



END OF STUDIES MASTER'S PROJECT

MASTER'S THESIS

Simulation of a Kubernetes Cluster with Validation in Real Conditions

Author

Théo LARUE

Evaluator

Surname NAME

FEBRUARY 24TH TO AUGUST 7TH 2020

Contents

1	Introduction / Abstract	2
2	Kubernetes and the scheduling problem	3
2.1	Kubernetes overview	3
2.2	HPC and Kubernetes	5
2.3	The scheduling problem	6
2.4	Simulating infrastructures	6
2.4.1	HPC simulators	6
2.4.2	Kubernetes simulation	6
2.5	The Batsim infrastucture simulator	7
2.5.1	Batsim concepts	7
2.5.2	Limitations	7
3	Problematic	8
3.1	Objectives	8
3.2	Translation	8
3.3	Synchronization	8
4	Implementation	9
4.1	Batkube architecture	9
4.2	API implementation	9
4.3	Time hijack	9
4.3.1	batsky-go	10
5	Evaluation	11
6	Conclusion	12

Introduction / Abstract

TODO

Kubernetes and the scheduling problem

2.1 Kubernetes overview

Cloud Native Computing

In the early stages of application development, organizations used to run their services on physical servers. With this direct approach came a few problematics: resources allocation, maintainability, scalability for exemple. Developers then went on with virtualized machines to run their services regardless of physical infrastructure, which then led to the concept of containers.

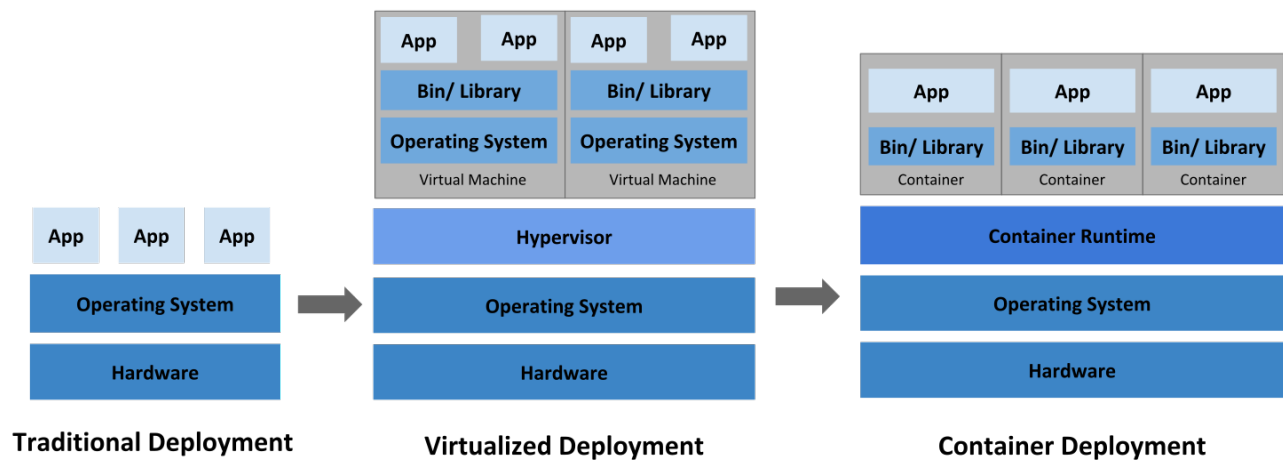


Figure 2.1: Evolution of application deployment.

Source: <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

Containers can be thought of as lightweight virtual machines. Unlike virtual machines, containers share the same kernel with the host machine but still allow for a very controlled environment to run applications. There are many benefits to this : separating the development from deployment, portability, easy resource allocation, breaking large services into smaller micro-services or support of continuous integration tools (containers greatly facilitate integration tests), to cite a few.

The Cloud Native Computing Foundation¹ was founded in the intent of leveraging the container technology for a overall better web. This is the definition of Cloud Native Computing according to the CNCF :

¹<https://www.cncf.io/>

“Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.”²

Kubernetes³ is the implementation of this general idea and was announced at the same time as the CNCF. It aims at automating of the process of deploying, maintaining and scaling containerized applications. It is industry grade and is now the de-facto solution for container orchestration.

