Master of Science in Informatics at Grenoble
Master Informatique
Specialization MoSIG

Development and evaluation of a Kubernetes cluster simulator based on Batsim

LARUE Théo

Defense Date, 2020

Research project performed at Laboratoire d'Informatique de Grenoble

Under the supervision of:
Michael Mercier

Defended before a jury composed of:
Céline Coutrix
Olivier Richard
Vania Marangozova-Martin

September 2020

Abstract

Containers, which are lightweight virtual machines sharing the same kernel as their host machine, have given developers much more control over their resources than they had before when there was no intermediate between the application and the physical servers. Eventually, developers realized they could go even further by automating container management operations and in this context we witnessed the rise of container orchestration software. Kubernetes is one such software and arguably the most popular container orchestrator on the market. However, while Kubernetes has become a standard, many fundamental questions remain. One such issue is the problem of scheduling, which in computer science is the action of allocating tasks submitted by users on available resources. In order to properly evaluate and develop schedulers, researchers use simulators to model complex networks and computing infrastructures accurately in a short amount of time. Indeed, the cost of running experiments on real systems will never be justified by the improvements these experiments would enable. While the default Kubernetes scheduler works well for most of the *Cloud Native* infrastructures, researchers would rather experiment with different batch processing policies on Kubernetes clusters. Our goal in this master thesis is to describe Batkube, an open source software enabling the study of Kubernetes schedulers by simulating Kubernetes clusters. Batkube is based on Batsim, which is itself a general purpose distributed system simulator focused on the study of *Resource and Jobs Management Systems (RJMS)* and built on the popular SimGrid framework, enabling scalable and accurate simulations of grid systems. During this work we implemented a custom Kubernetes API to form an adaptive layer between Batsim and the schedulers in order to run simulations as if the scheduler was running on a real system. Time synchronization between the simulator and the scheduler being key to the simulation, we developed a tool to patch any scheduler written in the Go programming language allowing us to intercept their time and synchronize it with the scheduler. Using these tools we were able to run simulations of Kubernetes clusters and conduct a study on the time synchronization parameters. Our API currently supports the default Kubernetes scheduler and is only able to run simple scenarios, but it offers great perspective for future development to improve it into a fully fledged Kubernetes cluster simulator.

Acknowledgements

I would like to express my gratitude to Michael Mercier, my supervisor, and Olivier Richard who also supervised this work for their invaluable implication during the whole project. I sincerely thank them for giving me a guideline, for changing my perspective when I was going the wrong way, for their contribution to this work, for their sympathy and for their precious advices throughout these months. I thank Adrien Faure for giving me a first review on my work and helping me with my graphs that would not be as readable and space efficient without him, Pedro Velho for his feedback on this report that helped me improve my text flow for those first paragraphs, and especially Michael Mercier for reviewing the whole paper. I warmly thank Milian Poquet for answering my interrogations whenever I had some. I also value the time I could spend at the LIG and I thank the DATAMOVE team for their sympathy for the time that I was there. Finally, my gratitude also goes to the Ryax team that helped me maintain human contact during the lockdown induced by the covid pandemic. The regular meetings we had and their positive feedback were essential to me under this very particular context.

Contents

— 1 —
Introduction

The need for scalable computing infrastructure has increased tremendously in the last decades, with systems now being designed to reach exascale computing. Nearly every field of computer science, from research to the service industry, now needs a proper infrastructure. By 2025 computation technology could reach a fourth of the global electricity spending [1], showing how much infrastructures are developing and that this need is still growing. The development of smart cities is also bringing new challenges to the table of distributed computing with the emerging technology of edge computing.

Organizations generally know what type of infrastructure will meet their needs. It can take the form of huge data centers to store and analyze data, HPC cluster with GPU banks to run application such as machine learning algorithms, crypto-currency mining, genome sequencing or weather forecast. However, studying those infrastructures extensively is challenging. As these computers require warehouse like surface [2], quantifying a system's efficiency under various loads, applications, scheduling policies, and system size quickly becomes unfeasible without expensive experiments that requires hours of labor and expensive measuring tools. Also, for businesses the improvements made in resource management efficiency by conducting such experiments hardly ever justify the cost of those experiments. Running experiments on real systems also pose a major issue of reproducibility because those systems are never open to the public. Additionally, the software environment of a system is susceptible to change over time further reducing the odds of getting the same conditions to reproduce an experiment. Finally, it is impossible for researchers to modify the hardware of one system which limits the range of their experiments.

Theoretical studies are also out of the question because of the multiplicity of the components at stake. In a RJMS[1] the scheduler interacts directly with the queuing system, the resource manager, external events like resource failures, jobs killed by users and the infrastructure itself: its topology, size, and type and capacity of the resources. Also, the objectives in performance vary from system to system: high resource utilization, lower energy consumption, or low waiting times for the user for example as well as the scheduling policies.

Simulation allows to tackle these issues by enabling users to draw conclusions empirically without the need to fire up real workloads. With simulation, the gain in both time and spent energy can be extreme: a HPC job spanning months on a real system can be simulated in a matter of minutes on any domestic computer. Another major point is that it also brings reproducibility to these experiments, that otherwise would have to be run on the exact same systems as their first iteration. With simulation, one can recreate the same conditions for any experiment anywhere they want, and expect the same results.

However, simulations need to be run with sound models for the results to be exploitable and in that regard, simulators may fall under several pitfalls [29]. Very often simulators are implemented with new schedulers or RJMS in order to validate their algorithms. Thus, they are strongly coupled together and are not usable with any other software. They are either shipped with the software itself or worst, they are never released and discarded at the end of the development process. Moreover, still according to [29], strong

[1]The RJMS is the software at the core of the cluster. It is a synonym for a scheduler and manages resources, energy consumption, users' jobs life-cycle and implements scheduling policies.

coupling may lead to unrealistic models because in that case the scheduler often has an instant access to all the information it requires about the system. This conflicts with the real world as schedulers may not have access to that much information on a system or may suffer from latency, impacting the scheduling decisions.

To try and assess these issues a team of researchers at the LIG developed Batsim[12] which is a general purpose infrastructure simulator with separation of concerns in mind. Batsim is based on SimGrid[10] which is a framework for developing simulators for distributed computer systems. Simgrid is now a 20 years old framework, which accuracy and scalability were extensively tested over the years under complex networks, topologies and workloads.

Batsim was designed to support algorithms written in any languages, as long as they support its communication protocol. It means that, while any scheduler found in the wild can potentially be run on a Batsim simulation, they still have to be adapted to make them compatible. This master's project is dedicated on developing an interface between Batsim and Kubernetes[2] schedulers to run clusters simulations. Kube[3] is an open source container management software widely exploited in the industry for its ease of use and wide range of capabilities. It has freed developers from the cumbersome task of setting up low level software on their servers to run their services and automates maintenance, scaling, and administration of their applications. For all these reasons it has become a standard solution for any organization that wishes to build new platforms from the ground up with limited means as well as for large and established institutions that wish to leverage container technology to have more control over their resources.

This memoir depicts how we were effectively able to run simulations of Kubernetes clusters for any scheduler written in the Go programming language. The objective is to prove that adapting Kubernetes schedulers to Batsim is possible and test the viability of the approach. One one side, Kubernetes schedulers follow the asynchronous paradigm of APIs to interact with the rest of the cluster, and on the other side Batsim was designed to run simulations with event based and deterministic schedulers. The challenge is to build an adaptive layer between those schedulers and Batsim. This interface is called Batkube and is capable of running simple simulations of Kubernetes clusters.

[2]https://github.com/kubernetes/kubernetes/

[3]A short term for Kubernetes. It is also sometimes called k8s.

— 2 —
Background and related work

2.1 Studying computer infrastructures

Even though the containers paradigm enabled developing new applications with ease, many questions remain: what type of infrastructure would be best suited for my application? Would my application benefit from more cpu cores? How would different scheduling policies affect my application? Would my batch jobs compute faster with a different topology? To answer these interrogations one must conduct studies to experiment with different configurations.

Studying an entire computing infrastructure is not an easy feat, first because every infrastructure is unique. There are as many types of infrastructure as there are use cases, each having a different vision on efficiency and what metrics are critical to the system: latency, bandwith, resource availability, computational power or cost effectiveness (which boils down to energy efficiency). This variety of purposes translates to the type of hardware used and the topology of the infrastructure. Some systems are centralized like HPC and Data Centers, others are meant to be used from a distance like Cloud Computing infrastructures and others are decentralized like Grid Computing, Volunteer Computing and Peer to Peer computing. There are as many systems as there are objectives to be achieved.

