UCLouvain

epl

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 and Engineering

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. To check the validity of a student's code, the latest will be executed, feeded with some inputs, and the outputs generated will be checked. 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.

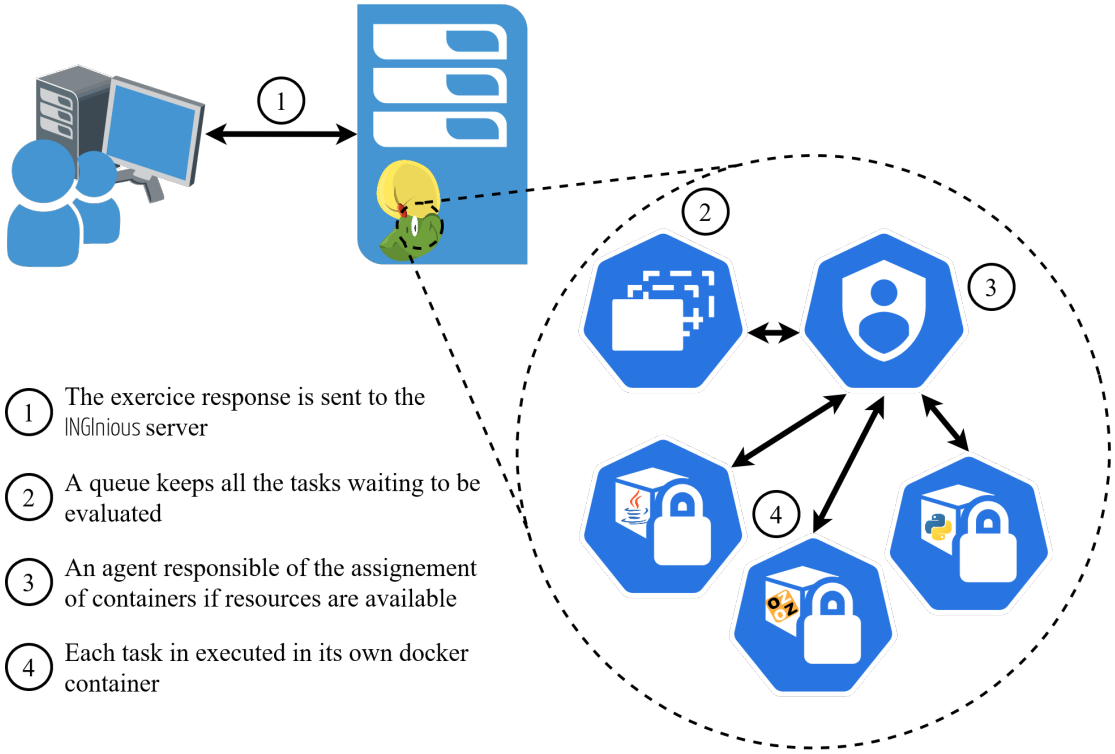

Figure 1.1: INGInious global architecture

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, Pids) 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 see if we can 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 grows, 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 and configurations that could be used instead of Docker's current configuration. The goal is to find if an alternative could 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

Background knowledge

In this chapter I will put some basis, and explain some concepts related to my work. Reading this chapter should allow the reader to understand what I did in this work, supposing it is already familiar with Linux.

2.1 Virtualisation, Containerisation, Runtime isolation

We always want the same thing, be able to execute some code, with no interaction with someone else's one. As of course we can not use one bare-metal machine for each application we want to run, some mechanism have to be used to provide isolation between coexisting processes on a same physical machine.

We can distinguish three level of isolation: Virtualisation, Containerisation and Runtime isolation (Figure 2.1).

2.1.1 Virtualisation

The isolation layer is located either between the hardware and the guest OS (hypervisor of type 1) or between a virtualisation of the hardware and the guest OS, by an hypervisor (type 2) running in another OS. In both cases, the guest OS, will run as if it was on a bare-metal machine, executing its own kernel, managing its drivers, its file system, etc. This is the stronger isolation we can get. Cloud providers can rent you some Virtual Machines, this is what we call IaaS (Infrastructure as a Service). You can use many different hypervisor solutions, usually, Cloud providers have their very own solution.

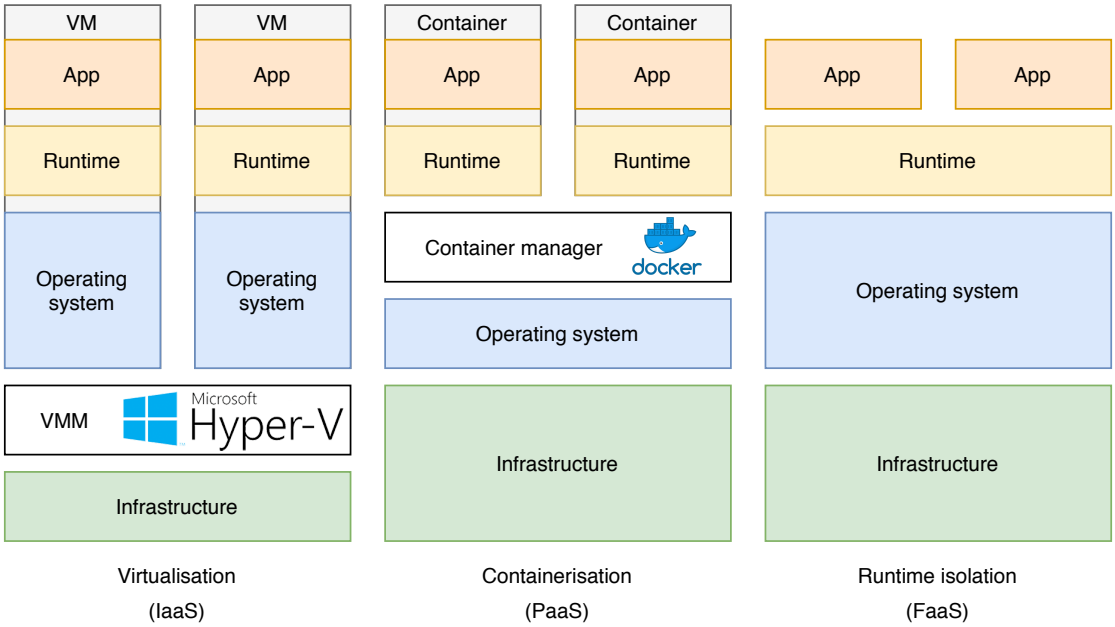

Figure 2.1: Different isolation level for an application.

2.1.2 Containerisation

Here we go one level higher, the isolation layer is between the OS and the guest application. This isolation mechanism is called a container, and is basically a set of processes, running in common namespaces, with a root filesystem different from the host. Containers share the same kernel as the host though, which means that a process running in a container can be seen from the host. Containers belong to the world of PaaS (Platform as a Service), a system where Cloud providers offers you to run your application (that you provide under a containerized form) whithout having to worry about all the infrastructure than needs to be deploied along with it, and sometimes even with some great scalability mechanism support.

