := delLayer.Metadata()
    delLayerFolder := delLayerMeta["UpperDir"]
    delLayerCtx := delLayerDownTrans.Context()
    // 1) clone the directory structure
    cmdStr := "rsync --archive --include '*/' --exclude '*' "
            + topLayerFolder + "/ " + delLayerFolder
    cmd := exec.Command("sh", "-c", cmdStr)
    err := cmd.Run()
    if err != nil {
        logrus.Errorf("error while cloning dir. str.")
    }
    ...
}
```

Listing 23: Cloning the directory structure into all the empty layers before the conclusion of `Download`

however, comes next: it consists of running a command line instruction that makes use of the `rsync` application. `rsync` [12] is a software available for Unix and Unix-like operating systems that provides an extensive set of functionalities dealing with the copy of files and directories, both between local and remote locations. The tool has been employed in our system by specifying the following parameters:

- `--archive` is used to preserve the metadata of the folders during the process (e.g. owner, group, permissions and so forth);
- `--include "*/"` tells the tool to consider all the directories;
- `--exclude "*"`, on the other hand, tells it to exclude all the files;

- the source of the transfer, i.e. the directory corresponding to the base layer; by appending a `/` we prevent the copy of the root component itself from happening;
- the destination of the transfer, i.e. the directory of each lower layer at each step of the iteration.

The effect of the execution of `rsync` with the mentioned configuration is a copy of the directory sub-tree from the source to the destination. When the task is over, all the directories contained in the base layer will be present also in the other previously empty tiers of the image.

3.4.5 Background download of the rest of the image

Once the base layer has been downloaded, extracted and installed, and its directory tree has been replicated across all the underlying empty layers, the pull operation can return and the execution of the container can begin. However, before returning, it is necessary to set-up the necessary software infrastructure to schedule the filling of the rest of the layers later on in the background. Since the directory tree is already present in all the layers, the newly-added content will be picked up and shown by the merged view; as a consequence, the content will gradually appear in the file system of the running container.

The Go programming language offers a feature called *goroutine*, a construct that provides a simple way to launch the execution of code in the background. Any function defined by the programmer can be run in a separate thread by inserting the keyword `go` right in front of the invocation. The actual implementation and management of goroutines depends on the Go runtime that operates behind the scenes and on the computational resources offered by the physical machine. However, this feature provides a convenient and effective tool that lets the programmer define the structure of concurrent applications in a straightforward way, which seems quite tailored to our needs.

In our system, we make use of goroutines to schedule the complete download of the image data in the background. The code is set in the same for loop that we described in the previous step of our solution: for each layer involved by the iteration,

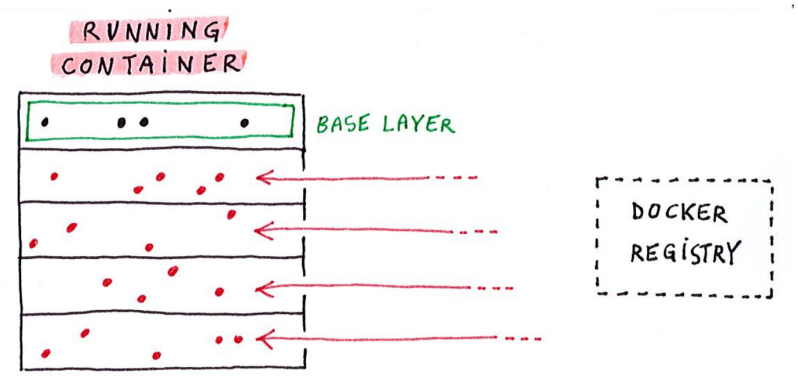


Figure 3.6: Filling image layers at run time

in addition to copying the directory tree, we also launch a goroutine with the aim of downloading, extracting and installing the data into the appropriate folder.

```
go func(delLayerDesc DownloadDescriptor, delLayerFolder string,
        delLayerCtx context.Context) {
    tarReader, _, _ := delLayerDesc.Download(delLayerCtx,
            progress.DiscardOutput())
    dataReader, _ := archive.DecompressStream(tarReader)
    var untar = chrootarchive.UntarUncompressed
    untar(dataReader, delLayerFolder, &archive.TarOptions{
        WhiteoutFormat: archive.OverlayWhiteoutFormat,
    })
    tarReader.Close()
}(delLayerDesc, delLayerFolder, delLayerCtx)
```

Listing 24: Launch of a goroutine that performs the deployment of a layer in the background

Listing 24 reports the code prepared to realize such behavior. By calling the `Download` function of the layer descriptor, a compressed archive containing the data is downloaded through the network, saved as a temporary file in the local machine and encapsulated into a reader that is then returned and memorized as `tarReader`. Then, ad additional reader performing decompression is built around it: it is the `dataReader`. At this point, the uncompressed archive is passed the function in charge to open it and save the content into the appropriate directory corresponding to the current layer. As a final step, the innermost reader is closed.