As a consequence, there are no general tools to study those systems. Furthermore, as the biggest supercomputers are approaching the exascale barrier[1] and consist of thousands of nodes with millions of cpu cores (more than 7M for the new "Fugaku" Japanese supercomputer), no human would be capable of building a general mathematical model that would be accurate enough to predict the behavior of those systems under varying conditions. Also, interactions between the various components of those systems may lead to unexpected behavior[17] that can hardly be predicted.

Scientist still have tools to study those systems. More precisely, there are 3 options as described in[22]: *in vivo*, *in vitro* and *in silico* studies, which correspond respectively to experiments on real testbeds, emulation and simulation.

in vivo and *in vitro* studies

The most direct approach to study an infrastructure is running *in vivo* experiments, that is to say running experiments on a real testbed. This will produce the most accurate results, however it poses major scalability and reproducibility issues.

Experiments conducted on real systems may prove difficult to reproduce, as one must have access to the same system in the first place. Event in the eventuality where access to the exact same system would be possible, changes in the infrastructure and software environment over time would diminish even more the chances of recreating a sane environment. Moreover, studying new algorithms imply testing them under a

[1] https://www.top500.org/news/japan-captures-top500-crown-arm-powered-supercomputer/

wide range of systems (under different system size, topology, and hardware) which is impossible to do with a real computer as we can not change the hardware of those infrastructures.

One solution to these issues is running *in vitro* studies, that is to say run an emulation of the system. With this approach the system is reproduced using containers or virtual machines to run the application as if it was run on the real system. MicroGrid [24] is one such emulator and allows the user to emulate an entire grid system to study the interaction between applications, middleware, resources and networks. This resolves the issue of reproducibility, however the matter of the cost in energy and time remains. If anything, emulation aggravates these costs: emulation adds a consequent overhead to the system and running the same workloads in reasonable time requires access to a system of (at least) equivalent size. This cost is exacerbated by the many iterations of a same experiment one must conduct in order to get statistically significant results. Workloads submitted by real users can last from hours to months and have substantial costs in energy: the means required to run them are too great and research to optimize or simply study these systems can not justify this waste of resources.

in silico studies, or simulation

The answer to these scalability considerations is simulated infrastructures. Simulation allows scientists to conduct experiments or thought experiments that would otherwise not be possible in the real world. One can think of simulations of the universe, prediction models for the weather or modeling some microbiome in biology. Computing itself makes no exception and researcher have created models of computer systems in order to experiment with new scheduling policies, network topologies or planning for systems capacity, for example. Simulation dramatically reduces experimentation cost and allows for reproducibility. Workloads on supercomputers may span weeks or months, whereas a single standard laptop can simulate this same workload in a matter of seconds or hours, depending on the simulator. More importantly, other scientists would need access to the same system the experiment was run on to reproduce the experiment whereas a simulator supposedly brings the same output regardless of the device it is run on. The only caveat that remains in terms of reproducibility lies in the application traces used to run off-line simulations that contain critical data concerning the application it was generated with.

However, even though simulators theoretically allow for effortlessly reproducible experiments, the way they are developed sometimes make them hardly usable. Indeed, lots of simulators are only intended to be used only by their developers, in order to validate the results of one particular work or paper. These simulators are generally unreleased, and when they are, they end up unmaintained. As a consequence, the experiences they served on are *in fine* just as difficult to reproduce as the experiences conducted on real systems. Another issue that this induces is that simulators are specialized to tackle one specific problem, and end up being built upon over-fitted models to validate this problem. These simulators cannot make sense in any other context and therefore can not be re-used for any other project.

Intuitively, one can believe that specialization is necessary to obtain the most accurate results as possible on a given problem. This is why so many domain specific simulators have been developed over the years and we count simulators in every field of distributed systems. PeerSim [25] and OverSim [3] for example are simulators specialized in the simulation of peer to peer networks. In a related domain, SimBA [13], SimBOINC [19] and SimGrid-BOINC [11] are simulators focused on the modeling of Volunteer Computing networks. In the HPC domain, we count two main branches: *off-line* and *on-line* simulation. In *on-line* simulation, an application is executed on a platform that tries to mimic a target platform. Therefore, it falls under the domain of emulation rather than simulation. In *off-line* simulation the simulator replays a time-stamped log from one execution of an application, as if it were happening on a different system. We count numerous simulators in this domain: LogGOPSim [16], Psins [32] or BigSim [38] for instance. Coud computing also has its simulators, we can cite CloudSim [7] for example or GroudSim [27] which simulates an environment with both Cloud and Grid capabilities. Some simulators are even specifically tied to some RJMS such as YARNSim [23] or the SLURM simulator [31].

SimGrid [10] however is a framework for simulation that builds on the idea that versatility serves accuracy, while ensuring high scalability.

2.2 SimGrid

SimGrid [10] is a framework for builing distributed systems simulators and is written in C. It uses simple analytical models for all its resources to ensure high scalability, and also because those models proved accurate. For instance, a task execution time boils down to a compute cost divided by the compute speed of the resource it is allocated to. Just like CPU which are not modeled at the cpu cycle level, network transfers and disks are not modeled at packet level and block level either. It was developed – and improved – with versatility in mind, which according to its makers was the key to its excellent performance both in accuracy and scalability. This *versatility* is not to be confused with *genericity*: their models provide some amount of genericity so that they can be improved for specific tasks, but they are versatile enough to cover various of domains, each accurately.

Projects based on SimGrid span accross a wide range of domains within distributed systems. WRENCH [9] is a *Workflow Management System* library that provide a high level API to eable the study of WMS. Simbatch [8] is a (discontinued) batch systems simulator made for the study of batch schedulers. Work has been done on the side of Volunteer Computing as well with SimBOINC [19] that has been discontinued as well due to major reconstructions of SimGrid and the BOINC client or SimGrid-BOINC [11] by some of the same people behind SimGrid. And of course the recent Batsim [12] this work is based upon, which is a RJMS simulator, relies on SimGrid. All these projects prove the ability of SimGrid to perform in very different contexts, and SimGrid models are known for their validity and scalability.

GridSim [6] is another library for development of grid computing simulators. Contrary to SimGrid it is coded in Java and it is therefore cross platform. GridSim has a broader view on what a grid is: when SimGrid models wired networks and therefore only allows for specifications of localized resources, GridSim allows resources to be specified in any time zone. GridSim also support resources in *time-shared* or *space-shared* mode while SimGrid only models time shared resources. Modeling space shared resources allows GridSim to simulate multiple users competing to submit jobs simultaneously on the same resources, which is a feature that would require extending SimGrid models.

Eventhough GridSim is very popular in its domain, its models are not as valid as SimGrid models. According to SimGrid's team a simple inspection of GridSim's code and its follow up cloud infrastructure simulator CloudSim [7] is enough to find invalidating cases to its networking models [35]. Moreover, still according to SimGrid team, scalability tests showed that GridSim complexity is quadratic in the number of tasks and linear in the number of workers, as well as having a considerable memory footprint compared to SimGrid [10]. In comparison, SimGrid simulation time and memory usage are more stable and polynomial at best both in the amount of tasks and workers.

2.3 The scheduling problem

One notorious problem on the field of distributed systems is the allocation of queued jobs to available resources.

> **schedule** *n.* : A plan for performing work or achieving an objective, specifying the order and allotted time for each part.

In a general way, scheduling is the concept of allocating available resources to a set of tasks, organizing them in time and space (the resource space). The resources can be of any nature, and the tasks independent from each others or linked together.

In computing the definition remains the same, but with automation in mind. Schedulers are algorithms that take as an input either a pre-defined workload, which is a set of jobs to be executed, or single jobs submitted over time by users in an unpredictable manner (as it is most often the case with HPC for example). In the latter case, the jobs are added to a queue managed by the scheduler. Scheduling is also called batch scheduling or batch processing, as schedulers allocate batches of jobs at a time. Jobs are allocated on machines, virtual or physical, with the intent of minimizing the total execution time, equally distributing resources, minimizing wait time for the user or reducing energy costs. As these objectives often contradict themselves so schedulers have to implement compromises or focus on what the user requires from the system.