Kubernetes concepts

The basic processing unit of Kubernetes is called a **pod**. A pod is composed of one or several containers and volumes (A volume is some storage space on the host machine that can be linked to containers, so they can read persistent information or store data in the long term). In the cloud native context a pod can be thought of as a service or micro-service, but we will see them a little bit differently.

Pods are bundled together in **nodes** (figure 2.2), which are most of the time physical machines (but might as well be virtual machines). They represent another layer to be passed through to access the outside world which can be useful to add some layers of security or facilitate communication between pods, for example. Each node runs at least one pod and also one kubelet which is a process responsible for communicating with the rest of Kubernetes (or more precisely, with the master node which in turns communicates with the api server). A set of nodes is called a **cluster**. Each Kubernetes instance is responsible for running one cluster.

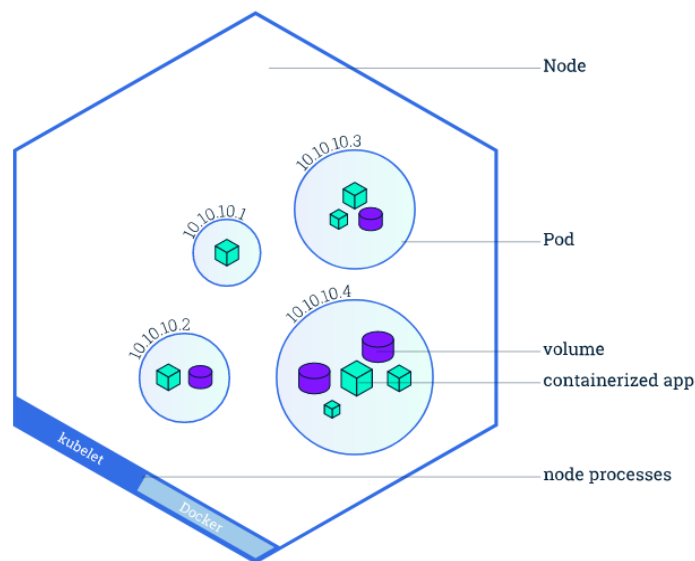


Figure 2.2: Node overview

Source: <https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/>

²<https://github.com/cncf/toc/blob/master/DEFINITION.md>

³<https://kubernetes.io/>

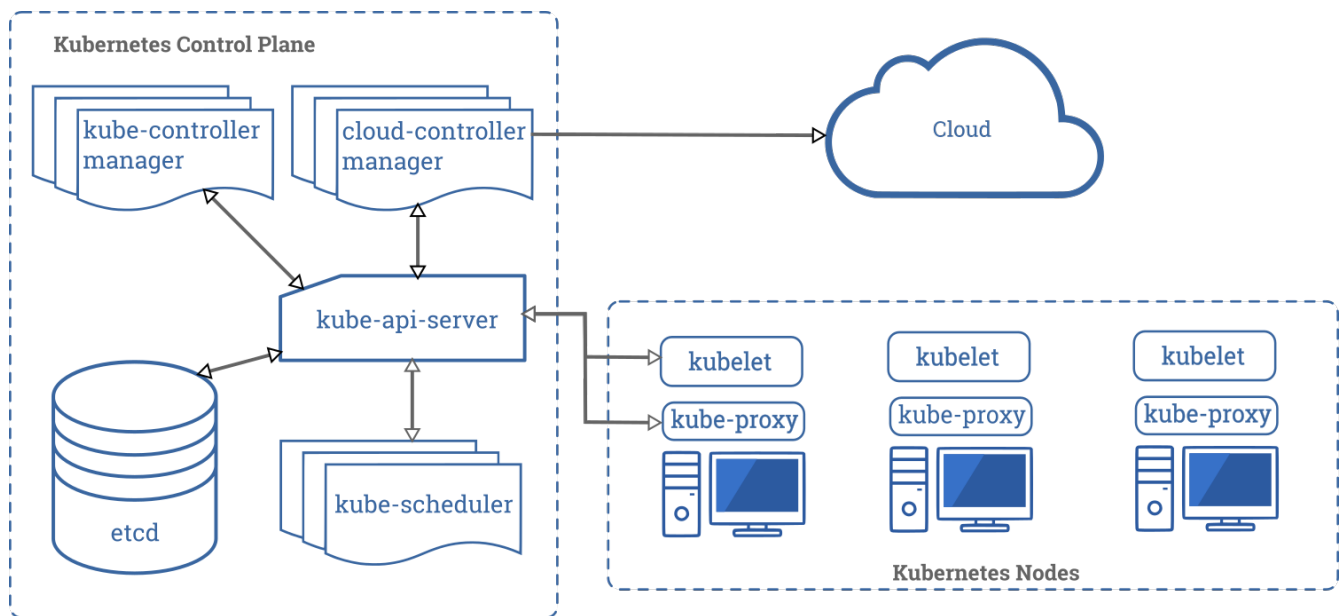


Figure 2.3: Components of Kubernetes

Source: <https://kubernetes.io/docs/concepts/overview/components/>

Kubernetes revolves around a central component, its api server (figure 2.3). The majority of every operations go through this api server like user interactions through `kubectl` or scheduling operations.

2.2 HPC and Kubernetes

The difference between HPC (High Performance Computing) and Kubernetes lies in the workloads they are intended to tackle.