3.5 Handling early access of delayed files

Having reached this point, the main part of the work is complete. However, there is still a major issue that remains unsolved: we have to define what happens if a container tries to access a file which has not been downloaded yet. Even if, in many cases, a container can run smoothly even with the minimal version of its file system, the initial phase is a delicate one because the execution could deviate from the standard behavior and consequently try to access a file which is not part of the base layer. If this happens in the first phase of execution, before the appropriate background routine has carried out its retrieval, the application does not find the file and reacts accordingly (e.g. by throwing a "file not found" error). Yet, that should not be the case: the file is missing because of our system, and the system should take care to avoid the consequences.

Multiple factors can contribute to the occurrence of such a situation. First of all, if the profiling phase is not done accurately, some crucial files may be left out of the base layer. For example, this could happen when the simulated activities do not represent an exhaustive sample of the typical use cases. In addition, it is always possible that the containerized application executes unexpected operations. Assuming the application is providing services to the users, unusual requests which have not been simulated before may still arrive at some point. The latter observation reminds us the fact that, no matter how accurately we engineer the profiling phase, the application has always the right to deviate from its typical behavior and access unforeseen resources. Given that our system must behave correctly under any circumstances, it is necessary to handle those "unlucky" cases as well.

3.5.1 Main idea

Our approach to address the challenge is inspired from the work conducted by Gene Cooperman [19, 11]. The main objective of the works is to checkpoint and restart multithreaded processes in a way which is transparent with regard to the application. Kernel-level checkpointing is difficult to implement, because it requires modifications to the code of the kernel. Application-level checkpointing, on the other hand, has the drawback of forcing developers to alter the code of their programs. Therefore, Gene Cooperman et al. propose a for user-level checkpointing method that avoids

both disadvantages. The work is based on the idea of wrapping system calls in order to execute additional operations before the original procedure finally starts. In the works, the `clone` function of the NPTL (Native POSIX Thread Library) is wrapped. The objective is twofold: first, to set up the internal data structure in which the status of a thread can be saved; second, to provide the new thread with an additional signal handler which will be used to trigger the checkpointing procedure. Also, system calls like `open`, `fopen` and `close` are involved. In this case, the aim is to intercept and record information regarding the file descrpitors opened by a process. By wrapping system calls, the authors manage to alter the behavior of an application without introducing changes its source code, thus ensuring the transparency of the system.

In concrete terms, wrapping a set of system calls can be achieved in Linux by making use of the following features. The `LD_PRELOAD` environment variable can be used to specify a shared library that must be loaded before any other library. Thanks to it, we can instruct the system to consider our customized library first when looking for functions during the resolution process at run time. The preloaded library can redefine one or more system calls by declaring functions with identical name and signature. As a consequence, a process executing one of them will be hijacked and our code will be run instead. Even if there is no restriction on the code that the body of those functions can contain, at some point it will be necessary to call the original system call to resume the standard behavior. The `dlsym` system call, together with the parameter `RTLD_NEXT`, allows to retrieve the pointer of the next function along the resolution path, which, in our case, would be the original one.

Even though we are trying to achieve a different objective compared to Cooperman's work, the proposed approach seems anyway suitable for our case. Wrapping the system calls that provide file-access functionalities can let us perform preliminary operations before a file is actually opened. Our objective is to avoid the failure of `open` function and alike when the involved file is supposed to arrive with some delay; on the contrary, we want such system calls to wait for the download of the file and then proceed only once the transfer is complete. Like in the aforementioned works, we also want this new feature to work in a completely transparent way from the point of view of the containerized application. The wrapping technique ensures this advantage as well.

3.5.2 Wrapping file-access system calls

In order to intercept the system calls related to file-access, we prepare a new shared library called **libpreload**. The source code of the library, which is written in C, lies in the `preload.c` file. Inside of it, we can find the redefinition of the appropriate system calls, i.e. the functions `open` and `fopen`.

```
static int (*real_open)(const char *pathname, int flags) = NULL;
int open(const char *pathname, int flags)
{
    if (!exists(pathname)) {
        char fullpath[PATH_MAX];
        realpath(pathname, fullpath);

        if (delayed(fullpath)) {
            wait_avail(fullpath);
        }
    }

    if (!real_open) {
        real_open = dlsym(RTLD_NEXT, "open");
    }

    return real_open(pathname, flags);
}
```

Listing 25: Wrapper for the `open` function of the standard library

When the container tries to open a file with the `open` system call, the code reported in Listing 25 comes into play. First, it verifies whether the file exists. If it does, there is no need for further checks: the operation can proceed with its original behavior and open the resource correctly. If, on the contrary, the file does not exist, it is necessary to perform an additional check to verify if it is part of that part of the image which is downloaded asynchronously by our system. In such case, we do not want the `open` to fail with a "file not found" error because the missing resource will arrive with some delay. Hence, we call the `wait_avail` function which waits for the

deployment of the file. Once it is finally available, mentioned function returns and the real system call is given the task of resuming its normal operations.