The scheduler has many factors to keep in mind while trying to be as efficient as possible, such as:

- Resource availability and jobs resource requirements
- Link between jobs (some are executed in parallel and need synchronization, some are independent)
- Latency between compute resources
- Compute resources failures
- User defined jobs priority
- Machine shutdowns and restarts
- Data locality

All these elements make scheduling a very intricate problem that is at best polynomial in complexity, and often NP-hard ([34], [28], [5]). In order to better study the effect of different scheduling policies on a system a researcher team at the LIG have created Batsim which is a versatile distributed system simulator built on SimGrid and focused on the study of schedulers.

2.4 Batsim

Batsim[12] is a distributed system simulator built upon the SimGrid framework. Its main objective is to enable the study of RJMS without the need to implement a custom simulator, by decoupling the decision process from the simulator itself.

It is deterministic, allowing the exact reproduction of experiments. Its event-based models will provide the same results given the same inputs and decision process. Unlike other HPC or grid computing simulations that run on existing application traces, Batsim takes a user defined workload as an input. This allows the user to fine tune workloads in order to study very specific issues and achieve different level of realism. In addition, it provides tools to translate workloads between its own format and the swf format, which is a commonly used structure for HPC workloads.

Batsim, just like SimGrid, aims at being versatile. Batsim computation platforms are SimGrid platforms meaning that theoretically, they may be as broad as SimGrid allows it. In reality any SimGrid platform is not a correct Batsim platform. Because Batsim aims at studying RJMS software, it requires a **master** node that will host the decision process. The other hosts (or computational resources) will have either the roles of **compute_node** or **storage**. Still, the user may study any topology he wishes using SimGrid models.

Thanks to its own message interface Batsim is language agnostic which means that any RJMS can be plugged into it as long as it implements the interface. This property allows us to plug any scheduler we wish to Batsim, including Kubernetes schedulers, which allowed us to effectively run simulations of Kubernetes clusters.

2.5 Kubernetes in the context of Cloud Computing

In the early stages of application development, organizations used to run their services on physical servers. With this direct approach came many challenges that needed to be coped with manually like resources allocation, maintainability or scalability. In an attempt to automate this process developers started using virtual machines which enabled them to run their services regardless of physical infrastructure while having a better control over resource allocation. This led to the concept of containers which takes the idea of encapsulated applications further than plain virtual machines (Figure 2.1).

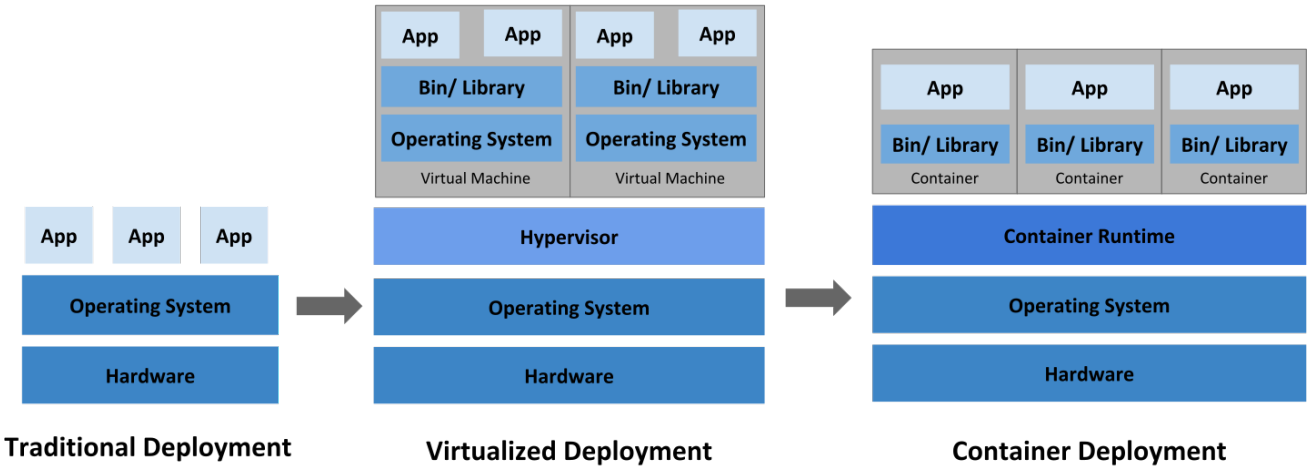


Figure 2.1 – Evolution of application deployment.
Source: *https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/*

Containers can be thought of as lightweight virtual machines. Unlike the latter, 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).

The *Cloud Native Computing Foundation (CNCF)* [2] was founded in the intent of leveraging the container technology for an overall better web. In a general way, we now speak of these containerized and modular applications as cloud native computing :

"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."[3]

Kubernetes[4] is the implementation of this general idea and was announced at the same time as the CNCF. It aims at automating the administration of a fleet of containers. Deployment and scaling (that is to say, replication of one same application to increase its availability) is described in yaml files and managed by Kubernetes. It is industry grade and is now the de-facto solution for container orchestration.

[2] `https://www.cncf.io/`
[3] `https://github.com/cncf/toc/blob/master/DEFINITION.md`
[4] `https://kubernetes.io/`

2.5.1 HPC and Kubernetes

In this work we use Kubernetes schedulers as batch schedulers because batch scheduling is our domain of origin (Batsim was developed in close relation with the teams behind the grid computing simulator SimGrid[10] and the resource manager OAR [26]), and because it is convenient for us to test our approach with HPC workloads. However, even though Kubernetes does have means to handle such workloads like batch schedulers [20] and a special type of resource[5], these two fields differ fundamentally because of the nature of the issues they tackle. We do not intend to evaluate scheduler performance but rather prove that such studies are possible with Batsim, which is why this work is conducted this way. Still, there are perspectives for Kubernetes in the HPC community and we believe HPC systems could benefit from this work.

The difference between HPC and cloud computing lies in the workloads they are intended to tackle. Kubernetes was designed to run applications. Services or micro services are run in containers and are expected to be available at all times : they are replicated as many times as desired and restarted whenever a failure occurs. High availability is at the core of Kubernetes container management. On the other hand, depending on implemented scheduling policies, HPC is focused on metrics such as user wait time, resource usage or energy cost. One example of a fundamental difference between cluster computing and batch scheduling is the handling of a task failure. In HPC it is sometimes not sufficient to restart the single job that failed and the entire submission must be re-run if it is part of several jobs computed in parallel. In cluster computing the task is simply restarted without any other consideration – other than ensuring its dependencies are restarted with it.

Kubernetes is now the standard for AI and Machine Learning as shown by the many efforts at making this coupling an efficient environment[21][33][30], which brought an increasing interest for container driven HPC aswell and Kubernetes for HPC in particular. Batch schedulers such as kube-batch[6] have been implemented for kube, and numerous HPC applications like slurm[7] now support containers as well.

Indeed, containers have many advantages from which HPC users could benefit from.

- First off, research has shown that Kuberenetes offer similar performance to more standard bare metal HPC[4].

- Users will get the same environment everywhere making up for a uniform and standardized workplace.

- Portability : users could seamlessly hop from one infrastructure to another based on their needs and criteria like price, performance, and capabilities rather than compatibility.

- Encapsulation : HPC applications often rely on complex dependencies that can be easily concealed into containers.

Despite all those advantages, Kubernetes is not ready yet to be used in proper HPC environment because it lacks vital components like a proper batch job queuing system and support for MPI applications. It cannot yet compete against the very well established HPC ecosystem, but that time may come soon as containers are becoming more and more integrated in modern infrastructures.

2.6 Related work

Not many projects exist on Kubernetes simulation or have been disclosed. We were able to find two projects that propose simulations of Kubernetes clusters.

[5] https://kubernetes.io/docs/concepts/workloads/controllers/job/

[6] `https://github.com/kubernetes-sigs/kube-batch`

[7] `https://slurm.schedmd.com/containers.html`

JoySim [37] is a fully fledged Kubernetes simulator developed in an industrial context. Simulations are based on synthetic events and their mock nodes simply simulate a resource usage. The strength of this simulator is its scalability which it obtains thanks to its very light weight simulated nodes. JoySim is aimed at studying the quality of the scheduling and its performance in complex scenarios by generating metrics such as resource usage and scheduling time. Although, to the best of our knowledge, it is not suited for batch scheduling and tackles more classic issues for Kubernetes such as services availability and resource utilization. The user may specify events to occur during the simulation in order to test out different scenarios. Unfortunately, while an open source released in planned, this piece of software is currently closed source.

k8s-cluster-simulator [36] is an open source cluster simulator for evaluating schedulers, which originated from a end of studies project like this work. Like JoySim, it relies on simple models: pods are submitted with certain resource requirements, for a certain amount of time. One can provide any scheduler implementation they want as long as they do so through their Go interface. This reduces the amount of schedulers that can be tested to Go implementations only, unless the user is willing to implement an adaptive layer to support other schedulers.