2.1.3 Runtime isolation

This is the highest (weakest) level of isolation you can get. The isolation is provided by the runtime on which the application is running (ex: The JVM for a Java program, a Python environment, ...). Multiple guest applications share the same OS, file system, whithout any mechanism to hide it from them. Cloud providers propose such service under the name of FaaS (Function as a Service). For this you can provide a small program, that you want to be executed on demand. Cloud providers usually allocate a container to this program, to provide a safer isolation,

but will execute multiple instances of this program in the same containers, to avoid the overhead of creating a new containers each time. The kinds of programs you usually run in these are computionnaly intensive parts of your application, that needs to have low response time and great scalability (for example resizing of user uploaded images). This model is often reference as the Lambda model (from Amazon Lambda service) or as serverless computing.

2.1.4 INGInious use case

The case of INGInious is a special one. The type of the workload caused by the tasks evaluation (fast response time, varying demand, potentially big computation, logic independent of the core application) would suggest to use a serverless strategy, but the isolation requirement are those of PaaS (as the various inputs of the functions are code of student to safely execute, isolated from one another). And for now the all application relies on IaaS, running in multiple VMs where are hosted the core application and multiple docker agents.

2.2 Containers

The isolation being the most important requirement, we will then look into containerisation solutions. Presenting the main concepts behind containerisation, the global container ecosystem, the current solution that INGInious uses and the alternatives we have.

2.2.1 Core concepts

Containers rely on three components: namespaces, control groups and chroot.

Namespaces are a feature of the Linux kernel since 2002. 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 processes. The first process in a namespace will then get the pid 1 inside of the namespace, but another one outside of it.

- **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. This way you could have a user root in the namespace, which actually isn't outside of it.
- **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.

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. The main change is the use of a new unified hierarchy to manage the control groups, where all the resources are centralized and where the allocation of some resources is done by adding a sub-resource-group in the hierarchy the current user has access to. Though, the adoption of the new version is a process, and takes times, and still now many applications use the original version of cgroup.

Chroot allows to change the root of the root filesystem for some processes. For example, we could create a directory `/tmp/myroot/` which contains all of the usual directories present in the original root folder (`/`) and set this as the new apparent root for the chosen processes. This is not a complete sandbox, and not a real isolation on its own, files from outside of the chroot could still be accessed.

2.2.2 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 modification 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 the common part of their file system, avoiding the need to duplicate it. It differs from file system deduplication mechanism by the level at which the action is taken. For deduplication, when a block is written on disk, we check if similar data isn't already located somewhere. Whereas for union file system, the common base image will be read-only, and when an attempt to modify a file is made, a bind mound will be made with a copy of the file on top of the original one, so that the common base stays preserved.

The storage of a container has to be backed by a file system, which will eventually already provide some interesting mechanisms for the container. Those four main filesystem used are:

- **BTRFS**, presented in 2013, [19] is a general purpose file system, based on copy-on-write and with efficient snapshots capabilities and strong data integrity.
- **zFS** is quite similar to BTRFS (but was there before), but handles some mechanisms differently, like the management of blocks (blocks vs. extends[1]),

[1] zFS can have blocks to up to 128KB, while BTRFS will only use the extend strategy (point to the next block).

snapshots (birth-times vs reference-counting[2]), ... It also support deduplication, which can be usefull for system were a lot of data is redundant and disk space is valuable.

– **xFS** is a network file system (as zFS and BTRFS), this means that it can manage several drives in different locations, connected over the network in one storage unit. It manages to deliver better performances and availability than other similar solutions at the time it was created (1993). [20] It has no focus on any copy-on-write mechanism as the first two though.

– **ext4** was meant to be the replacement to ext3, as the "Linux Filesystem" when it was presented in 2007. [16] It aims at providing a good balance between scalability, reliability, performance and stability. It doesn't provide any of the fancy feature of the previous file system.

The Table 2.1 present a short summary of the different storage driver available today to use with container managers.

Storage driver	**C-o-W**[3]	**FS**[4]
`overlay`	file based	ext4, xFS
`aufs`	file based	ext4, xFS
`devicemapper`	block based	direct-lvm[5]
`btrfs`	block based	BTRFS
`zfs`	block based	zFS
`vfs`	no	any
`directory`	no	any
`lvm`	block based	ext4

Table 2.1: Summary of storage driver solutions.

2.2.3 Container runtime

A container runtime is a tool that creates containers, executes process in it, and deletes dead containers. It will have the responsability to create the namespaces, change the root directory, and attach processes to a control group. The most

[2]When doing snapshots, zFS will store on the block the time at wich a new use of the block has been added, while Btrfs handles it with a reference-counting mechanism.

[3]Copy-on-write

[4]Backing file system

[5]Note that this is a logical volume manager, not a file system, which uses in our case the xFS file system

common container runtime nowadays is `runc`, created by the Open Container project.

2.2.4 Container manager

A container manager will use the container runtime, to provide a "user-friendly" interface to manage containers. It has the responsability to set up the network interfaces, to provide the image of the containers and any Copy-on-write mechanism that could go along with it. Some of them offer the possibility to create custom images, to create pods, or swarm, which are entities of multiple interconnected containers.

2.2.5 Rootless containers

As we saw previously, creating a container requires to create a bunch of namespaces, and launch some processes in it, limiting their resources with control groups. If you don't do this manually, all of this is taken care of by the container runtime. There are two modes in which you can run containers:

Rootfull This is the default choice when using Docker, the processes launched in the container are owned by root, and the container runtime is executed as root as well. The container manager also need root permissions then. It means that either the user willing to launch a container has to be root, or some trick has to be used to allow the user to gain the priviledge required to launch the containers. Docker does it by allowing any user member of the group `docker` to send requests to the daemon, which runs as root and can do the required operations.

Rootless This is less common, and much more secure. The processes launched in the container are not owned by root (but are mapped to the root uid inside the container) and the container runtime is not executed as root either. This requires the use of second generation control group to have resource management capabilities, as the unified hierarchy allows any user to create more sub-resource-group, to manage the resource it has access to. It means that we don't need to use a daemon or any other trick to give a user permission to do "root stuff" on the host.

There is also an intermediate solution, which actually map the root user of the container to an unpriviledged user on the host, as in the rootless case. It means that processes running as root in the container won't be root on the host, which is already better, but the container runtime still runs as root.

2.3 Ecosystem overview

A small overview of the current container ecosystem can be found on Figure 2.2. Note that solutions like Kubernetes[6] which are more oriented towards hosting and continuous deployment of container based applications than to single container provisionning are not presented here[7].