A general definition of HPC - which can also be applied to HPDA - would be : “*High Performance Computing (HPC) most generally refers to the practice of aggregating computing power in a way that delivers much higher performance than one could get out of a typical desktop computer in order to solve large problems in science, engineering, or business.*”⁴. HPC can either refer to “High Performance Computing” or “High Performance Computer” but it is generally clear which one it refers to, given the context.

A HPC workload is composed of numerous tasks hungry for computational resources which are executed in parallel on different machines (that we can refer to as compute resources or nodes). These tasks may be completely independant like when different users each submit a single task, or they may also be tightly coupled together as when a single user submits a job that is composed of several tasks than can be run in parallel. In that case, the whole system becomes very sensitive to latency as these tasks have got to communicate together. This is done through MPIs (Message Passing Interfaces) which represent a large part of the HPC field.

Kubernetes was designed for cloud native applications. Services or micro services are run in containers and are expected to be available at all times : it ensures that containers are restarted

⁴https://www.eni.lu/university/high_performance_computing

when a failure occurs. It also provides portability, monitoring and easier administration. It was then developed with this objective in mind which hardly complies with the challenges of HPC.

However, there are many issues with the aging HPC field that can be resolved with a container driven approach. Kubernetes is now the standard for AI and Machine Learning as shown by the many efforts at making it an efficient environment[3][6][4], and many believe it represents the future of HPC aswell.