The code of our wrapper functions make use of a number of additional calls that ought to be explained. As one would expect, `exists` verifies the existence of a given file. The function, provided by the library itself, carries out the task by trying to retrieve its metadata from the operating system through the `stat` system call: the success of the operation means that the file is accessible; a failure, on the other hand, proves that the file does not exist. The function `delayed`, instead, accepts as input an absolute path and returns `true` if the file belongs to the part of the Docker image which gets deployed at run-time by the system. Regarding the implementation of `delayed`, multiple approaches are possible: Section 3.5.3 goes into the detail of some of the potential choices. Moreover, the function `wait_avail` is used. Its goal is to cyclically wait for a fixed amount of time and check if the file has arrived; when this happens, the procedure returns. Last but not least, the wrapper makes use of `dlsym`, a system call provided by Linux that allows to get the address of a symbol in memory. The effect of passing `RTLD_NEXT` and `"open"` as parameters is that the next occurrence of the desired symbol is retrieved, according to the search order. As a result, we get a pointer to the original `open` system call, which can be used, as the last instruction of the wrapper, to forward the request to the real function and return the result to the client. As a conclusive remark, the reference to the original system call is stored into a global variable `real_open` which enables re-usability across consecutive calls coming from the same process.

Once the code of the new library is ready, the source file must be compiled into a shared object that can be dynamically linked by the OS. The steps for the compilation procedure are shown in Listing 26. First off, the source is compiled into an object file. The `-fpic` parameter accounts for "position independent code". Multiple programs can use our library, and each of them will load its instance into a different memory address. It is therefore vital that the code is guaranteed to work regardless of its location in memory. After the compilation of the source, a shared library is created from the object file. The parameter `-ldl` instructs the linker to link the *dynamic link* library, which is needed by ours as it makes use of the `dlsym` function. Finally, the name given to the result follows the Unix conventions according to which a shared library must be called with a name that begins with `lib` and terminates with the `.so` extension.

```
$ clang -c -fpic preload.c
$ clang -shared preload.o -o libpreload.so -ldl
```

Listing 26: Compilation of the wrapper library

The system makes use of a feature of the Linux operating system which provides the possibilities to preload a user-defined library before all the other libraries. The functionality can be used by setting the environment variable `LD_PRELOAD` with the path of the library that should be given priority during the loading phase. In our case, the new `libpreload.so` redefines some of the functions exposed by the standard library (`open` and friends) by declaring functions with the identical name and signature. As a consequence, the original system calls gets shadowed by our new implementation: this is how we are able to execute additional code when a system call is used by the application.

The adoption of the wrapping technique has a number of promising advantages. First of all, the execution does not fail when the file that is missing is a crucial one. On the contrary, the container execution will pause and then resume once the content is available. In addition, by introducing our changes at the system-call level, transparency is guaranteed. The application need not be altered in its source code and is not aware of what is going on behind the scenes either: from its perspective, the new `open` is just a system call that can take longer to return. Last but not least, by appropriately setting the `LD_PRELOAD` variable in the Dockerfile of the container, all the processes running inside the isolated environment can benefit from our wrapper. In this regard, it should be noted that, even if the common usage of Docker suggests the approach of placing only one application per container, it is anyways possible to run many of them in the same unit. The wrapping approach is still guaranteed to work.

3.5.3 Look-up of delayed files

The previous section has shown how a new version of the `open` system call is implemented in our library. However, by omitting the internals of the `delayed` function we have glossed over a fundamental part. The `delayed` function is a pluggable component of the library that allows to specify which procedure should be followed when

it comes to determining whether a given file will be downloaded asynchronously at run time. This last section is going to explain the main challenge of this operation and propose different approaches for its design.

The first thing to do, before the look-up itself, is to retrieve the list of files. It is an essential piece of information, because it indicates which files may be absent during the early phases of a container's execution. Such list consists of the path of the files that have been excluded from the base layer of the image (please refer to Section 3.2). To obtain it, we can compute the difference between the list of files (and links) of the whole original file system and the list of file (and links) included in the base layer. Assuming that those lists are contained in files respectively named `full.list` and `min.list`, Listing 27 shows how to generate the new list of delayed files.

```
$ cat full.list min.list | sort | uniq -u
```

Listing 27: Computing the delayed-files list as a difference

Unfortunately, the potential length of the list is likely to slow down the process of look-up. For example, such list for the httpd Docker image contains more than 6000 units, including files and links. Depending on how often a container attempts to access a non-existent file, the look-up function of the wrapper library may be used many times. For this reason, when choosing an implementation for the `delayed` plugin, it is necessary to think carefully about its performance implications.

A first solution that can be considered is to keep the list in a **text file**. In this case, the text file should be prepared offline and added to the base layer before building the customized version of the Docker image. Its location in the file system could be either fixed, and therefore hardcoded into the library, or flexible, in which case it could be provided to the wrapper by setting and retrieving the value of an appropriate environment variable. When the container tries to open a non-existent file, the wrapper in turn opens the delayed-file list and looks for the path at stake; if it is found, then the resource is supposed to arrive soon and can be awaited. In spite of its simplicity, the approach seems very inefficient, because it involves opening and reading from disk a considerably long file.