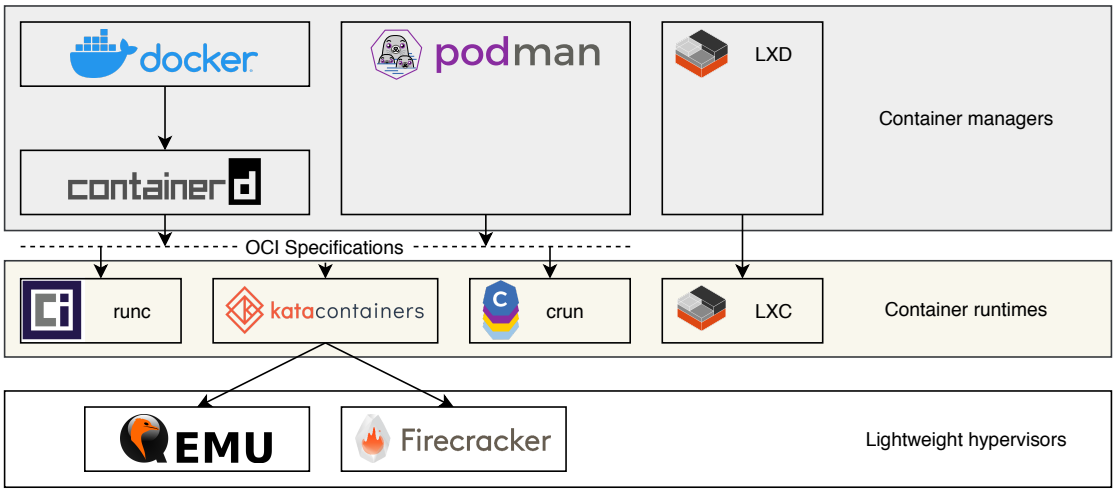

Figure 2.2: Small overview of the current container ecosystem

OCI (Open Container Initiative) is "an open governance structure for the express purpose of creating open industry standards around container formats and runtime." [3] They currently have two specifications: the format of the image that has to be given to an OCI compatible runtime, and the commands to interact with such runtime.

2.4 Containerisation solutions

Docker [17] is today probably the most known container manager solution for containerization. Since its apparition in 2014, its interest for the PaaS sector hasn't ceased to grow. It consists in a daemon, running on the host, that allows to easily manage different containers. It relies on Containerd, "An industry-standard

[6]Kubernetes is "an open-source system for automating deployment, scaling, and management of containerized applications" [5]

[7]A more detailed overview for those kind of applications is provided by Containerd at the following address: https://containerd.io/img/architecture.png

container runtime with an emphasis on simplicity, robustness and portability" [1], which is a more complete container runtime than what I presented before, with embedded network management, storage driver management, and other mechanism. Containerd is not meant to be used standalone though, which is why we still use Docker. By default, Docker uses `runc` as container runtime, but is compatible with any runtime that fulfill the OCI [3] requirements.

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-swarm` that come along with it). This is the solution currently used by INGInious.

"**Podman** is a daemonless container engine for developing, managing, and running OCI [3] Containers on your Linux System." [6] Podman presents itself as a viable true alternative to Docker. Its main difference is the fact that it runs daemonless, it runs containers as detached child processes and containers can be rootless by default (while it is only a recent feature coming up to Docker). 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 (2020-05-30) is v1.9.3, released 8 days ago, and already 797 commits have been done since then!

"**LXC** is a userspace interface for the Linux kernel containment features." [7] 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 recently 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. The only missing feature of LXD compare to Docker, regadering our use case, is the possibility to launch a container with a command, and stop it when it is finished. LXC actually start a complete init process for each container, in which you can then come and execute your command.

Linux-VServer [2] is a bit of a special one. No namespaces are used to provide isolation here, they rely on contexts to separate multiple virtualised systems. They have modified the Linux kernel to provide some utilities like process isolation, network isolation and CPU isolation. The main advantage of this approach is scalabality as it can run a large amount of containers, the drawback is the portability. It requires a modifed version of the kernel and it also misses some features like checkpoint and resume that we can have with more classical containerization and real virtualisation. This is an project much older than Docker and its contemporaries and differ in goal, it is not appropriate for the INGInious workload. This is an

experience more similar to what classical VM provides, where you will setup multiple application in one environnment. The container is not build for one single purpose.

OpenVZ [4] is another container-based virtualisation solution. But this time they use namespaces to provide it. They also have their own way of dealing with system resources management, more complete than cgroup. Even though this is more close to what runc and LXC provide in therm of isolation, this is still not the kind of experience appropriate for INGInious use case. This is designed, as for Linux-VServer, for a more complete server experience.

Kata Containers is not another container manager. It is an OCI compatible runtime. With the specificity that it doesn't run mainstream containers as `runc`, but actually lightweight virtual machines, using KVM virtualisation and with Qemu (originally), Firecracker or Cloud Hypervisor (still in its early days) as hypervisor. Those two first currently available hypervisor are not really equivalent though, as the original one has more features (like devices assignment) and the second one has lower memory footprint and smaller attack surface (smaller code base). 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.

crun is a complete equivalent of `runc`, yet another OCI compatible runtime, but this one is implemented in C, which give it a small performance advantage over `runc`, implemented in GO. `crun` has also full support for cgroupv2, which is still lacking for `runc` at the moment (2020-04-22).

Chapter 3

Related work

This is not the first time that someone tries to compare different containerization solutions. In this chapter I will present other work that has been done for that matter, and what I can add up to that. I will end up by presenting some improving solutions that some people tryied to bring to the container solution panel.

3.1 Performance studies

2013 *Performance evaluation of container-based virtualization for high performance computing environments* [21]: They compared a native experience to LXC containers, OpenVZ, VServer and Xen. They showed that virtualisation (with Xen) was giving really poor performances compared to the others (which give nearly-native experience), in terms of I/O, memory bandwidth, and even network latency and bandwidth. What they have to grant to virtualization though, is the better isolation it provides, as an application running in a VM could run almost seemlessly when the system is stressed (in another VM) with varying tests. While for other solution the impact can vary a lot, even making it impossible for the application to start in some case.

2014 *Virtualization vs containerization to support paas* [11]: Virtual Machines have been kept away from the PaaS world, because of the greater overhead that Virtualization has on booting time and resources consumptions of the isolated system created. This paper made a detailed point about it, emphasizing on three points that needed to be improved for the next generation of containers: Security, OS Independence and Standardization. The last one meaning that containers runtime should establish some common requirements, come up with a common container format. We have seen this appear since then, under the OCI (Open Container Initiative) specifications. They also showed that

a tradeoff with the performances (mostly I/O) of the application running isolated is much more evident with virtualisation than with containerisation.