Two simulators are very close to Batsim in terms of capabilities and objectives, that is to say the study on scheduling policies.

Alea2[18] is a grid computer system simulator based on GridSim. It is therefore written in Java and cross platform. Like Batsim, its *ready-to-use* philosophy allow experimenting with different scheduling policies with little set up overhead. Alea2 strength lies in its modularity: its object oriented paradigms make it very easily extensible and customizable. Unlike Batsim it does not offer a decoupling of the simulator and the decision process and therefore the user will have to rely on the implemented scheduling algorithms, although these algorithms cover the most standardly used scheduling policies. The user will have to implement other policies he would like to test out. The simulator handles inputs in the form of *Standard Workloads Format (SWF)* or *Grid Workloads Format (GWF)*. Batsim proposes a script to process SWF files into its own input format but does not handle them directly.

Accasim[14] is a simulator for studying *Workload Management Systems (WMS)* in HPC infrastructure. It is similar to Alea in the sense that the decision process is not decoupled from the simulator and standard scheduling algorithms are implemented in Accasim, so that the user does not need to set up extra software in order to start experimenting. In addition, the user may provide extra data about the system status (power consumption, temperature or resource failures) in order to support advanced scheduling algorithms. These take the form of additional events transmitted to the simulator. Accasim loads jobs incrementally and cleans them upon completion which reduces its memory usage compared to Batsim, which loads everything in memory at the start of the simulation.

What truly distinguishes Batsim from these simulators is the strong decoupling between the RJMS and the simulator, which allows us to experiment with RJMS software rather than only studying scheduling policies. This allows us to evaluate the scheduler computing needs or memory usage independently, as if it was running in real conditions. Performance wise, we cannot really compare Batsim with Alea and Accasim because of the fundamental differences in their models. When Alea and Accasim chose to implement simple models in order to focus on scalability, SimGrid models are more advanced and require more computing power.

— 3 —
Integrating the simulator into Kubernetes

Batsim is able to run simulations of any distributed system, to study any event-based scheduler that would implement its message protocol. Kubernetes is a piece of software where all its component, including the scheduler, revolve around a central API. Everything is asynchronous as the API can be accessed anytime by any component.

The question that arises is, can we adapt Batsim to make it support Kubernetes schedulers? Is it possible to implement an adaptive layer between a synchronous event based simulator like Batsim and a scheduler implemented following the asynchronous paradigms of APIs? To answer these issues we developed Batkube[1], which is an adaptive interface to Batsim for Kubernetes schedulers.

It will follow that in order to do so, we re-implemented an API following Kubernetes specifications and intercepted the scheduler's time to synchronize it with the simulation time. This allows us to simulate lengthy workloads in a short period using a scheduler otherwise supposed to rely on "real" machine time. We first describe some technical concepts about Kubernetes and Batsim, and then describe how we re-implemented the API, intercepted the time, and handled the synchronization of the different times between Batsim and the scheduler.

3.1 Batsim concepts

A Batsim simulation is divided into two processes: Batsim itself and the decision process (the scheduler). As a consequence, Batsim defines its own messaging protocol to be able to standardize exchanges with the scheduler. This protocol takes the form of a text based interface that conveys the events occuring during the simulation.

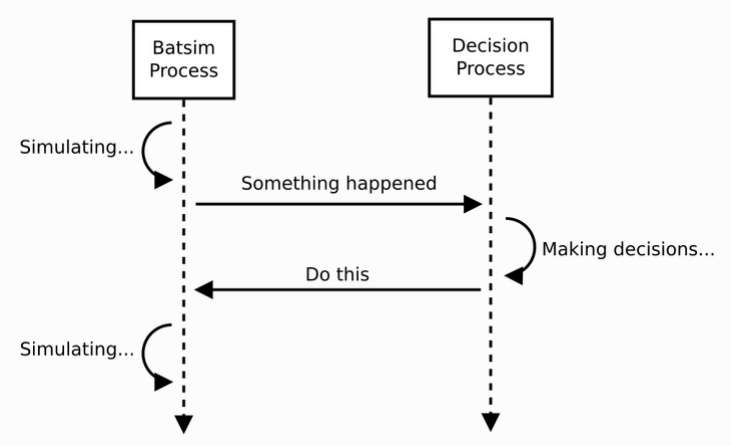


Figure 3.1 – Exchanges between Batsim and the scheduler.
Source: *https://batsim.readthedocs.io/en/latest/protocol.html*

[1] github.com/oar-team/batkube

A Batsim platform is a SimGrid platform, defined in the xml format. A node can endorse the role of *master*, *compute_resource* or *storage*. Here we only consider master nodes which host the decision process, and compute resources to which we add our custom resource capacities. These additional fields are *core*[2] which is the amount of cpu the node has and *memory* which is memory capacity of the node. Of course, storage resources can be taken into account in future development of Batkube. Also, we do not consider the links between the nodes for now because we do not support parallel jobs. Finally, Batsim proposes an energy model that we decided to ignore as well.

Batsim takes one or several workload as inputs, which are json files containing jobs definitions. Figure .9 gives an example of such workload. The jobs are defined with the following inputs. First, they are identified by an *id* which is unique within each workload and are submitted at a time defined by the *subtime* field. The *res* field states the number of resources each job requires, although we don't use this field because we can't specify *which* resource we require. Instead we pass resource requests as optional parameters. We support the additional fields *cpu* which has a minimum value of 100m cpu just like Kubernetes cpu requests[3] and *memory* which also complies with Kubernetes compute resource definitions. Jobs follow a certain *profile* that defines the nature of the job. These profile may be as simplistic as *delay* profile which makes the resource wait for a given amount of time, or describe parallel tasks using matrices to describe the amount of exchanges between the allocated nodes. For now we only consider delays to simplify the implementation of the simulator. Also, because Kubernetes allows the use of multiple schedulers[4], we support the field *scheduler* which contains the name of the scheduler the job should be scheduled with.

Batsim messaging interface is based on its protocol. Each message is composed of the field *timestamp* which contains the current simulation time, as well the field *events* which is a list of events either from Batsim to the scheduler, or from the scheduler to Batsim. Figure .8 depicts a standard message sent from the scheduler to Batsim. Batkube features are limited because we focused on building a working proof of concept rather than a fully fledged Kubernetes simulator which is why only consider a subset of these messages. We provide a list of supported events and their description in the section **??** of the appendix. More information on Batsim's protocol is available on Batsim documentation[5], and a list of the events handled by Batkube can be found in section .2, with examples of some Batsim messages.

Batsim's output takes the form of a csv file containg information about the jobs executions. Mainly we take interest in their submission time, execution time and waiting time. Again, a detailed list of Batsim outputs can be found on the documentation[6]. During our experimentations with Batkube we interest ourselves in two metrics that can be computed from this output:

- The *makespan*, which is the total length of the simulation. It is defined as the timestamp at which the last job finishes executing, minus the origin (in this case, zero).

- The *mean_waiting_time*, which is the mean time the jobs spent waiting for a scheduling decision. The waiting time is defined by the duration between the submission time and the starting time (here the starting time is equivalent to the time at which the job was scheduled. We will see later that these two times do not correspond in Kubernetes.)

These metrics were chosen because in our case, they are representative of the accuracy of the simulation. Time synchronization (see section 3.5) may introduce delays in the scheduling decision, thus increasing the overall makespan and mean waiting time of the simulation.

[2]The core field is already present in the resource definition, but it is not forwarded to Batsim for unknown reasons at the time this report is written.

[3]https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[4]https://kubernetes.io/docs/tasks/extend-kubernetes/configure-multiple-schedulers/

[5]https://batsim.readthedocs.io/en/latest/protocol.html

[6]https://batsim.readthedocs.io/en/latest/output-jobs.html

3.2 Kubernetes concepts

Kubernetes is now a large ecosystem to which more than 2 700 developers have contributed over the years. In this section we present the fundamental concepts and the components that make up Kubernetes.