A more efficient solution is to use an **environment variable** to keep the list of delayed files directly in memory. The list of paths should be assembled in a single

string, using a certain character as separator between the different components. Such string should then be assigned to a variable by an `ENV` instruction in the Dockerfile. The variable set in such a manner is available from within the container and, as a consequence, can be accessed by the preloaded library as well. Listing 28 finally shows the internals of the function `delayed` in this case. First of all, the value of the environment variable (`DELAYF`, in this case) is retrieved. Then, the path in question is delimited by the previously-chosen special character, at both ends of the string. In this way, the look-up process can be implemented as searching a substring (our `elem`) inside a string (our `list`). The function `strstr` of the C standard library fits our case, returning a pointer to the first occurrence of the element or `NULL` if the sequence is nowhere to be found. As a last remark, it can be noticed from the code that in the event that the environment variable is not set, the `delayed` function handles the case by returning `false`.

```
static int delayed(const char *fullpath)
{
    char *list = getenv("DELAYF");
    char elem[PATH_MAX+2];
    if (list == NULL) {
        return 0;
    }

    sprintf(elem, ":%s:", fullpath);
    return strstr(list, elem) != NULL;
}
```

Listing 28: Code for the `delayed` function when reading delayed-file list from the environment

Storing the list of delayed files in an environment variable is without any doubt a more efficient solution than using a "special" file on disk. Indeed, the information is already in memory, and the library has just to retrieve its address. However, further optimizations are possible.

As a last approach to the task of efficient look-up, we show a more sophisticated method which involves the usage of a **bloom filter**. Bloom filters are data structures that can be used to test whether an element is part of a set. Their name derives from

Burton H. Bloom, who introduced the idea in 1970 [5]. A Bloom filter is composed by the following components:

- a vector of bits, of arbitrary length m, initialized to the value 0;
- a set of independent hash functions, that process an element and produce a value in the range $[0, m-1]$.

To insert an element in the set that is mapped onto the Bloom filter, all the hash functions are computed and each result is used to set the corresponding bit of the vector to 1. As multiple elements are added to the set, it is possible that the same bit is set to 1 more than once. Besides adding new elements, it is also possible to test the membership. To verify if an element belongs to the set, all the hash functions are applied to the element itself and the corresponding bits of the vector are checked: if at least one of them is 0, the element does not belong to the set; otherwise, if all of them are 1, the element is likely to be a member.

Bloom filters are very efficient in terms of memory usage, as they are able to record information regarding the membership of its elements in the limited space of a vector of bits. However, this efficiency comes at a price, since false positives are possible: a Bloom filter can respond to a membership test either with a "definitely no" or with a "maybe yes". The choice of the length of the bit vector represents a trade-off between memory-usage and accuracy: the smaller the vector, the higher the error, and vice versa.

In our case, Bloom filters can be used to avoid storing the whole list of delayed files in memory. Indeed, once they have been used to initialize the filter, the bit vector is the element that contains all the information useful to recognize the membership. Such bit vector can then be saved in an appropriate environment variable that provides access from within the container. The `delayed` function of the wrapper library can retrieve the configuration of the Bloom filter and use it to test if a file belongs to the set of those which will be downloaded with some delay. Listing 29 reports the code of the `delayed` plugin when it is configured to work with a Bloom filter. The operations are self-explanatory.

Even though the Bloom filter data structure has considerable advantages from the point of view of memory usage, the issue of false positives can produce serious

```
static bloom_t filter = NULL;

static int delayed(const char *fullpath)
{
    if (!filter) {
        char *bits = getenv("BLOOMF_BITS");
        filter = bloom_load(bits, strlen(bits));
    }

    return bloom_test(filter, fullpath);
}
```

Listing 29: Code for the `delayed` function when using a Bloom filter to perform the look-up

consequences in our scenario. If a non-existent file is improperly expected to be downloaded later on, the container would enter the wait loop and stay there forever, waiting for a resource that Docker will never deploy. As it will be discussed in the related work section, there are possible solution for this inconvenience. However, for the sake of simplicity, in our prototype we decided to adopt the approach of storing the list of delayed files into an environment variable accessible by the wrapper library.

Chapter 4

Evaluations

Once the system has been prepared, it has been necessary to perform an accurate evaluation in order to evaluate both the correctness of its behavior and the gain in terms of performance. This chapter opens with an outline of the methodologies used for the tests. Then, we report the results that have been obtained and we highlight their significance when put in relation with the objectives of this thesis. Finally, we conclude by identifying the advantages of our system, both in general and in a Fog computing context, and by providing a number of directions for possible future enhancements and optimizations.

4.1 Methodologies

As far as the performance evaluation of our system is concerned, a careful choice of the methods is crucial in order to obtain results that are meaningful for the context in which the work is set. For this reason, we have thoroughly considered the diverse possibilities regarding which Docker images to test, where to upload them and to which machines to execute the deployment. The following subsections go into the details of the choices that have finally been made.