2015 *An updated performance comparison of virtual machines and linux containers* [12]: In this paper they show the overhead that both containerization and virtualization have, compared to a bare-metal system. The most important point is the I/O comparison, where they show that virtualization (here with kvm) provide much lower performances than containerization (with docker, using aufs). And docker performances can even be improved to nearly meet the native ones when using volumes to store the solicitated files.

2017 *Performance overhead comparison between hypervisor and container based virtualization* [14]: In this paper they showed that even though container-based virtualization (with Docker) is more lightweight than real virtualisation, its performances are not always better (they are actually worse when trying to read or write single bytes to disk). And they plan to compare it into more details later.

2017 *A performance comparison of container-based technologies for the cloud* [13]: They compared here two different containerization solutions (Docker and LXC) against the native experience with focus on CPU usage, memory consumption, network and I/O performances. Once again, overheads are detected for I/O operations, where high demand of input-output operations have a greater latency that a native experience would have.

3.2 Improvement studies

3.2.1 SAND

SAND [9] 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. For now in the case of INGInious, student and teacher container always run on the same host and communicate though the mean of file descriptors. So this is still not something for us.

3.2.2 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." [18] 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 files we want to, avoiding the cost of copying the file to a higher layer.

- 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. This is actually already done by the docker agent of INGInious.

- 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.

3.2.3 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). [15] 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. [10]

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 intrerest-ing 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 (§3.2.2).

3.2.4 Firecracker

Firecracker is a new VMM (hypervisor) created specifically for serverless and containers applications. [8] 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 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 classical containerisation regarding security.

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

3.2.5 Summary

In this section, on Table 3.1, are quickly reminded the several new solutions we explored in this chapter, along with some interesting infos about them.

Name	**Pub.**	**Update**	**Open-Source**	**Com.**	**Isol.**
SAND [9]	2018	?	?	No	Cont.
SOCK [18]	2018	?	?	No	Cont. or Runt.
LigthVM [15]	2017	2017	Yes	No	Virt.
Firecracker [8]	2019	2020	Yes	Yes	Virt.

Table 3.1: Summary table of the different solutions explored in this chapter.

Caption:

- *Name*: The name of the project.
- *Pub.*: The first publication year of the project.
- *Update*: The last update year of the project.
- *Open-source*: If the project is open-source.
- *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.

3.3 My master thesis

In this chapter I presented an overview of the work of other people in the field of containerization, with a focus on performance improvement or measurement. Though, we have more things to test, and more questions to ask than what those papers had to offer.

First, most of those performance measurement are quite old, and none of them considered all of the solutions that we could use for INGInious case. For example Podman is never mentionned once (which is normal as version 1.0.0 of Podman was only released in January 2019). Neither is the interesting case of Kata-container. The apparition of lightweight hypervisor like Firecracker is a game changer, but the paper didn't provide any performance comparision with containerization solutions.

Second, none of the research here had a real focus on the time required for each solution to provide a ready to run environnement. We have got a lot a CPU and memory overhead analysis, but those are not our main concerns.

This is the direction I will take in this report. Trying to respond to the question: "What would be the most appropriate containerisation (or virtualisation) solution for INGInious use case?", or in other word, how to get the shortest instantiation time, along with the best theoretical isolation possible and as low overall performance overhead as we can get.

Chapter 4

Benchmark tool

As said previously, the goal of this thesis will be to identify if some improvements could be made to INGInious, by changing the containerization solution in use. In order to do this, I have created some tests, and listed different possible configuration that should face those test. Based on the results to those tests, I should be able to determine whether or not we can bring improvement to INGInious.

4.1 Setup

4.1.1 Testing environment

To be fair in the result comparison, all the different configurations have to be tested with as much in common as possible. This will allow to determine the influence of varying parameters in those configurations. To do so, the same machine will be used for all the tests, with its configuration beeing updated accordingly to the requirements of each solution.

The testing environment used for the final results presented in this work is an an old laptop, refurbished with the following configuration:

processor	Intel(R) Core(TM) i5-2410M CPU @ 2.30GHz (x4)
memory	8GB
storage	256GB SSD
operation system	Ubuntu 18.04.4 LTS
linux kernel version	4.15.0-101-generic

The choice of going with a baremetal setup wasn't the original one. But the early results I got showed that some solutions (especially the ones using virtualization) did perform way worse when running inside of a VM. To get the best out of them, the setup has then been moved out of VMs.

Note about cpu: The cpu is a little bit weak, and not representative of what a real production server would use to host a container workload. Unfortunetly this was all I had available, and it will have to do. This means that the results presented in this work can still be a little bit less performant than what we could get typically in the cloud.

Note about memory: This is also typically less than what we would have in the cloud, but this is just enough for our case, as we only execute one container at a time, with limited access to memory.

Note about storage: The storage type is important, as it will highly influence the performance of my tests, given the high solicitation of I/O they require. Using an SSD is therefore essential, both to avoid the need to run the tests for several weeks, and to have measurements more in pair with what cloud provider actually propose.

Note about operating system: The main reason I choosed Ubuntu is because I am familiar with it, and it made things easier for me to setup. Also, Ubuntu is often (if not always) an option that cloud providers offer when it comes to choose the OS to install in a VM.

4.1.2 Configuration of the environment

As a lot of different solutions are going to be tested, and all of them with their own requirements (sometimes conflicting with the ones of other solutions), some work has to be done to make it easier for someone to replicate them. The configuration of the environnment is going to be done with Ansible[1] playbooks, with one playbook for each solution to test.

All the different configurations are presented in section 4.2.1. The different playbooks can be found in the repository of this project.

4.2 Tests

As we alreay said previously, time is an important aspect in the experience INGInious provides to its users. The time for a task to be evaluated and the time before this task is evaluated. In order to improve those two, we will focus in those tests on the booting time of containers and their IO handling (still related to the time overhead that some solution could have compared to another).

[1] https://docs.ansible.com/

While we are at it, we will also verify if some solutions handle network much poorly than others. Each test done is presented in more details in section 4.2.2.

4.2.1 Candidates

A candidate solution is a combinaison of different elements, variabilities, choices to make. We will consider here those ones:

- **Container manager**: *Docker*, *Podman*, *LXD*, those are the different user friendly solution that can be used to manage container, and that we will compare here.
- **Container runtime**: *runc*, *crun*, *LXC*, *Kata containers* (Qemu or Firecracker), the (less user-friendly) container runtimes, taking care of all the isolation that a container require.
- **Control group**: *cgroup* and *cgroupv2*, there is no real choice to make here, cgroupv2 is the successor of cgroup, and should be used, when supported by the other elements of the configuration.
- **Storage driver**: *aufs*, *btrfs*, *devicemapper*, *directory*, *overlay*, *vfs*, *zfs*, *lvm*, those are the different strategies that can be used to manage the file system of the container.
- **Base container image**: *Alpine* because it is the default choice when it comes to conceive containerized applications today or *Centos* because it is the current choice made by INGInious.
- **Rootless container**: Whether if *yes* or *no*, we run the container in rootless mode.

