École polytechnique de Louvain

Improving the performance and the scalability of INGINIOUS

Author: **Guillaume Everarts de Velp**
Supervisor: **Ramin Sadre**
Readers: **Olivier Bonaventure, Anthony Gégo**

Academic year 2019–2020
Master [120] in Computer Science

Acknowledgements

Guillaume Everarts de Velp, 2020

Abstract

Contents

Chapter 1

Introduction

In this chapter I will simply introduce the subject of this master thesis, showing some basic concepts related to it and some key aspects.

1.1 INGInious

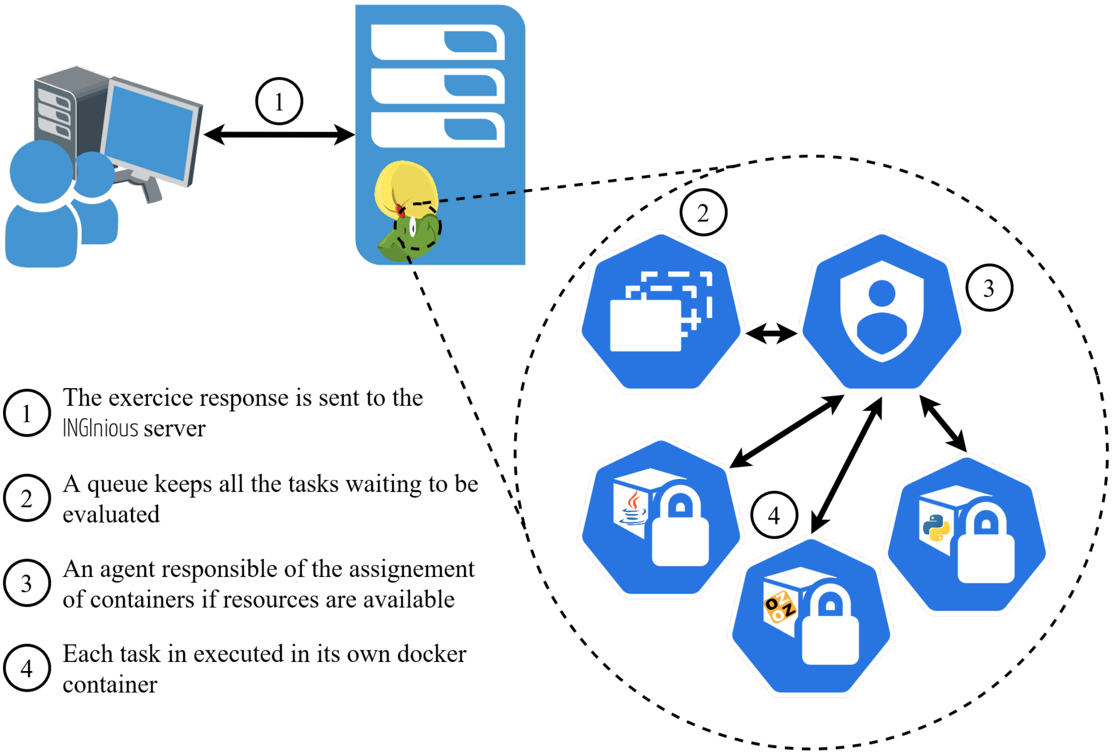
INGInious is a web platform developped by the UCL. It is a tool for automatic correction of programs written by students. It currently relies on Docker containers to provide a good isolation between the machine hosting the site and the execution of the student's codes. So that a problem in a program submitted by a student couldn't have any impact on the platform. Docker also allow to manage the resources granted to each code execution. For now INGInious can meet the demand and provide honest performances and responsiveness. But looking at the growing usage of the platform, we might soon come to a point where we reach the limits of the current implementation.

1.1.1 Architecture

INGInious counts four main components:

1. The front-end: the website with which each student interacts when submitting a task.
2. The back-end: a queue of all the tasks that need to be graded.
3. The docker agent: responsible for the container assignment to the pending tasks when resources are available.
4. The docker containers: one for the student code and one for the teacher tests evaluating the student's code behaviour.