4.1.1 Considered Docker images

When looking for the target images to be used during the testing phase, we mainly focused on the most popular images from the public Docker Hub registry. Our choice has been to consider the following images: **httpd**, which is the containerized version of the notorious Apache HTTP Web Server; **redis**, a popular key-value store that can be used, among other things, as a database. We made the deliberate choice of picking two different applications that provide distinct kinds of services. In this way, we are able to check the correctness of our operations and evaluate the results under different circumstances. We also chose them because they are good examples of applications that are relatively simple to use and to test. As a result, we can be confident about the reproducibility of both our experiments and their respective outcomes.

In addition to the official images, we also prepared a new one composed by an **sample application binary** (from now on called "glife") on top of a Linux distribution. For the purpose, we chose a Go implementation of John Conway's Game of Life. The fact that the program is written in the same programming language as Docker does not matter, because the source code is compiled to a binary that executes directly on top of the OS. On the other hand, the Game of Life was chosen because it has the merit of giving an immediate visual feedback to the user without requiring any input from him. In this way, we are able to underline the fact that the application is actually running even though the deployment of the image has not yet been completed. Furthermore, a general-purpose application binary with no constraints on its behavior lets us evaluate a case which is as general as possible.

4.1.2 Source for the deployment

As source for the deployment procedure, we employed the official Docker Hub, which is the default public registry for the Docker architecture. A reason for this choice is the simplicity of the approach. Being the default source of data for many users, the Docker Hub is already up and running and optimized for heavy workloads. Furthermore, since the public Docker Hub is the default registry for every daemon, we do not have to change any setting to personalize the behavior. A last reason is the relevance of the scheme: by using the Docker Hub, we are able to evaluate the

performance of our system in the most common use case.

On the public registry, we opened a dedicated account and set up a repository for each of the tested images. We then uploaded the images in their multiple versions, i.e. the standard one and the customized one (with the additional base layer).

4.1.3 Destination for the deployment

As for the machine used for the deployment, we have first considered the usage of a **personal computer** to evaluate the time saved during the fulfillment of a `docker pull` command issued by a user. For the purpose, we have used a machine equipped with the following facilities:

- CPU Intel® Core™ i7-6500U with 4 cores and 2.50GHz of speed;
- 12.00 GB of RAM;
- traditional HHD;
- wireless connection to high speed network.

In addition to the mentioned set-up, which belongs to a reasonably good machine, we have also performed our tests on a small **Raspberry Pi 3** computer. Raspberry Pis are single-board computers available at low prices on the market. Their extremely constrained resource availability gives us the opportunity to evaluate the benefits of our system in a situation where performance improvements are truly needed. Furthermore, given their low cost and small dimensions, Raspberry Pis are likely to become the implementation choice for the next generation nodes of a Fog computing architecture. Since the Fog is the context of this work, it is absolutely relevant to test the deployment of Docker containers on such kind of low-end machines.

The technical characteristics of the used Raspberry Pi are reported hereafter:

- CPU Quad Core 1.2GHz Broadcom 64bit;
- 1.00 GB of RAM;
- Micro SD card for operating system and storage;

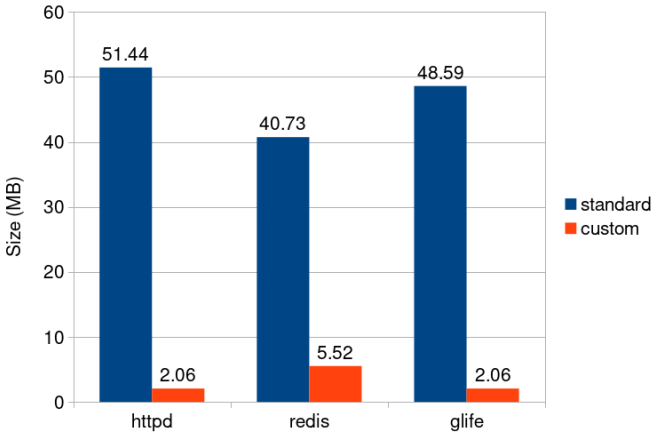


Figure 4.1: Comparison between the amount of data transferred during a standard and a customized deployment

- Ethernet connection to high speed network.

4.2 Results

First of all, we have measured the reduction of the data that must be **transferred over the network** in order to start a container with our system. As it has been explained, Docker transfers each layer's data from the registry to a daemon in the form of a compressed tar archives. According to the standard system, all the layers are deployed before container start-up; in our case, instead, the base layer is enough, since it contains a minimal version of the file system which should be enough to support the first phases of execution. Figure 4.1 compares the volume of data that is transferred by standard Docker against the data transferred by our system.