Though, we can not compose a candidate solution with one element of each category, randomly picked. Some element of the solution might only be usable with some of the possibilities for a variability. The different constraints that we have are listed here:

- Docker doesn't support cgroupv2 yet (actually the support has to be added by containerd).
- Docker only supports aufs (deprecated), btrfs, devicemapper (until Docker 18.06), overlay, vfs, zfs as storage driver.
- LXD doesn't support cgroupv2 yet.
- LXD only supports LXC as runtime, and LXC is only supported by LXD.

– LXD only supports btrfs, zfs, directory (dir) and lvm as storage driver.

– Kata Container with Firecracker only supports devicemapper as storage driver.

– zfs, lvm, aufs can not be used with rootless containers.

– cgroup (v1) can not be used with rootless containers.

– runc doesn't support cgroupv2 yet.

The list of different candidate solution we can compare is presented in the table below. Each container will be launched with a CPU limitation of one single core, and memory limitation of 1GB. This is more than enough for each test that will be done. A network interface will be only added to the container if it needs it for the specific test.

#	**Manager**	**Image**	**Storage**	**Cgroup**	**Runtime**	**Rootless**
1	Docker	Alpine	aufs	v1	runc	No
2	Docker	Alpine	btrfs	v1	runc	No
3	Docker	Alpine	devicemapper	v1	runc	No
4	Docker	Alpine	overlay2	v1	runc	No
5	Docker	Alpine	vfs	v1	runc	No
6	Docker	Alpine	zfs	v1	runc	No
7	Docker	Alpine	aufs	v1	crun	No
8	Docker	Alpine	btrfs	v1	crun	No
9	Docker	Alpine	devicemapper	v1	crun	No
10	Docker	Alpine	overlay2	v1	crun	No
11	Docker	Alpine	vfs	v1	crun	No
12	Docker	Alpine	zfs	v1	crun	No
13	Docker	Alpine	aufs	v1	kata-runtime[2]	No
14	Docker	Alpine	btrfs	v1	kata-runtime	No
15	Docker	Alpine	devicemapper	v1	kata-runtime	No
16	Docker	Alpine	overlay2	v1	kata-runtime	No
17	Docker	Alpine	vfs	v1	kata-runtime	No
18	Docker	Alpine	zfs	v1	kata-runtime	No
19	Docker	Alpine	devicemapper	v1	kata-fc[3]	No

[2]Default Kata Containers runtime, using qemu as hypervisor.
[3]Kata Containers runtime, using Firecracker as hypervisor

20	Docker	Centos	aufs	v1	runc	No
21	Docker	Centos	btrfs	v1	runc	No
22	Docker	Centos	devicemapper	v1	runc	No
23	Docker	Centos	overlay2	v1	runc	No
24	Docker	Centos	vfs	v1	runc	No
25	Docker	Centos	zfs	v1	runc	No
26	Docker	Centos	aufs	v1	crun	No
27	Docker	Centos	btrfs	v1	crun	No
28	Docker	Centos	devicemapper	v1	crun	No
29	Docker	Centos	overlay2	v1	crun	No
30	Docker	Centos	vfs	v1	crun	No
31	Docker	Centos	zfs	v1	crun	No
32	Docker	Centos	aufs	v1	kata-runtime	No
33	Docker	Centos	btrfs	v1	kata-runtime	No
34	Docker	Centos	devicemapper	v1	kata-runtime	No
35	Docker	Centos	overlay2	v1	kata-runtime	No
36	Docker	Centos	vfs	v1	kata-runtime	No
37	Docker	Centos	zfs	v1	kata-runtime	No
38	Docker	Centos	devicemapper	v1	kata-fc	No
39	LXD	Alpine	btrfs	v1	LXC	No
40	LXD	Alpine	zfs	v1	LXC	No
41	LXD	Alpine	directory	v1	LXC	No
42	LXD	Alpine	lvm	v1	LXC	No
43	LXD	Centos	btrfs	v1	LXC	No
44	LXD	Centos	zfs	v1	LXC	No
45	LXD	Centos	directory	v1	LXC	No
46	LXD	Centos	lvm	v1	LXC	No
47	Podman	Alpine	aufs	v1	runc	No
48	Podman	Alpine	btrfs	v1	runc	No
49	Podman	Alpine	overlay	v1	runc	No
50	Podman	Alpine	vfs	v1	runc	No
51	Podman	Alpine	zfs	v1	runc	No
52	Podman	Alpine	aufs	v2	crun	No
53	Podman	Alpine	btrfs	v2	crun	No
54	Podman	Alpine	overlay	v2	crun	No
55	Podman	Alpine	vfs	v2	crun	No
56	Podman	Alpine	zfs	v2	crun	No
57	Podman	Alpine	btrfs	v2	crun	Yes

58	Podman	Alpine	overlay	v2	crun	Yes
59	Podman	Alpine	vfs	v2	crun	Yes
60	Podman	Centos	aufs	v1	runc	No
61	Podman	Centos	btrfs	v1	runc	No
62	Podman	Centos	overlay	v1	runc	No
63	Podman	Centos	vfs	v1	runc	No
64	Podman	Centos	zfs	v1	runc	No
65	Podman	Centos	aufs	v2	crun	No
66	Podman	Centos	btrfs	v2	crun	No
67	Podman	Centos	overlay	v2	crun	No
68	Podman	Centos	vfs	v2	crun	No
69	Podman	Centos	zfs	v2	crun	No
70	Podman	Centos	btrfs	v2	crun	Yes
71	Podman	Centos	overlay	v2	crun	Yes
72	Podman	Centos	vfs	v2	crun	Yes

4.2.2 Experiments

For all the experiments presented here, time measurements are taken at four important steps of the execution of the container:

t_0 Before its creation, this is time zero.

t_c After its creation. Container's state is *created*.

t_s After its startup. Container's state is *running*.

t_e After the end of the execution of the command. Container's state is *running*.

This gives us three interesting measurements:

1. Creation time: $t_{create} = t_c - t_0$
2. Starting time: $t_{start} = t_s - t_c$
3. Execution time: $t_{exec} = t_e - t_s$

The command available to interact with each step of the lifecycle of the containers for each solution can be found in Appendix A, along with some more explaination about what is done in each step.

In order to do this, each container will be running as first process a shell[4], which will never receive any input, but is here to ensure that the container will stay up

[4]except for LXD, which natively has an init process running in each container

as long as we need it. When we don't need the container anymore, we simply kill all the processes in it.