The journey of a task submitted on INGInious is represented on Figure 1.1.



1 The exercice response is sent to the INGInious server

2 A queue keeps all the tasks waiting to be evaluated

3 An agent responsible of the assignement of containers if resources are available

4 Each task in executed in its own docker container

Figure 1.1: INGInious global architecture

1.1.2 Key features

The key features of INGInious, that allow it to meet the requirement of a code grading plateform are the following:

- Isolation between the student's code and the platform.
- Resource limitation (CPU, RAM, Network) for the student's code execution.
- Modularity and versatility regarding the tasks that INGInious can correct. Multiple programming language are supported, and new ones can easily be added.

1.1.3 Bottlenecks

When a task is submitted, we can count five delays before the answer can be delivered to the student:

- The sending time: the time it takes for the task to be sent to the back-end.
- The waiting time: the time the task will spend in the queue, waiting for an available container.
- The booting time: the time it takes to the container to boot and be ready to evaluate the task.
- The grading time: the time it takes for the code to run and for the teacher's container to grade it.
- The response time: the time to send the reponse back to the student.

For the first and the last one, supposing that the machine hosting the website is not overwhelmed, the delay depends entirely on the network, this is a bit out of our hands here. The waiting time is directly related to the current load on the platform, this is a more a symptom of the server overwhelming than its cause, it could be directly solved by using a scaling strategies (see section 1.2). And then we come to the booting time and the grading time, which directly depends on the containerization technology used and on the hardware performances, this is were we are going to try to improve things in this thesis.

1.2 Scaling

Currently, the resources provided to INGInious vary depending on the load that the platform is expected to be facing. Typically, when the grading of an exam is done by INGInious, the platform is scaled up, and during the holidays it is scaled down. When it comes to scaling, two strategies can be used; vertical scaling and horizontal scaling.

Vertical scaling consists in adding more resources on a single machine, to improve its performances when needed. For example, when the number of tasks arriving to the server grow, we could increase the number of virtual Cores allocated to the Virtual Machine hosting the platform in order to be able to threat more of them concurrently. If the size of the waiting queue is increasing, we might want to make more RAM available.

Horizontal scaling consists in sharing the workload across multiple machines, so that each machine can handle a small part of it. This is a solution widely used nowadays as it allows to scale up virtually indefinetely, which is not the case with vertical scally where we depend on the maximum capacity of the hardware. This requires to rethink the architecture of the platform globally.

1.3 Intentions

The master thesis aims at improving INGInious, regarding its performances and its scalability. To do so, I will search and compare different containerization technologies that could be used instead of Docker. The goal is to find an alternative that decreases the booting time (and the grading time) without loosing any of the key features of the platform. If such an alternative is found and proven to be worth the change, INGInious could then be refreshed with it.

Chapter 2

State of the art

In this chapter I will present how things are today in the field of containerization. I will start with a quick reminder of the difference between virtualization and containerization, then go through some existing solutions for both of those concepts.

2.1 Virtualization vs Containerization

Both virtualization and containerization aims at providing an abstraction layer that allows the application or the OS (Operating System) located above the abstraction to behave like if it was the only one present on this layer. The key difference between those two is where this abstraction layer is located.

To keep it simple, for the virtualization, the abstraction is between the OS and the hardware, the OS thinks it is running on a baremetal machine but is actually hosted on another machine. For the containerization, the abstraction is between the application and the OS, the application thinks it is the only one running in the system but it is actually only isolated in its own namespaces. Multiple containers on a same host share then the same kernel. As a good illustration is better than a thousand words, Figure 2.1 represents this difference.

Because of what they are, those two solutions usually didn't apply to the same situations. A VM (Virtual Machine) is a much more complicated and complete structure, that takes more time to boot but provides a better isolation. And the Container is much lighter, quicker to launch but less isolated. Traditionally, in the Cloud Computing world, VMs rely in IaaS (Infrastructure-as-a-Service) while Containers rely in PaaS (Platform-as-a-Service). This paper [12] made a detailed point about it back in 2014.