The reduction in terms of download size is evident. As far as redis is concerned, the transferred data is only the 14% of the original size; in the case of httpd and glife, the ratio is around 4%. This figures do not depend on the specific hardware configuration that hosts the Docker daemon, because they show an intrinsic characteristic of the transfer itself. For this reason, the specifications of the machine on which the test has been carried out do not affect the results.

In addition to measuring the amount of data transferred over the network, we also evaluated the final **size of the image on disk** once the deployment is complete. As shown in Section 3.3, the customized version of a Docker image contains an

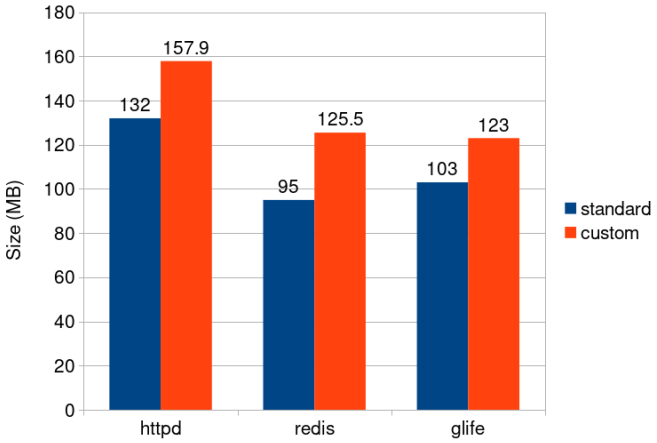


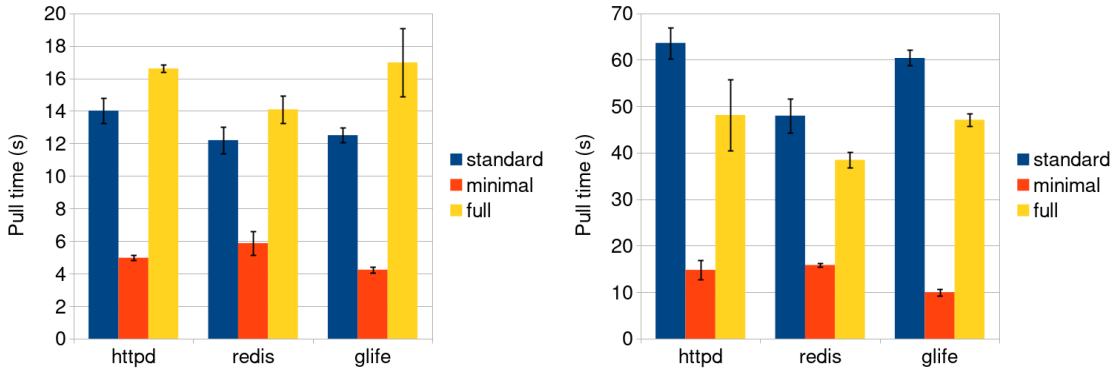
Figure 4.2: Comparison between the size of a standard Docker image and that of our customized version containing the additional base layer

additional layer. Furthermore, a copy of the complete directory tree is present in all the layers, in order to support the run-time deployment of the image content. We expect these two factors to cause an increase of the overall size of a customized image compared to its original version. Figure 4.2 shows the results of the analysis performed on the three different use-cases.

As it had been foreseen, our system causes the size of an image on disk to grow by a considerable extent. In the case of httpd and glife, the size is 20% bigger; with redis, the figure grows by 30%. Of course, just like in the previous case, the machine on which this experiment has been conducted is not relevant.

Without any doubt, the most important aspect that ought to be evaluated is the reduction of the **time required to pull a runnable image** from the registry. Figure 4.1 has shown that our system reduces by a great extent the amount of data that must be transferred over the network before container start-up. For this reason, we expect the pull time to be much shorter.

Figure 4.3 graphically reports the average results of the time measurements for container deployment as well as the respective confidence intervals. In contrast to the previous cases, the characteristics of the machines performing the tests are now relevant. For this reason, the data is divided into two separate graphs: Figure 4.3a shows the pull time reduction obtained when running the Docker daemon on a personal computer; Figure 4.3b reports the case of the Raspberry Pi 3. In both circumstances, our system achieves a substantial reduction in terms of duration of



(a) Results on personal computer

(b) Results on a Raspberry Pi 3

Figure 4.3: Evaluation of pull time

the `docker pull` operation. However, there is an interesting difference between the two machines that must be noted. On the personal computer, the time is reduced by a ratio in the range of 2 to 3; on the Raspberry Pi, instead, the operation gets 3 to 6 times faster.

Interestingly, the time required to download the full customized image is slightly shorter than the standard deployment when the operation happens on the Raspberry Pi. One would expect the contrary, due to the presence of the additional base layer in our version of the image. The result can be explained by considering that our system performs the download and extraction of the remaining layers in parallel, since it builds and launches goroutines that perform their job concurrently. On the contrary, the traditional implementation of the Docker daemon follows a sequential order, which, as mentioned in Section 2.5.4, can cause performance degradation within constrained environments.