Booting time

This tests aims at determining the candidate solution that can provide a ready to run container the fastest. In order to measure this, a simple container composed of only the base image is created, in which the simple command `/bin/echo Hello World` is executed. The best candidate would be the one that has the shortest creation and starting time ($t_{create} + t_{start}$). The execution time (t_{exec}) is also a good illustration of the overhead caused by all the steps taken before actually starting the process in the container.

Basic I/O performances

Those tests aim at determining mostly the influence of the usage of each storage driver on the performances of an application that relies highly on it. Four different tests are done here: big file read and write, and great amount of file read and write. The best candidate solution would be the one to with the shortest creation, starting and execution time ($t_{create} + t_{start} + t_{exec}$).

For the first one, five sqlite databases have been created, of five different size (`151.6 kB`, `536.6 kB`, `2.6 MB`, `11.9 MB` and `111.6 MB`, about one order of magnitude appart from each other). We use an sqlite database as everything will be stored in one file, and no deamon needs to be runnning, which simplify the set up of the tests. Plus this is a more realistic workload than just reading/writing one gigantic file. The read test will do a select operation on each table of the database without any filter, displaying all of the table content. The write test will duplicate each table in the database. The database chosen as basis to generate all the presented one is `tpcc`[5], which is a benchmark database, perfect for our case. The biggest database used is an export from the tpcc database, the other ones are created by removing content of some tables from this same database.

For the second one, five loads of small file have been created (10, 100, 1000, 10 000 and 100 000 files). The content of the file is alphanumeric, and completely random (no file should be a duplication of another one), each file is 4096 character (and bytes) long. This represent the best the usual kind of file a user could access, to read or write content from/to it. The read test will load the content of each file in memory, then discard it and move to the next one. The write test will extract an uncompressed archive (`tar`) containing all those files.

[5]ttps://relational.fit.cvut.cz/dataset/TPCC

Network setup time

This tests aims also at determining the candidate solution that can setup a ready to run container the fastest, but with the added complexity of adding a network interface, and mapped ports, so that the host machine can send requests to an http server running inside of the container. I choosed a `lighttpd` server, for its lightness, and its wide use in container application. We take one more time measurement here, the time at which we received the first response at the http requests we send to our server (t_r). The best candidate would be the one that has the shortest delay before this response ($t_{create} + t_{start} + t_{exec} + t_{response}$, with $t_{response} = t_r - t_e$). The execution time corresponds to the execution of the command that launches the server as detached process.

Ping response time

This really simple test aims at determining if some solution provide a much weaker network solution to the container than the others. We simply perform a ping request to `1.1.1.1`, and compare the response time we get for each solution. The best solution would be the one with the shortest ping response time.

Chapter 5

Results interpretation

Based on the results of the different experiments I made, I will try to answer in this chapter to three main questions:

1. Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform? How much better could it be? How easy would it be to improve it?
2. Could there be a solution tailor-made for the specific case of INGInious? What would it be? What would it take to use it?
3. What would be the cost of providing a stronger/safer isolation to the containers used by INGInious? What opportunities could it bring?

5.1 Current INGInious situation

We will here consider the first question:

"*Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform? How much better could it be? How easy would it be to improve it?*"

The current configuration of INGInious is the following:

Container manager	Docker
Base image	Centos
Storage driver	overlay2
Container runtime	runc
Control group version	v1
Rootless containers	no

This configuration is quite decent, and gives good performances, their is mainly one change that can improve those. But let's analyse this step by step.

Storage driver

Overlay2 (referenced later as simply overlay) is the storage driver that Docker recommands to use by default (when OverlayFS is supported on the host). Its layer mechanism allows to limit the redundancy of information when you use the same base image to create different images. And its file-based copy-on-write strategy gives overal good performances.

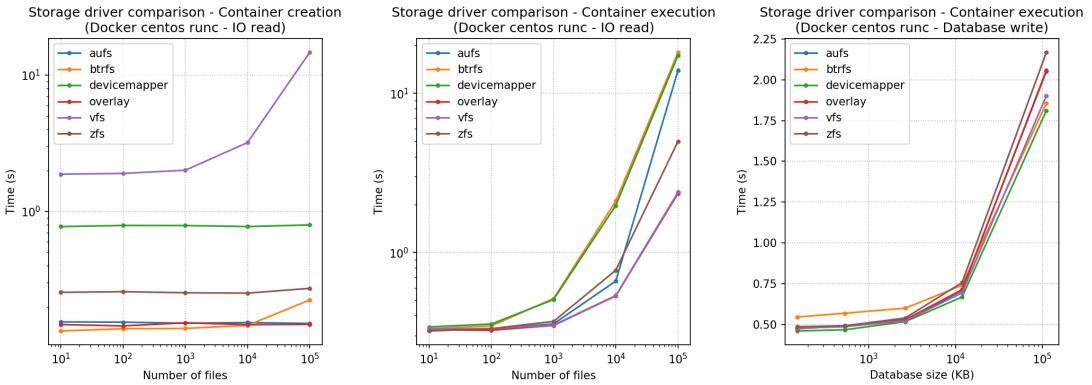

Figure 5.1: Storage driver performance comparison for Centos containers, launched with Docker and runc

On Figure 5.1, we can see how much it cost to have a full-copy mechanism like vfs for each container creation. We can also see that overlay, aufs and btrfs offers similar performances for the creation of containers. The main differences between those three is shown by their performance when solicitating I/O.

- Aufs, even though performing similarly to overlay in all the other cases (not shown here) and relying on the same union file system mechanism as overlay, gets far behind the latest when a lot of read operations are done. This most likely comes from one big difference in the implementation of those two union file system: when a file is opened with OverlayFS, all the operations on it are directly managed by the underlying file systems. This simplify the implementation, and improve the performances quite a lot as we can see on the second graph of the figure. (For this test, for each `openat` system call, four `read` system call are done)

- On the last graph of the figure (on the right), we can see the advantage of using block-based copy-on-write, when modifying big files. Indeed, when doing so, where overlay and aufs will need to copy the whole file before modifying it, the other solutions will only need to copy the file system blocks

needing to be modified (or not even copy the file in the case of vfs). We can see that devicemapper handles this a little bit better than btrfs, but given the much higher container creation cost of the first one, it will most likely be more intersting to use btrfs in most cases.

- One more thing to pay attention to, is that is seems that storage drivers with block-based copy-on-write mechanism, face some difficutlies when creating container with a large amount of files in it. This might be harder to see here for devicemapper and zfs, but it is the case. Zfs seems to also suffer from big files in the container filesystem, for the container creation.