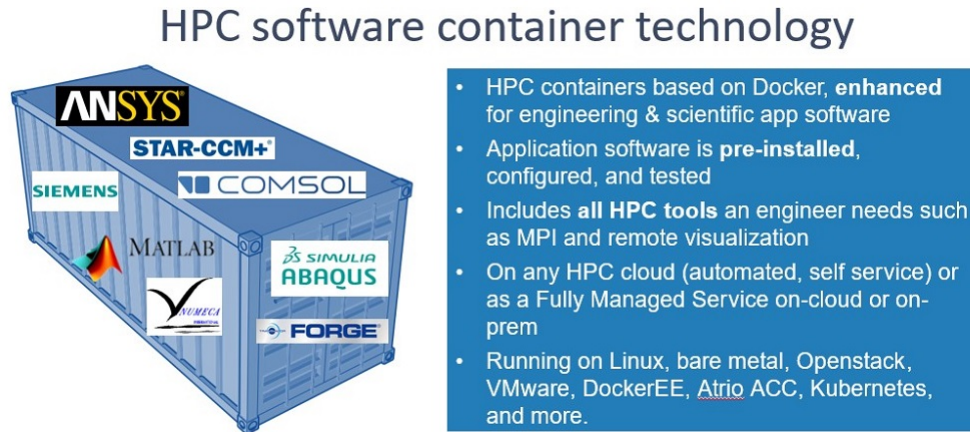


Figure 2.4: The container technology for HPC

Source: <https://www.hpcwire.com/2019/09/19/kubernetes-containers-and-hpc/>

TODO

2.3 The scheduling problem

2.4 Simulating infrastructures

2.4.1 HPC simulators

2.4.2 Kubernetes simulation

This raises the question of scheduler development. Developing a scheduler implies being able to test its performances throughout the development process, however, testing in real conditions is time consuming and expensive. Organizations can either have enough resources to cover these costs, or test their scheduler against a simulation.

Kubernetes cluster simulations is an open problem and is the subject of this master project. Our approach relies on the Batsim[1] infrastructure simulator, which is itself built upon Sim-grid[5]. Batsim is currently mostly used to simulate HPC infrastructures but was designed to be able to simulate any kind of infrastructure and therefore is theoretically able to simulate any Kubernetes cluster, moreover, Kubernetes was designed to run services but is capable of handling High-Performance Computing[2]. This project aims at adapting Batsim so it can evaluate Kubernetes schedulers.

2.5 The Batsim infrasctucture simulator

2.5.1 Batsim concepts

2.5.2 Limitations

Problematic

3.1 Objectives

The goal of this project is to design and implement Batkube, which will be an interface between Batsim and Kubernetes schedulers. With this interface, we want to compare Batsim results against data from a real Kubernetes cluster, given HPC workloads.

3.2 Translation

3.3 Synchronization

Implementation

4.1 Batkube architecture

TODO

4.2 API implementation

4.3 Time hijack

TODO

4.3.1 batsky-go

Algorithm 1: Requester loop

Input: req: request channel, res: result channel map

```
1 while Batkube is not ready do
2   | wait
3 requests = []request
4 while req is not empty do
5   | m = <- req /* Non blocking receive */
6   | requests = append(requests, m)
7 sendToBatkube(requests) /* Only requests with duration > 0 are actually sent.
   Batkube will always answer. */
8 now = responseFromBatkube()
9 for m in range requests do
10  | res[m.id] <-now /* The caller continues execution upon reception */
```

Algorithm 2: Time request (time.now())

Result: Current simulation time

Input: d: timer duration, req: request channel, res: response channel map

Output: now : simulation time

```
1 if requester loop is not running then
2   | go runRequesterLoop() /* There can on ly be one loop runing at a time */
3 id = newUUID()
4 m = newRequestMessage(d, id) /* Requests are identified using uuids */
5 resChannel = newChannel()
6 res[id] = resChannel /* A channel is associated with each request */
7 req <- m /* The code blocks here until request is handled */
8 now = <-resChannel /* The code blocks here until response is sent by the
   requester loop */
9 return now
```

Evaluation

Conclusion

Bibliography

- [1] *Batsim docs*. URL: <https://batsim.readthedocs.io/en/latest/>.
- [2] *Kubernetes Meets High-Performance Computing*. URL: <https://kubernetes.io/blog/2017/08/kubernetes-meets-high-performance/> (visited on 04/21/2020).
- [3] Mikyoung Lee, Sungho Shin, and Sa-Kwang Song. “Design on distributed deep learning platform with big data”. In: (2017).
- [4] Seetharami R. Seelam and Yubo Li. “Orchestrating Deep Learning Workloads on Distributed Infrastructure”. In: *Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning*. DIDL '17. Las Vegas, Nevada: Association for Computing Machinery, 2017, 9–10. ISBN: 9781450351690. DOI: 10.1145/3154842.3154845. URL: <https://doi.org/10.1145/3154842.3154845>.
- [5] *Simgrid official website*. URL: <https://simgrid.frama.io/>.
- [6] Boris Tvaroska. “Deep Learning Lifecycle Management with Kubernetes, REST, and Python”. In: Santa Clara, CA: USENIX Association, May 2019.