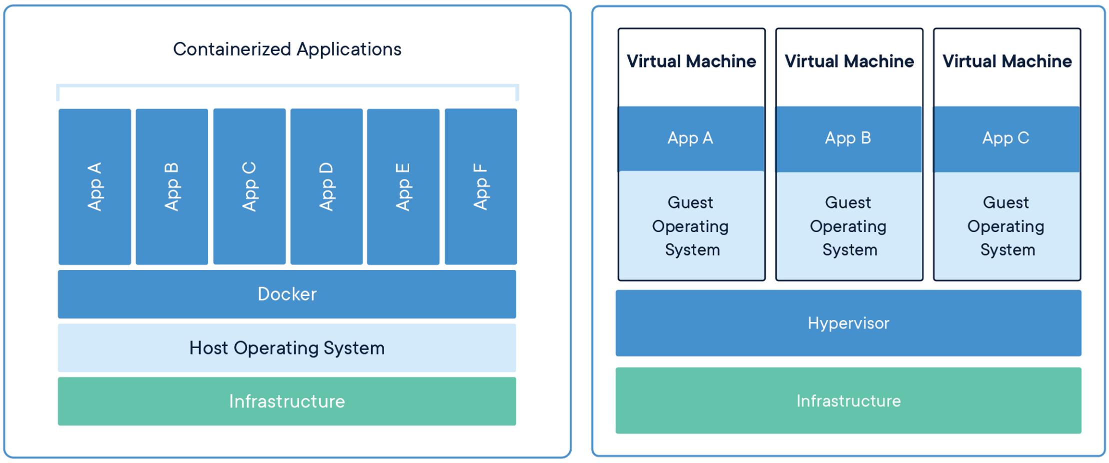


Figure 2.1: This image comes from Docker's website [6].

But recently things started to change, some new solutions seem to have the advantage of both sides, the isolation of a VM and the portability and lightness of a Container. This is why some VM-based solution are presented here. Even though VMs do not come from the same domain of application as Containers, and even though in the case of INGInious, Containers were 5 years ago the obvious choice, it might now be time to reconsider it.

2.2 Serverless

Serverless can be seen as an improvment of Paas, with better scalability and efficiency. This is often called Faas (Function as a service), two open-source solutions for this are Apache OpenWhisk and OpenLambda. It can really still be associated with what INGInious does when testing a submission; it is basically, running a function, on user demand, with no direct link with the structure of the application. The main difference between what INGInious does and what serverless offers is that usually, in serverless, you run a bunch of functions over and over again, only with different inputs. With INGInious, the different inputs to those functions are the code of the student to execute, which makes the function execution unique for each submission. The function as a service style is often referenced as the Lambda model (from Amazon's Lambda). Compared to a container, the isolation here will be even at a higher level, as the same runtime is usually shared between multiple execution of the function. [13] This makes serverless solutions not really appropriate for INGInious use case, as we really *need* to provide an isolation between student's tasks execution.

2.3 Namespaces

Namespaces are a feature of the Linux kernel since 2002, they are a key element that made containerization possible. A namespace can be associated to a context, which is a partition of all the resources of a system that a set of processes has access to, while other processes can't. Those sets of resources can be of seven kinds:

- **mnt** (Mount): This controls the mount points. Processes can only have access to the mount point of their namespace.
- **pid** (Process ID): This allows each process in each namespace to get a process id assigned indepedently of other namespaces porcesses.
- **net** (Network): This provides a virtualized network stack.
- **ipc** (Interprocess Communication): This allows processes of a same namespace to communicate with one another, for example by sharing some memory.
- **uts** (Hostname): This allows to have different hostnames on a same machine, each hostname being considered as unique by the processes of its namespace.
- **user** (User ID): This allows to change the user id in a namespace.
- **cgroup** (Control group): This allows to change the root cgroup directory, this virtualize how process's cgroups are viewed.

When a Linux machine starts, it initiate one namespace of each type in which all the processes run. The processes can then choose to create and switch of namespaces.