One more reason to justify the use of overlay is that in most container use cases, read operations are the most important ones. Normally, no large amount of data should be written in the container file systems, volumes (external storage from the host, mounted into the container file system) should specifically be used for that purpose as it provide nearly native writing performances and the data persists, even after the end of the container life cycle.

Base image

Alpine is quite popular in containers file systems. Thanks to its minimalist default configuration, its image is really light compare to the ones of more complete file systems. The Alpine base image provided by Docker is only 5.61MB, while Centos's one is 203MB. But the size taken by images on disk isn't really something we worry about in our case.

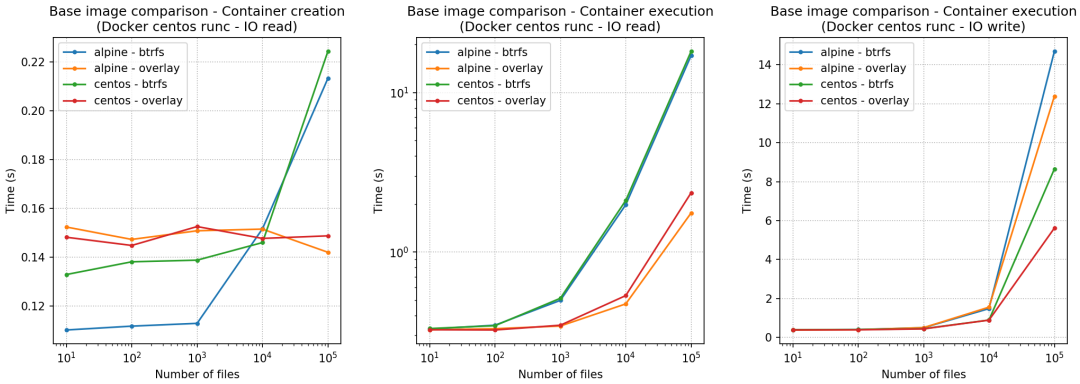

Figure 5.2: Base image performance comparison for containers launched with Docker and runc

From the graphs of Figure 5.2, we can notice several things:

- We can see on the first graph (on the left) that, once more, btrfs offers better creation performances when the number of files is smaller. Therefore, being a more complete linux distribution, with more functionnalities, and so, more files, Centos-based images are more costly for container creation.
- On the second graph, it also appears that simple read operations are a little bit faster when using Alpine as base image. This trend has also been observed with bigger file read, with the *Database read* test.
- One thing really interesting that we can see on last graph however, is that the trend seems to be inverted for write operations. This difference is really likely to be caused by the difference in the implementation of `tar` (used for this test) in each distribution's package repository. Indeed, the implementation coming from Alpine makes a lot more system calls (about three times more) than Centos's one.

The choice of Centos as base image can then be justified by the maturity of such distribution. It had been around for a while, it is widely used outside of container applications, and is more likely to count optimization in the different tools at disposal. If we don't require those tooles though, the minimalist and ligthweight aspect of Alpine makes it a better choice.

Container runtime

On Figure 5.3 is chown the influence of different container runtime solutions on the different execution step of the container.

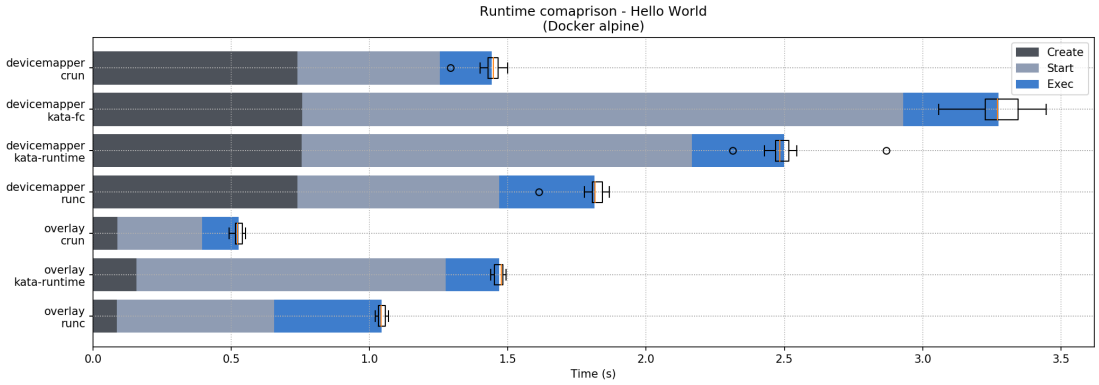

Figure 5.3: Runtime performance comparison for Alpine containers, launched with Docker

By first comparing crun to runc, we see how much of a difference it makes to use an efficient C implementation, over a less efficient Go one. The difference is much more obvious for the `start` and `exec` phases as those are the steps where real low-level operations are made by the container runtime (entering namespaces, setting cgroups, forking processes).

Then we have the Kata Containers solutions, based on virtualization. It obviously will lead to a greater overhead for creating and starting containers, as a whole kernel as to be loaded. However, the `exec` phase seem to have a much lower overhead. We can't miss how much worse is `kata-fc` (using Firecracker hyperviser) compared to `kata-runtime`, and it might be disapointing given the recent popularity that has embrassed Firecracker. In response to that here are some important elements:

1. Firecracker has not been conceived to run under Kata Containers's hood. It is really possible that it would perform better when running standalone.

2. Some advantages that Firecracker has and that are not obvious in our study case are, first, the memory footprint, which is way smaller than with Qemu, and second, the reduced code base, which ensure a much lower attack surface. Those are also reason why Firecracker is so popular.

3. After some discussion with Kata Containers community[1], it turned out that Kata Containers make use of one feature of Qemu that Firecracker doesn't have, that would justify this difference in performance: vNVDIMM. Thanks to vNVDIMM, the root file system of the guest VM is fully loaded in memory and directly accessed in it, which is faster than reading it from disk as Firecracker does.

Container manager

On Figure 5.4 are shown the different container manager considered in the experiements, in their most performant configuration. Docker and Podman using crun as runtime, and overlay or btrfs as storage driver. LXD uses lxc as runtime and btrfs as storage driver.