4.3 Closing remarks

All in all, the results of our evaluations seem to confirm the validity of our approach. By shrinking the amount of data initially transferred over the network, our system is able to achieve a great reduction of the time needed to deploy and start a container on a fresh machine. Furthermore, improvements in resource-constrained environments are even more pronounced. The objective of this thesis was to tackle the problem of slow Docker container deployment by providing a way to speed up the process. The

results of our evaluations prove that our system successfully achieves the objectives. This final part of the chapter is going to expose the strengths of our system as well as some possible improvements.

4.3.1 Strengths

A crucial merit of this work is that short deployment times can **boost the adoption of Docker** containers in a Fog computing architecture. In such paradigm, we want to benefit from the advantages of virtualization without compromising the performances. Docker containers can accommodate services that must follow the user through a geographical area. In such situations, the deployment of a Docker container may be triggered as a reaction to user movements, and it would be appreciable if the system were able to deploy a service on a Fog node before the user has left its vicinity. It is therefore clear that the results achieved by our system represent a major enhancement that can promote the adoption of Docker containers in next generation Fog computing architectures.

As a further strength, it is fundamental to note that our work seems to be **compatible with other optimizations** that have been carried out by previous studies. In this regard, a special reference goes to the work conducted by Arif Ahmed and Guillaume Pierre in 2018 [1]. In their work, the authors managed to reduce the deployment time on single-board machines by 60% to 70%. Our system could be build on top of that, and bring the performance gains to a further level of efficiency.

4.3.2 Ideas for future work

A valuable improvement for the system can be to introduce a **separate command for "fast pull"**, while preserving compatibility with the traditional operation and therefore, most importantly, with traditional images. Over the course of this thesis, the behavior of the standard `docker pull` command has been altered. The choice has be made for the sake of simplicity, as the introduction of a new command would have resulted in plenty of additional work on parts of the Docker architecture which are not related at all to the objectives of this thesis. However, a big drawback of the selected approach is that the system works correctly only with images that have been specifically customized for it. The self-evident solution to the problem can be

to finally introduce a separate command, such as `docker fast-pull`, to launch our own procedure, whereas leaving the standard command available and unchanged. In this way, we would be able to preserve the compatibility of our system with all the general-purpose images that are publicly available on the Internet.

A further enhancement could be to **handle image-removal commands** issued during the background download of an image's layers. Our `docker pull` command returns after the base layer has been deployed. This is necessary in order to let the user start the application with `docker run` before the end of the deployment. However, a user could issue a command to remove the image before the data has been downloaded completely. In such case, the files already on disk would be deleted; on the other hand, the remaining data would still be downloaded and installed by the goroutines, without being accessible by the daemon. This would result both in pointless disk usage and the impossibility for the daemon to get rid of the corrupted data. We can devise two possible solutions for the problem: the first one is to insert a mechanism to disable the `docker image rm` command when a download is still going on in the background; another one is to provide the daemon with communication channels that allow to interrupt the currently executing goroutines when the image they are deploying has to be deleted.

Restoring the integrity check of the layers can also be a promising idea. Section 3.4.1 has shown how Docker checks the correctness of a layer's data after the download from a registry. In our case, such control is harder to perform because most of the layers are empty at the time when such control is supposed to be done. Indeed, their respective content is retrieved by background procedures when the container may be already in execution. In order to avoid errors during this preliminary check, we decided to disable it altogether. A possible improvement would be to reintroduce a control over the integrity of the data of the layers by launching it once they have been deployed. In case of error, the download could be attempted again, transparently for the container.

Moreover, it would be desirable to be able to **disable the wrapper library** once the deployment of the image is complete. In Section 3.5 we have justified the introduction of a wrapper around certain system calls with the need of handling the "unlucky" case in which a container tries to access a file whose download has not been completed yet. Under certain circumstances, the wrapper launches a look-up to check if the file in question is present in the usually long list of all the delayed

files. As a consequence, a significant performance overhead may occur. The problem is that this effort becomes pointless once the whole image has been deployed. For this reason, it would be reasonable to introduce a mechanism by which the Docker daemon could disable the wrapper when the appropriate time comes. A possible approach could be to reset the `LD_PRELOAD` environment variable of the container, which would prevent our library to be loaded before the standard one.

A last opportunity for future work could be to **study the internals of Overlay FS** in order to comprehend how it merges the different directories across the layers of an image. Our experiments have confirmed that by providing the complete directory structure across the layers and dropping the cache of the host file system it is possible to make the unified view pick up the underlying changes. A deeper analysis of the functioning of the Overlay FS may provide some valuable insights regarding possible adjustments that could improve the support for our operations.

Conclusions

Software containers are getting increasingly popular in the last years because, unlike virtual machines, they provide the benefits of virtualization without compromising the application's performance. However, the generally large size of Docker images significantly increases the deployment time, especially on resource-constrained machines. This is a major obstacle for the adoption of the technology in a Fog computing environment, where it is necessary to follow a user by deploying services as a reaction to his movements: if the procedure takes too long, it may be impossible to start a container on a Fog node before the user has left the vicinity.