2.4 Control groups

Control groups are a feature of the Linux kernel that allows to limit and control the resources allocated to some processes. You can for example control the cpu usage, the memory consumption, the io... Recently (since kernel 4.5) a new version of cgroups (cgroups v2) appeared, which comes to tackle the flaws of the original implementation, while keeping all of its functionalities. Though the adoption of the new version is a process, and takes times, and still now many applications use the original version of cgroup (like Docker).

2.5 Containerization solutions

Here bellow are presented some containerization solutions.

2.5.1 Docker

Docker [15] is today probably the most known solution for containerization. Since its apparition in 2014, its interest for the PaaS sector hasn't ceased to grow. It is now tightly binded with Kubernetes, "an open-source system for automating deployment, scaling, and management of containerized applications" [4]. It consists in a daemon, running on the host, that allows to easily manage different containers. It relies on Containerd, "An industry-standard container runtime with an emphasis on simplicity, robustness and portability" [2], which itself uses runc as runtime for the containers themself ("runc is a CLI tool for spawning and running containers according to the OCI specification." [5]).

The main advantages of Docker are its simplicity of use, its production-grade quality, the huge fleet of ready-to-run containers publicly available and a lot of useful tools (like `docker-compose` that come along with it). This is the solution currently used by INGInious.

2.5.2 Podman

"Podman is a daemonless container engine for developing, managing, and running OCI [3] Containers on your Linux System." [7] Podman presents itself as a viable true alternative to Docker. Its main difference is the fact that it runs daemonless, it runs containers as child processes, it isn't therefore based on containerd as docker is. It can also manage pods, which are groups of containers deployed on the same machine. Its default runtime is runc as well. Podman is an Open-Source project, and is still growing a lot, the latest version at this day (2020-02-28) is v1.8.0, released 21 days ago, and already 287 commits have been done since then!

2.5.3 LXC

"LXC is a userspace interface for the Linux kernel containment features." [8] This solution is developed and maintained by Canonial Ltd. Docker used to be based on LXC until it created its own execution environment. LXC came out with a new solution; LXD, which offer about the same things as Docker does. A bunch of ready to run images publicly available, and a nice command line interface to interact with containers.

2.5.4 SAND

SAND [10] aims at improving the performance of serverless computing with two techniques: application-level sandboxing, and a hierarchical message bus:

- **Application Sandboxing**: The key idea here is to differentiate multiple executions of different functions with multiple execution of a same function. In the first case, as usual, a new container is launched on the new function demand and the function is executed inside of it. In the second case, instead of lauching a new container with exactly the same configuration as an already running one, we only fork a process inside de running container to deal with the new demand, which is much faster than creating a new container. The problem in our case, is that we loose the isolation between student code execution, which is not desirable.
- **Hierarchical Message Queuing**: The goal to this is to facilitate communication between functions that interact with one another (i.e. the output of a function is the input of another one). To do so they use a two-level communication bus: global and local. Global means that the communication is made between different functions on different hosts, while local means different functions on the same host. As accessing the local bus is much faster than the global one, it can decrease latency for functions on the same host.

2.5.5 SOCK

"Sock (roughly for serverless-optimized containers), [is] a special-purpose container system with two goals: (1) *low-latency invocation* for Python handlers that import librairies and (2) *efficient sandbox initialization* so that individual workers can achive high ready-state throughput." [16] The final product created here isn't really something we could use for INGInious, it is a serverless solution (based on the Lambda model), targeting specifically python applications. Though, in the process of creating this solution, they started from containers and deconstructed their perfromances, indentifying their bottlenecks. And from this we can take those things:

- Bind mounting is twice as fast as AUFS. Even though AUFS (used by Docker) as a usefull Copy-on-write capability, we don't need to write to most of our files in our case, so using a read-only bind mounting for those files could allow us to avoid copying all file before startup, while still beeing able to edit the one with want to.
- Network namespaces creation and cleanup are costly, due to a single lock shared across all network namespaces. We might gain some performances buy not adding network interfaces to containers that don't need it.
- Reusing a cgroup is at least twice as fast as creating a new one each time. We could keep a pool of initialized cgroups, and only change the current container it controls.