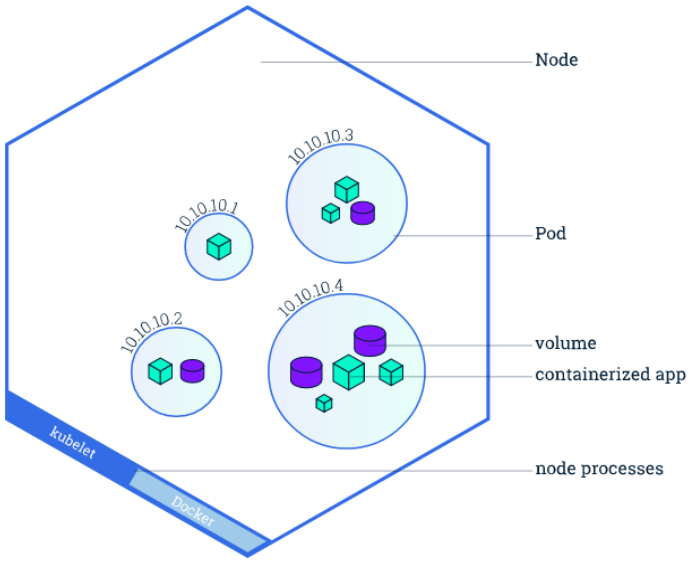


Figure 3.2 – Node overview
Source: *https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/*

The basic processing unit of Kubernetes is called a **pod** which is composed of one or several containers and volumes[7]. The type of application they contain vary depending on the context: in a web platform context a pod most often hosts a service or micro-service that must be available at all times, in opposition to a batch processing context where it runs an application that is to be executed in a finite amount of time. Pods are bundled together in **nodes** (figure 3.2) which are either physical or virtual machines. They represent another barrier to pass through to access the outside world which bundles pods under the same network to facilitate communication between them, and enables the use of proxies to access the underlying services. A set of nodes is called a **cluster** which is the highest abstraction layer in Kubernetes.

Nodes take the idea of containerization further than plain containers by encapsulating the already encapsulated services. Each node runs at least one pod, the **kubelet**, which is a process responsible for communicating with the rest of Kubernetes. More precisely, the kubelet communicates with the **kube-api-server** which is responsible for the whole cluster. We refer to this API as the api-server, as it is called within the Kubernetes community. This API server, as well as the other components of the **Control Plane** (figure 3.3), can be run on any machine but for simplicity they are set up on the same machine at start up. This machine is often called the **master** node and typically does not run any other container.

As stated before Kubernetes revolves around its API server which is its central component. All operations between components go through this REST API. These operations take various forms like user interactions through the commande line interface **kubectl**, scheduling operations or management of cluster data on **etcd**. Kubernetes schedulers can communicate with the api-server in various protocols. These can be text based like json messages or binary with the use of Google's protocol buffers. To help with building schedulers, Kubernetes has developed several clients[8] to facilitate communication with the API.

The most notable Kubernetes scheduler is the `kube-scheduler`[9] which is the default and also most commonly used sheduler. `kube-batch`[10] is a project aimed at developing batch scheduling in Kubernetes,

[7]Because of their transient nature, containers can not store data on their own. A volume is some storage space on the host machine that can be linked to containers, in order for them to read and write persistent information.

[8]https://kubernetes.io/docs/reference/using-api/client-libraries/

[9]https://github.com/kubernetes/kube-scheduler

[10]https://github.com/kubernetes-sigs/kube-batch

which is already used by several projects. One of those projects is Volcano[11] already used in some platforms for computational and data intensive domains such as deep learning or genome sequencing in Kubernetes. Yet another important scheduler is Poseidon [12] which is rather an adaptive layer to Firmament. All those schedulers make use of the Go client to communicate with the api-server.

We then decided to re-build the API in order to simulate any cluster to any Kubernetes scheduler.

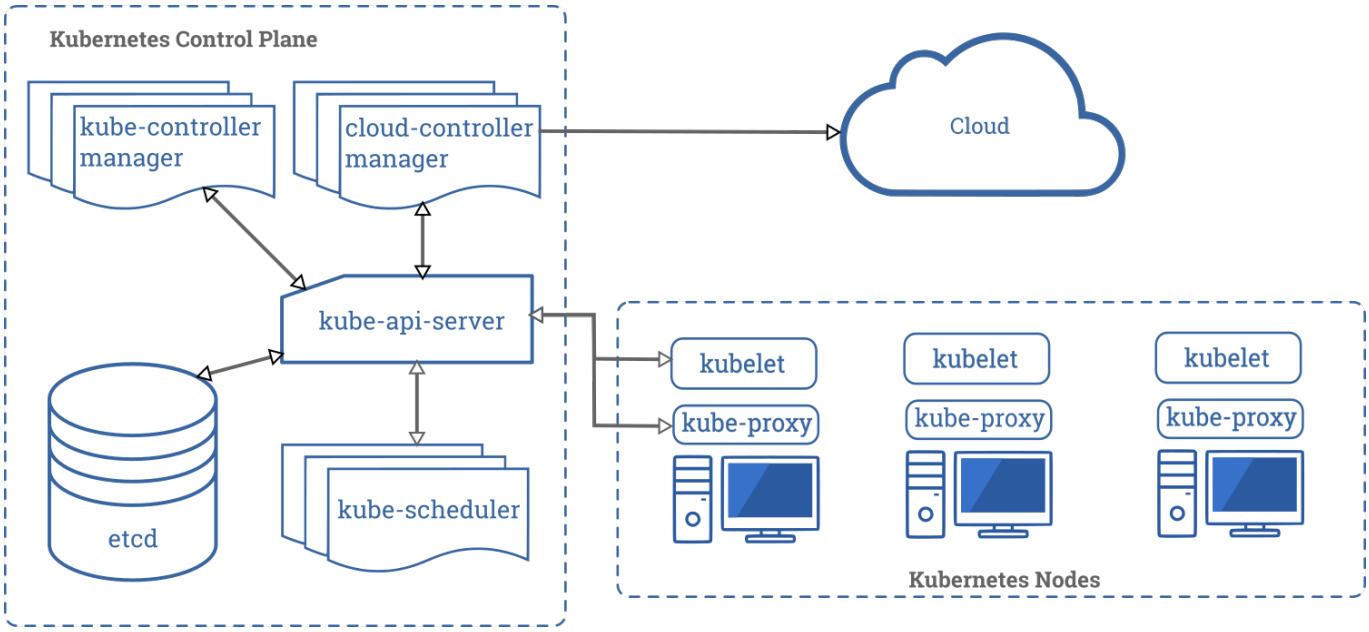


Figure 3.3 – Components of Kubernetes
Source: *https://kubernetes.io/docs/concepts/overview/components/*

3.3 General architecture of Batkube and its integration with Kubernetes and Batsim

3.3.1 Integration with Kubernetes

In order to adapt Kubernetes schedulers for use with Batsim we need to position ourselves between the Kubernetes scheduler and the cluster. Several level of implementations are possible, and after some considerations described in section .3 we decided to reimplement Kubernetes API. This new implementation will act as a Kubernetes cluster to the scheduler, and as an event based scheduler to Batsim.

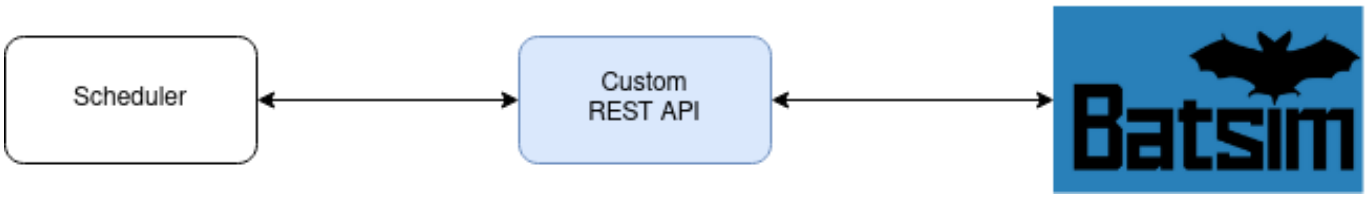


Figure 3.4 – Custom REST API in between the scheduler and Batsim.