We can see that Docker is still the best solutions for us, even though none of the other solution presents really bad performances. Given the relatively young age of Podman, its focus might not be yet fully on performances, but rather on functionnalities. We can then hope for improvements on this side as time goes, it would be worth check on them later this year, or the next one. Some interesting things to note regarding Podman's rootless solution:

[1]The full discussion can be found at this link: https://github.com/kata-containers/runtime/issues/2642

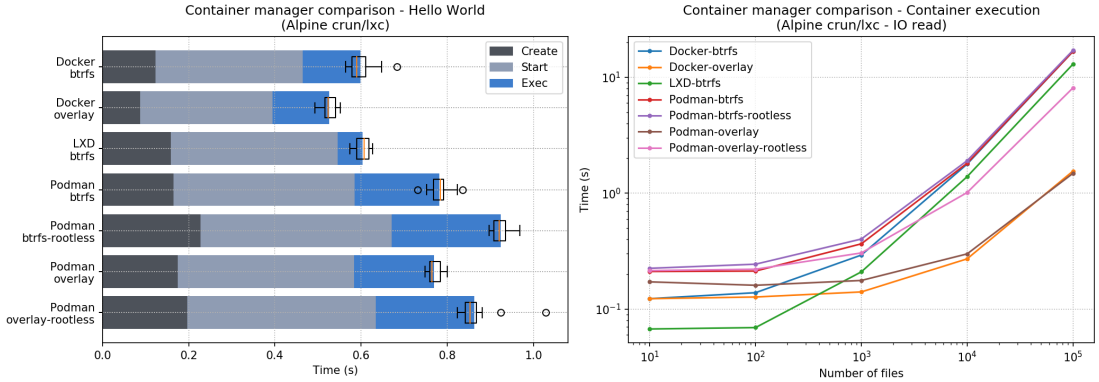

Figure 5.4: Container manager performance comparison for Alpine containers

- There is a cost to create rootless containers with Podman compared to going rootfull with the same container manager. This is very likely caused by the extra steps required to be taken to launch rootless container with all the functionnalities of a rootfull one. Podman needs first to unshare user and mount namespaces, to be able to make bind mounts on the container filesystem. It also requires to add a cgroup scope to all the commands that setup the containers, to be sure that the container runtime will later be able to move process created to the cgroup attached to the container.

- Because of the lack of *real* root permissions on the machine, rootless containers can not use OverlayFS, instead they use a fuse implementation of the latest, which, as we can see on the right-most graph, induces great performances cost. It get even worse when doing write operations.

Final configuration

Based on the previous observations, the ideal configuration would then be:

Container manager	Docker
Base image	Alpine/Centos
Storage driver	overlay2
Container runtime	crun
Control group version	v1
Rootless containers	no

As we can see, the final configuration is not that much different from the original one. The quick answer to the original questions would be:

Compared to other available solutions, how good is the current configuration chosen by INGInious to face the responsiveness challenge of the platform? The current configuration is good. It could be improved, but is definitely not the worse one. The choice of going with Docker was the most soundfull when INGInious was created, and it is still the case now. The current storage driver offers great performances for almost every cases, only some specific case, sollicitating a lot of IO operations on big files, can give an advantage to another storage driver, btrfs. The choice of a Centos base image is not bad either, but the size of the image being greater than Alpine's one, except for the situations where Centos offered better write performances, using Alpine seems to be better. The move here might then be to go for an hybrid solution, using Centos only in situations where its small writing performance advantage become meaningfull.

How much better could it be? As we have seen, changing the current container runtime makes a significant difference. The c implementation of `crun` is claimed to be twice as fast as the go implementation of `runc` by `crun`'s contributors. And we can definitely see the difference. The change of base Image though doesn't show as obvious improvement.

How easy would it be to improve it? This is the real good news, it truely is super easy to apply the most significant change. You only need to install `crun`, reconfigure Docker to use it by default, and you are good to go, no change as to be done to INGInious! For the base of the base image the story is different though, as it would require to change all the containers images, which is a lot of work.

Chapter 6

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. The storage will be lost after this phase.

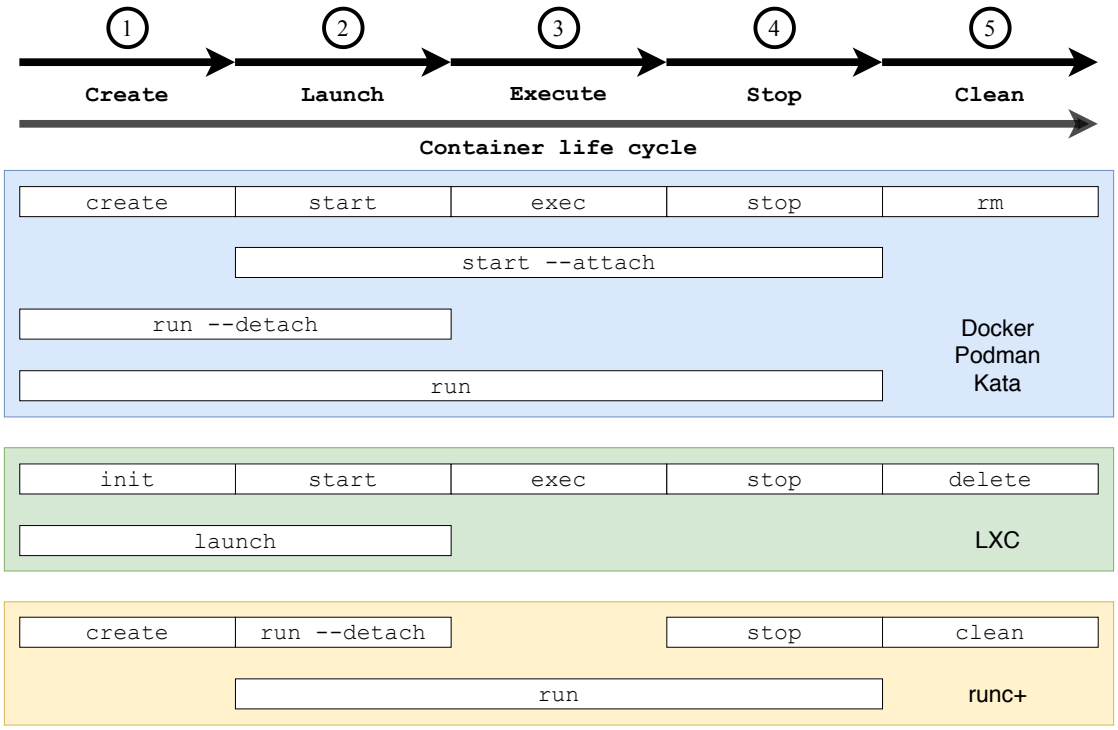

Figure A.1: Container life cycle and how to interact with it.

Bibliography

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

[2] Linux vserver. `http://linux-vserver.org/Paper`. Accessed: 2020-04-25.

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

[4] Openvz. `https://openvz.org`. Accessed: 2020-04-25.

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

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[13] Zhanibek Kozhirbayev and Richard O Sinnott. A performance comparison of container-based technologies for the cloud. *Future Generation Computer Systems*, 68:175–182, 2017.

[14] Zheng Li, Maria Kihl, Qinghua Lu, and Jens A Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, pages 955–962. IEEE, 2017.

[15] 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.

[16] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

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

[18] 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.

[19] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[20] Randolph Y Wang and Thomas E Anderson. xfs: A wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*, pages 71–78. IEEE, 1993.

[21] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

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