2.6 Virtualization solutions

Here bellow are presented some lightweight virtualization solutions.

2.6.1 LightVM

LigthVM is a complete redesign of Xen's toolstack which tried to bring some container's characteristics to VMs. Such as fast instantiation (small startup time) and high instance density capability (high number of instances running in parallel on the same machine). [14] Xen is a Type-1[1] hypervisor presented in 2003 with really low virtualization overheads and high hosting capacity, allowing a machine to host up to a 100 guest OS. [11]

They achieve such container's like performances by:

- reducing the image size and the memory footprint of virtual machines. They do so by including in the VM only what is necessary to the application that is meant to be executed in it.
- introducing noxs (no XenStore), a new implementation of Xen, without XenStore (which was a real bottleneck for fast instantiation of multiple VMs).
- splitting the Xen's toolstack into what can be run before the VM creation and what as to be done during it. Allowing to pre-initialize VMs.
- replacing the Hotplug Script by xendevd, a binary deamon that can execute pre-defined setup more efficiently.

They present four use cases with this solution, one of them beeing more intresting for us: lightweight computation services, for which they rely on Minipython unikernel, to run computations written in Python, as a Faas could propose to do. This would still be hard to use for INGInious, for the same reason as for SOCK (§2.5.5).

2.6.2 Firecracker

Firecracker is a new VMM (hypervisor) created specifically for serverless and containers applications. [9] This is a solution provided by AWS, that very recently got deployed for two of their web services: Lambda (Faas) and Fargate (Paas). We are basically getting here the good isolation of virtual machines and nearly as good

[1] A Type-1 hypervisor is an hypervisor that runs on a bare-metal machine, whithout additional host.

performances and low overhead of containers. Firecracker is based on KVM and provides minimal virtual machines (MicroVMs). The configuration is done through a REST API. Device emulation is available for disks, networking and serial console. Network can be limited and so can disk throughput and request rate. If proven to be easily usable in INGInious case, this would provide a better alternative to Docker regarding security.

2.6.3 Kata Containers

"Kata Containers is an open source container runtime, building lightweight virtual machines that seamlessly plug into the containers ecosystem." [1] They put forward four features of their solution:

- **Security**: Thanks to their virtualization solution, each runtime has its own kernel, with real network, i/o and memory isolation.
- **Compatibility**: They support industry standarts, as the OCI [3] and legacy virtualization technologies.
- **Performance**: Their performances are consistent with classical containerization solutions.
- **Simplicity**: They eliminate the need to have a virtual machine dedicated to host containers. This is not really something for us though, as we still need to run INGInious on the host VM.

Same as for Firecracker, if proven to be easily usable in INGInious case, this would provide a better alternative to Docker regarding security.

Note that as it is only a container **runtime**, it couldn't replace completely Docker on its own, it would be more a replacement for runc, the current default container runtime of Docker. Kata Containers actually even proposes to use Docker to manage its runtime. Kata containers also allow to integrate Firecracker as VMM instead of QEMU, which is the default one.

2.7 Summary

In this section, on Table 2.1, are quickly reminded the several solutions we explored in this chapter, along with some interesting infos about them.

Caption:

- *Name*: The name of the project.

Name	**Pub.**	**Update**	**Open-Source**	**Pot. Sol.**	**Com.**	**Isol.**
Docker	2013	2020	Yes	Yes	Yes	Cont.
Podman	2018	2020	Yes	Yes	Yes	Cont.
LXC	2014	2019	Yes	Yes	Yes	Cont.
SAND	2018	?	?	No	No	Cont.
SOCK	2018	?	?	No	No	Cont. or Runt.
LigthVM	2017	2017	Yes	No	No	Virt.
Firecracker	2019	2020	Yes	Yes	Yes	Virt.
Kata cont.	2019	2020	Yes	Yes	Yes	Virt.

Table 2.1: Summary table of the different solutions explored in this chapter.