Building a new API allows us to consider only the endpoints we need and have complete control over the source code. The technically challenging aspect here is Kubernetes resource management. Indeed, we need to provide the scheduler with expected informations about the cluster state if we want to obtain a

[11] https://github.com/volcano-sh/volcano
[12] https://github.com/kubernetes-sigs/poseidon

correct behavior from it, and while the endpoints of the API are well documented, Kubernetes team did not write lengths about how resources are managed internally. In order to save us the hassle of writing the entire code of the server, we used tools to generate the code from the api-server specification as described in section 3.4.

3.3.2 Architecture of Batkube

Figure 3.5 depicts the architecture of Batkube, which is written in Go. The central component is the **broker**. It handles the messages coming from Batsim and the scheduler while ensuring time synchronization between them. It is responsible for translating and forwarding messages between Batsim and the scheduler and orchestrates the synchronization between the two parties. **batsky-go** intercepts the calls to Go *time* library to ensure the scheduler's time is based on the simulation time instead of machine time. Time requests are forwarded to Batkube which replies with the current simulation time. The complete process to "hijack" scheduler time is explained in section 3.5. The **rest api** is the reimplementation of the Kubernetes api-server. It ensures the scheduler gets all the necessary information on the cluster state to make its scheduling decisions, and it is also the receiver of those decisions. Section 3.4 explains how we built this API and a tool to automatically generate its code from the api-server specification. **translate** is a utility package providing functions to translate Kubernetes resources to Batsim messages, and *vice versa*.

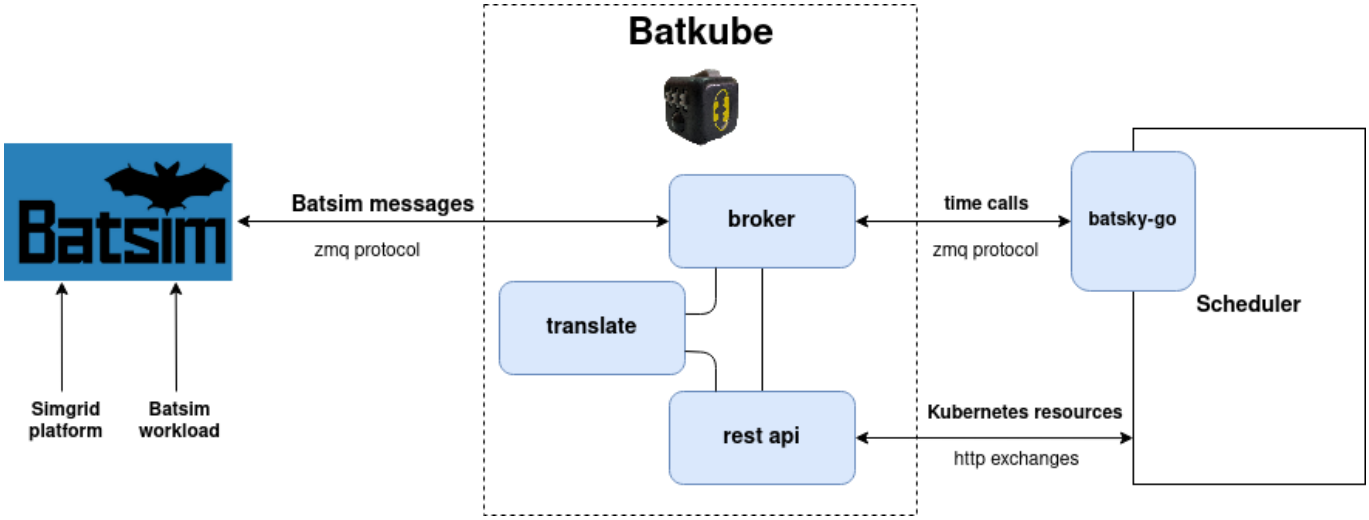


Figure 3.5 – Architecture of Batkube

3.4 Building the API

The API of Kubernetes follows the OpenAPI[13] 2 specification which is a standard for describing APIs. Luckily tools exist to generate such specification from source code, but also to generate code from a specification. Since the Kubernetes API specification is available on their reposotory[14], we were able to use such tools. This allowed us not to implement boiler plate code by hand and fill the gaps where they needed to be filled, leaving empty the endpoints we do not need. These endpoints can be dealt with later for future development of the simulator. For this project we used go-swagger[15] to generate our code and the API specification corresponds to the release 1.18 of Kubernetes. One downside of this method is that go-swagger forbids to tamper with the code of the server itself, although we did not need to during this project.

[13] https://www.openapis.org/about
[14] https://github.com/kubernetes/kubernetes/blob/release-1.18/api/openapi-spec/swagger.json
[15] https://github.com/go-swagger/go-swagger/tree/master/examples/stream-server

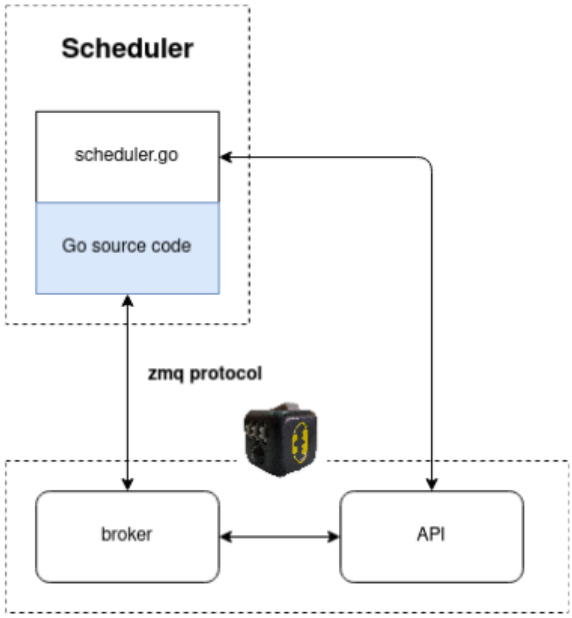


Figure 3.6 – Partially reimplementing Go source code allow us to intercept calls to get machine time.

In order to enable communication between Batsim and the scheduler we need to translate Batsim messages into Kubernetes resources that can be retrieved by the scheduler and scheduler decisions to Batsim messages. Batsim Jobs are simply translated to pods. Jobs do exist in Kubernetes, but they are simply wrappers around pods: when submitting a job to the (real) api-server, the api ensures that a pod is created and executed to completion. In our case, we do not need such intermediate. Compute resources are simply translated to Kubernetes nodes.

3.5 Time interception

Kubernetes schedulers are not event based schedulers. They constantly check on the cluster state and make decisions accordingly, therefore they are based on machine time to make their decisions. However, in order to have correct simulations, the scheduler needs to be synchronized with simulation time. We then need to intercept all calls to machine time to redirect them to the simulator.

This has been done already with the OAR[16] scheduler which was isolated and adapted to Batsim using a script written in python[17]. In order to ensure synchronization of the time between the scheduler and Batsim, a c library called batsky[18] was developed. Once compiled with this library, all calls from the scheduler to get machine time are redirected to the simulator.

Unfortunately, we were not able to recompile Go using a custom C library, which would have redirected every call to machine time to Batkube. Insted, we reimplemented parts of the Go *time* library and patched the kube-scheduler with it (figure 3.6) to redirect the calls to machine time. This tool was also tested against the kube-batch scheduler which received the patch without any issues, but that we couldn't use with the simulator because it implied more development on the batkube api to support it.

[16]https://github.com/oar-team/oar3
[17]https://github.com/oar-team/oar3/blob/master/oar/kao/bataar.py
[18]https://github.com/oar-team/batsky

```
ch <- v    // Send v to channel ch.
v := <-ch  // Receive from ch, and
           // assign value to v.
```

Figure 3.7 – Usage of a Go channel

3.5.1 Redirection of time requests to Batkube

The module in charge of this time redirection is called **batsky-go**[19], which re implements `time.Now()` as well as timers and tickers. Timers are structures that are instantiated with a duration as input, that notify the caller once this duration is elapsed. Timers can be reset, modified or deleted after initialization, which make their implementation tricky in a parallel computing context. Tickers are essentially the same structures as timers, except they regularly notify the caller with the given period of time instead of exiting after they fire like timers would.