The objective of this thesis has been to improve the efficiency of the deployment process by reducing the time needed to run a container on a new machine. Results show that our system widely achieves its objectives. The amount of data that has to be transferred over the network and installed on the machine before a container can start is approximately 10 times smaller compared with the original implementation. As a consequence, the time needed to pull and run a container drops up to a factor of 6. For example, when pulling an image containing a binary application to a Raspberry Pi the container is ready to start after just 10 seconds, as opposed to the 60 seconds of a traditional deployment. Improvements are particularly good in constrained environments, which proves that the approach can be of remarkable utility when applied in the context of a Fog computing architecture. Nevertheless, the system can have general applicability as well, since it ensures a speed-up of the pull process regardless of the available computing resources.

Although the work achieves substantial success, a number of further improvements can be considered. A useful enhancement could be to introduce a separate command for the new efficient pull, instead of modifying the behavior of the original procedure; in this way we would preserve backward compatibility with the standard

version of Docker images. Another promising direction can be to combine our system with other optimization techniques proposed in the recent literature. Section 2.5.4 has presented the significant achievements obtained by Arif Ahmed and Guillaume Pierre concerning the reduction of container deployment time. A combination of the two works could bring the performance of the system to a new level of efficiency.

Bibliography

[1] Arif Ahmed and Guillaume Pierre. "Docker Container Deployment in Fog Computing Infrastructures". In: *IEEE EDGE 2018-IEEE International Conference on Edge Computing*. IEEE. 2018, pp. 1–8.

[2] Ali Anwar et al. "Improving docker registry design based on production workload analysis". In: *16th USENIX Conference on File and Storage Technologies*. 2018, p. 265.

[3] IEEE Standards Association. *IEEE 1934-2018 - IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing*. URL: https://standards.ieee.org/standard/1934-2018.html.

[4] Paolo Bellavista and Alessandro Zanni. "Feasibility of fog computing deployment based on docker containerization over raspberrypi". In: *Proceedings of the 18th international conference on distributed computing and networking*. ACM. 2017, p. 16.

[5] Burton H. Bloom. "Space/Time Trade-offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: http://doi.acm.org/10.1145/362686.362692.

[6] Flavio Bonomi et al. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM. 2012, pp. 13–16.

[7] Eric Carter. *2018 Docker Usage Report*. May 29, 2018. URL: https://sysdig.com/blog/2018-docker-usage-report/.

[8] Vitaly Chaykovsky. *strace – linux syscall tracer*. URL: https://strace.io/.

[9] Christopher Clark et al. "Live migration of virtual machines". In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association. 2005, pp. 273–286.

[10] OpenFog Consortium. URL: https://www.openfogconsortium.org/.

[11] Gene Cooperman, Jason Ansel, and Xiaoqin Ma. "Transparent adaptive library-based checkpointing for master-worker style parallelism". In: *Sixth IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*. Vol. 1. IEEE. 2006, 9–pp.

[12] Wayne Davison. *rsync*. URL: https://rsync.samba.org/.

[13] Tyler Harter et al. "Slacker: Fast Distribution with Lazy Docker Containers." In: *FAST*. Vol. 16. 2016, pp. 181–195.

[14] Cisco Systems Inc. *Internet of Things At a Glance*. 2016. URL: https://www.cisco.com/c/dam/en/us/products/collateral/se/internet-of-things/at-a-glance-c45-731471.pdf.

[15] Docker Inc. *Docker: Enterprise Container Platform*. URL: https://www.docker.com/.

[16] Bukhary Ikhwan Ismail et al. "Evaluation of docker as edge computing platform". In: *Open Systems (ICOS), 2015 IEEE Confernece on*. IEEE. 2015, pp. 130–135.

[17] Senthil Nathan et al. "Comicon: A co-operative management system for docker container images". In: *2017 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2017, pp. 116–126.

[18] Martin Pitt. *Fatrace: report system wide file access events*. URL: https://piware.de/2012/02/fatrace-report-system-wide-file-access-events/.

[19] Michael Rieker, Jason Ansel, and Gene Cooperman. "Transparent User-Level Checkpointing for the Native Posix Thread Library for Linux." In: *PDPTA*. Vol. 6. 2006, pp. 492–498.

[20] Subhadeep Sarkar, Subarna Chatterjee, and Sudip Misra. "Assessment of the Suitability of Fog Computing in the Context of Internet of Things". In: *IEEE Transactions on Cloud Computing* 6.1 (2018), pp. 46–59.

[21] J. E. Smith and Ravi Nair. "The architecture of virtual machines". In: *Computer* 38.5 (May 2005), pp. 32–38. ISSN: 0018-9162. DOI: 10.1109/MC.2005.173.

[22] Miklos Szeredi. *Overlay FS*. URL: https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt.

[23] Shanhe Yi et al. "Fog computing: Platform and applications". In: *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*. IEEE. 2015, pp. 73–78.