- *Pub.*: The first publication year of the project.
- *Update*: The last update year of the project.
- *Open-source*: If the project is open-source.
- *Pot. Sol.*: If the project could be a solution in our case.
- *Com.*: If the project is a "commercial grade" solution.
- *Isol.*: The type of isolation used in the project.
- *Cont.*: Isolation by containerization.
- *Virt.*: Isolation by virtualization.
- *Runt.*: Isolation by the runtime of the application.

Chapter 3

Benchmark tool

3.1 Setup

3.2 Tests

Chapter 4

Measurements

4.1 Results

4.2 Interpretation

4.3 Recommendations

4.4 Summary

Chapter 5

Conclusion

Appendix A

Container life cycle

During its life cycle, a container will go through five stages: creation, launch, execution, stopping and cleaning. Those five steps are explained below and presented on figure A.1, along with the corresponding command for each presented tool. Other optional stages (like pause/unpause) are not presented here.

1. **Create** This is the set-up phase, depending on the tool, the file system of the new container might be copied, or any other things that need to be done before launching the container. This phase can only be done one time by container.

2. **Launch** This is the proper instantiation of the container, the namespaces are created, the new file system is adopted. Depending on the container, a complete init process might me executed. This state can be repeated as many times as we want for a container, as long as it is in a stopped state.

3. **Execute** This corresponds to the execution of a process in the container. This can be done as many times as we want for a container, as long as the container is running.

4. **Stop** This is when the container need to be stopped, all running process are killed, and the namespaces are exited. No modification should be done on the filesystem, the storage of the container remains.

5. **Clean** This is the un-create phase, where we delete everything that was done during the creation phase.

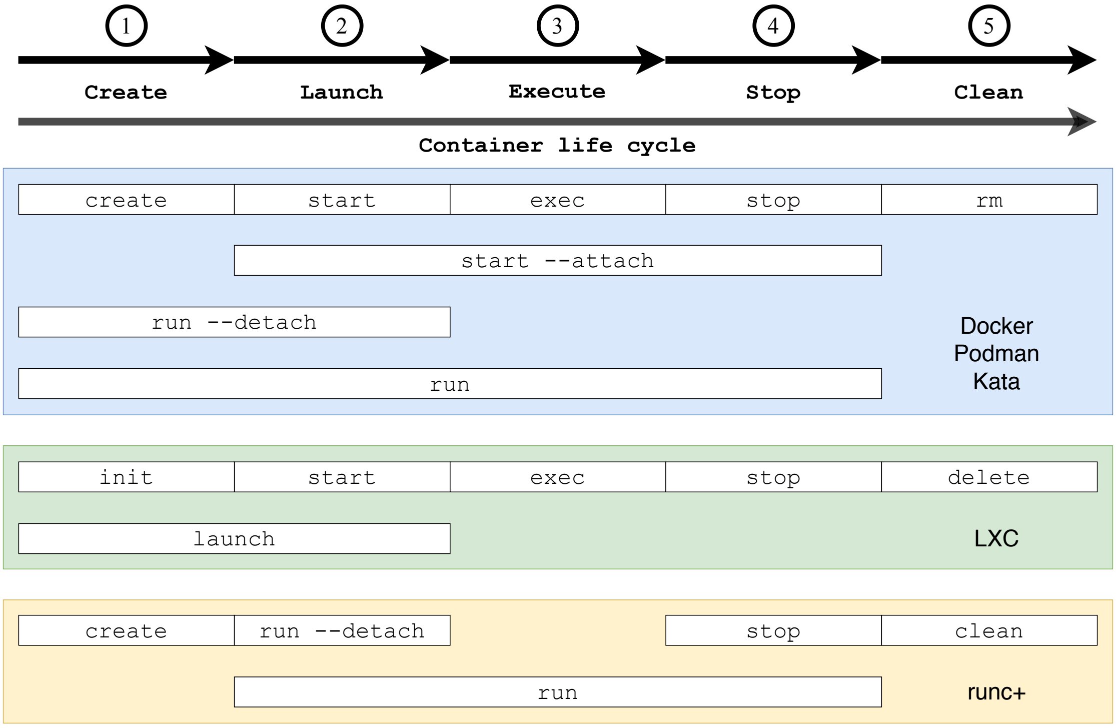


Figure A.1: Container life cycle and how to interact with it.