In order to explain batsky-go algorithms as clearly as possible we need to provide some context about Go channels. Go allows the user to run multi threaded code easily thanks to its **go routines**. These routines allow the user to run code in parallel whithout requiring any setup by simply calling `go func()` where `func()` is a function. It creates a new thread and launches the given code in parallel with the encapsulating function. Essentially, this creates a child process. Go *channels* are *"a typed conduit through which you can send and receive values with the channel operator, <-"*[20]

A channel can be shared amongst several process and allow several pieces of code run in parallel to share data, without having to implement any mutex. By default sends and receives on a channel are blocking operations. If the two lines from figure 3.7 were to be executed one after the other, the program would remain stuck on the first line, as it would wait for some process to receive the data sent on `ch` which only happens on the second line.

Algorithm 1: Time request (time.now())

Result: Current simulation time
Input: req: requests channel, resMap: response channel map
Output: now : simulation time

```
1 if requester loop is not running then
2     go runRequesterLoop() /* There can only be one loop runing at a time */
3 id = newUUID()
4 res = newChannel()
5 resMap[id] = res /* A channel is associated with each request */
6 req <- id /* The code blocks here until request is handled */
7 now = <-res /* The code blocks here until response is sent by the requester
      loop */
8 return now
```

A small disclaimer is due here. Algorithm 1 describes the general concept behind batsky-go, but the true implementation differs slightly from what is presented here to account for timer requests. Timers durations are forwarded as an array of integers to Batkube which in turns translates them to appropriate CALL_ME_LATER events. This allowed Batkube to requests Batsim to wake up exactly when the scheduler timers are supposed to fire. This feature however was not used for our experimentations and wasn't tested any further.

[19] https://github.com/oar-team/batsky-go
[20] source: https://tour.golang.org/concurrency/2

Let us call `time.Now()` calls *time requests*. Channels enable us to centralize time requests from multiple simultaneous callers in one place before forwarding all requests to Batkube. There are two parts to `batsky-go`. One is composed of the many requesters, and the other one is the loop centralizing requests and handling exchanges with Batkube which we will call "the main loop". The algorithm 1 gives the new implementation of time.Now(), which is the "requester" part. Each request is identified using a unique id to keep track of the pending requests that are waiting for a response. `resMap` is a thread safe channel map that is shared amongst the requesters and the main loop containing (id, response channel) key-value pairs. When a new request is made, a new entry is created on this map and the requester will wait for a response on the newly created channel, associated with its id. All requests are sent to a unique channel `req`. To make a new request, the requester simply sends its id to this channel.

Algorithm 2: Requester loop

```
Input: req: requests channel, resMap: response channel map
1 while Batkube is not ready do
2     wait
3 requests = []request
4 while req is not empty do
5     id = <- req /* Non blocking receive                    */
6     requests = append(requests, id)
7 now = getCurrentTimeFromBatkube()
8 for m in requests do
9     resMap[id] <-now /* The caller resumes execution upon reception */
```

Each iteration of the main loop first waits for a signal from Batkube indicating that the broker is ready to handle the time request. That way, we make sure we don't miss out on any request because the requests accumulate on the `req` channel in the mean time, while minimizing potential downtime because we know Batkube will answer instantly. Once we get the signal that Batkube is ready all is left to do is retrieve the ids from our requesters, retrieve the time from the simulation, and answer each individual request. Figure 3.8 presents a typical exchange between the scheduler, batsky-go and Batkube.

3.5.2 Patching schedulers

We have developed a tool for patching the schedulers, so they use our library instead of the standard `time` library. This tool is called `batsky-go-installer` and is available on github alongside Batkube[21].

Our approach replaces all calls to the functions of the `time` library while leaving the objects as is to ensure compatibility. In that regard, our approach makes use of standard `time` objects instead of redefining new custom objects, which would break compatibility with the rest of the code. It makes use of Go Abstract Syntax Tree (AST) to go through the source files and replace the appropriate symbols, while adding the import to our module whenever it is necessary. Go `ast` package make it very convenient to search and replace symbols in the syntax tree. Unfortunately, we cannot replace the calls of the Go source code itself with this approach which creates inconsistencies between the scheduler code now based on simulation time, and native Go code still based on machine time. Aside from creating warnings, these inconsistencies did not prevent the scheduler to function correctly.

[21] github.com/oar-team/batsky-go-installer

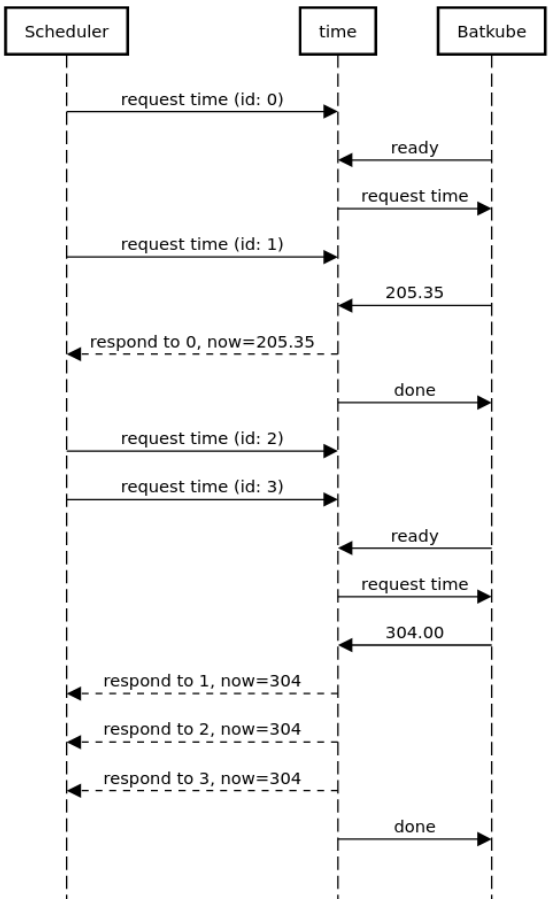


Figure 3.8 – Exchange breakdown between batsky-go and Batkube.

3.6 Time synchronization

Synchronization of the time between the scheduler and the simulator is the critical part of Batkube, because this is where we make compromises between the speed and the accuracy of the simulation. The scheduler is not event-based and therefore we can never know for sure when it will send a decision. Therefore we need to listen to the scheduler as much as we can, but while we do so the simulation advances slowly. In fact, whenever we listen to the scheduler and wait for a decision we synchronize machine time and simulation time so the scheduler gets a time that advances as it would if it was not simulated. This ensures that the internals of the scheduler function correctly. We could speed up this time to increase simulation speed and study the effect it has on the scheduler, but we have to leave this for future work unfortunately. During that time Batsim is blocked and waits for a message from the decision process. When we give back priority to Batsim time stops advancing on the scheduler side, effectively suspending the decision process. At this time Batsim advances forward in the simulation and replies with new events. Figure 3.9 gives a breakdown of the exchanges between Batsim, Batkube and the scheduler with their associated times.

The simulator can be tuned with several parameters which have various effect on the simulation. We study those effects in section 4.2.

- To limit the time we spend waiting for the scheduler we implemented a timeout policy. The *timeout value* defines the maximum amount of time we spend waiting for a scheduler decision: after this amount of time we get back to Batsim to go forward in the simulation. Whenever we receive a decision from the scheduler, we immediately give back control to Batsim by sending it the decision.

- Also, we added the *minimum delay* parameter, which is analogous to *timeout value*. It defines the mini-