Appendix B

Storage drivers

When it comes to handle a container file system, different solutions can be used. The goal of each is to provide the most efficient writable root directory (/) for each container, but keeping each container inaffected by the modifification done in the other containers.

In order to do so, three main strategies can be used:

- **Deep copy**: for each container, the whole image is copied during the creation of the container. This is simple, but gets terribly slow as the file system size increases.
- **File based copy on write**: for each container, will be copied only the files that are edited during the container life cycle. This is more complex, but get more efficient as the container's size grows.
- **Block based copy on write**: for each container, will be copied only the blocks (in the filesystem) that are edited during the container life cycle. This is even more complex, but get more efficient as some small part of big files are edited.

Containers are a specific kind of workload in the sense that many information, data, is redundant in different containers. For example, for a simple application, we could use several containers with different responsabilities and tools embedded in it, but all based on the same Alpine image. This brought a new space problem, as we don't want to avoid duplicating too much data. In order to face this, **union filesystems** are used, along with layered container images. This basically means that different container images but with the same basis, will actually share this same basis, avoiding the need to duplicate it.

Here is a non-exhaustive list of current available solutions for container storage:

- `overlay` and **overlay2** (*Docker*): Those are based on OverlayFS (Linux kernel driver), a union filesystem, similar to AUFS, but Docker claims it to be faster and simpler. Overlay2 is the new and more stable version of overlay.
- `aufs` (*Docker*): This solution is based on AUFS, a union filesystem. For Docker, it was the predecessor of overlay, and is a bit less performant than the latest.
- `btrfs` (*Docker*, *LXC*): Btrfs is a copy-on-write filesystem. Docker's and LXC's btrfs storage driver rely directly on it, using its block-level capabilities, thin provisionning and its copy-on-write snapshots.
- `zfs` (*Docker*, *LXC*): ZFS is another filesystem, with many features, like snapshots, compression, deduplication and more. Docker's and LXC's zfs storage driver rely on it, taking advantages of its capabilities.
- `vfs` (*Docker*): This is the simplest and yet more reliable storage driver. As it doesn't provide any advanced feature. It as no copy-on-write capabilities. Each container filesystem is compied on creation. The performances of this driver are very poor then.
- `directory` (*LXC*): This is the exact equivalent of vfs.
- `devicemapper` (*Docker*) [deprecated]: This relies on Device Mapper, a kernel-base framework with some interesting capabilities as snapshots and thin provisionning. This is a block-based approach.
- `lvm` (*LXC*): This is the exact equivalent of devicemapper.

Note that the `storage` Go library for containers, wrapped under the `containers-storage` CLI[1] provide support for all of those drivers[2].

[1] https://github.com/containers/storage

[2] https://github.com/containers/storage/tree/master/drivers

Bibliography

[1] About kata containers. https://katacontainers.io/. Accessed: 2020-03-04.

[2] An industry-standard container runtime with an emphasis on simplicity, robustness and portability. https://containerd.io/. Accessed: 2020-02-26.

[3] Open container initiative. https://www.opencontainers.org/. Accessed: 2020-03-04.

[4] Production-grade container orchestration. https://kubernetes.io/. Accessed: 2020-02-26.

[5] runc's github repository. https://github.com/opencontainers/runc#introduction. Accessed: 2020-02-28.

[6] What is a container? https://www.docker.com/resources/what-container. Accessed: 2020-02-24.

[7] What is podman? https://podman.io/whatis.html. Accessed: 2020-02-28.

[8] What's lxc? https://linuxcontainers.org/fr/lxc/introduction/#whats-lxc. Accessed: 2020-03-01.

[9] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications.

[10] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[12] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *2014 IEEE International Conference on Cloud Engineering*, pages 610–614. IEEE, 2014.

[13] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[14] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[15] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[16] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**