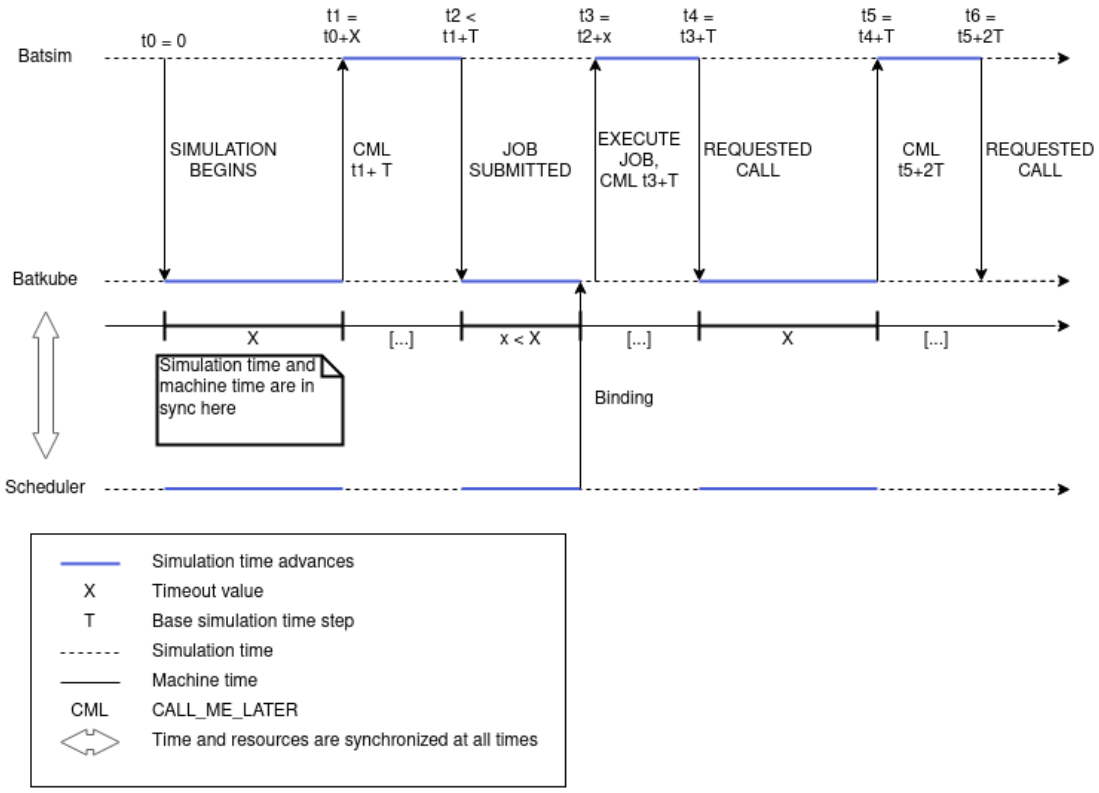


Figure 3.9 – Time sync between the three components. The broker has to take into account both machine time and simulation time.

mum amount of time we remain in Batkube each cycle: we will always wait at least this amount of time for the scheduler even though we already received a scheduling decision. This parameter was implemented because we observed incorrect behavior of the scheduler – it crashed – when Batkube responded too quickly. However, this issue seems resolved with latest updates of the Kubernetes scheduler and it seems we don't need this mechanic anymore. It might still be useful for other schedulers or earlier versions of the kube-scheduler which is why we keep it in the simulator.

- Whenever we block the scheduler to send a message to Batsim, the latter supposedly advances the simulation as far as it desires. This may impact the decision process because we can't know for sure if the scheduler would have made a decision in the mean time. If that is the case, this decision will be drastically delayed or canceled. This is why we only allow Batsim to jump forward in time up to a certain *simulation timestep*. This time step varies between *base-simulation-timestep* and *max-simulation-timestep*: it starts at the base timestep and increases following a backoff policy. This policy is implemented as follow: the time step is multiplied by a factor every time the scheduler did not provide any decision, and reset when we receive a decision from the scheduler. This ensures Batsim does not jump to much forward in time while increasing simulation speed when the scheduler does not have any decision to make. We tell Batsim to "wake up" at certain times thanks to CALL_ME_LATER (CML) events. When Batsim receives such events, it replies with a REQUESTED_CALL at the timestamp specified in the CML event.

- To speed up the simulation in its last steps, we implemented a parameter allowing Batsim to jump forward in time as much as desired when no jobs or in *pending* state, that is to say waiting to be scheduled. We allow such behavior when the scheduler is not supposed to make any decision when no jobs are to be scheduled, which is the case with the kube-scheduler. This would lead to incorrect simulations with more advanced schedulers implementing features such as *preemption*, which allows schedulers to kill running jobs in order to make room for large jobs, to resume the jobs that were killed later. This improves the overall makespan of the simulation.

- To improve the experiments stability we implemented a feature to detect scheduler crashes. Despite all our efforts at making simulation as stable as possible, simulated time still causes some crashes on the scheduler side. We detect such crashes and exit Bakube with an error code when it happens. A crash is detected when the scheduler had to make a decision but did not react for some time (that we can configure, independently of the *timeout value*). We consider that the scheduler has to make a decision when no jobs are running and some jobs are pending, waiting for a decision.

Like stated in section 3.5, Batkube supports a feature to wake up the scheduler exactly when its timers expire. However, this approach increases drastically the number of exchanges between Batkube and Batsim thus decreasing the simulation performances. This feature is considered deprecated and wasn't tested any further.

— 4 —
Evaluation and discussion

Because SimGrid has already been thoroughly tested and validated, we do not need to run extensive experiments to validate Batkube simulation models. Moreover, since we only consider simple workloads containing only delay jobs, whose resource requests are only cpu, a simple look at the gantt charts allow us to evaluate the quality of the scheduling. Still, even though the underlying models are sound, Batkube adds a considerable overhead to Batsim because of the time synchronization between the simulator and the scheduler. We want to verify to what extent time manipulation impacts the scheduler behavior, and also that Batkube's fake Kubernetes API mimics the real API well enough to let the scheduler run as expected. The validation experiments we conducted aim to verifying that the scheduler works as intended.

In the next sections, we present the workloads and platforms we chose to study, how we conducted experiments on a real cluster, and a study on Batkube's parameters and their effect on the outputs.

4.1 Experiments environment

The entirety of the experiments are done with the default Kubernetes scheduler **kube-scheduler** release **v1.19.0.rc-4** (commit 382107e6c84). This choice was made because it is the standard choice for Kubernetes and therefore the most commonly used scheduler in the industry, and because supporting another scheduler would mean more development time which we did not have. Still, it allowed us to conduct experiments to verify that the scheduler behavior was not altered by Batkube. Some instructions to reproduce the experiments are given in the appendix, section .1.

4.1.1 Real experimental testbed

In order to validate the simulator results we then need to compare it against workloads run on a real cluster. For reproducibility and simplicity sake, we choose to validate the simulator with an emulated cluster run in containers. We use k3s, which is a lightweight version of Kubernetes (k8s). It has all the essential features of Kubernetes we would need and has become a standard in the industry whenever administrators do not wish to run fully fledged versions of Kubernetes.

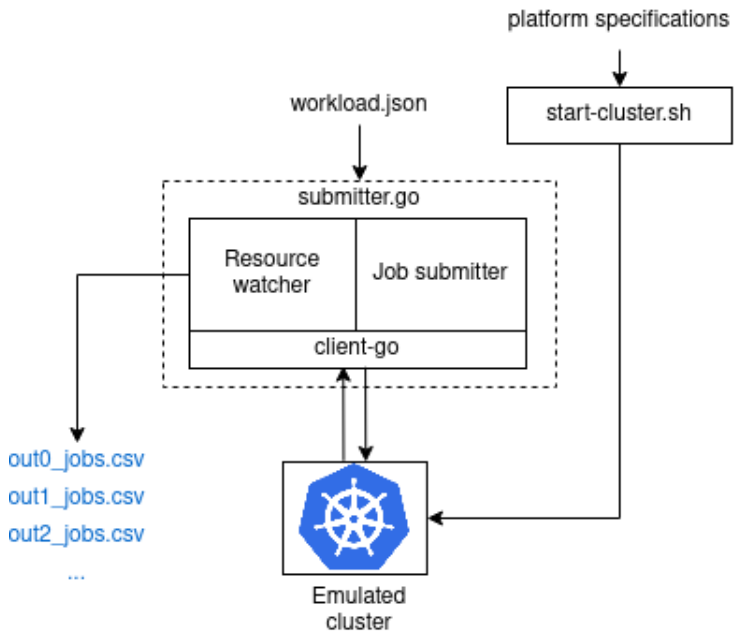


Figure 4.1 – An emulated experiment.

Figure 4.1 illustrates how this is done. First, we create a k3s cluster run using docker-compose, which is a tool enabling us to conveniently create one or several network of containers. Our cluster contains one master node and several workers which will run our pods. Then, a Go script which takes a Batsim workload as an input submits the jobs at the right time, watches the cluster state regularly, and writes the ouputs to csv files – which have the same format as Batsim's csv output files. We run each workloads 10 times in order to get statistically meaningful results, except for the realistic workload which we only run once because it is already 10 hours long.

The emulated cluster is limited in terms of variety and capacity. First, `start-cluster.sh` only allows the nodes to have the same amount of available cpu, memory or storage because there was no need for any complex system for our experiences. Secondly, the maximum amount of cpu, memory or storage we can make available for each node is capped to the host system capacities. For example, if the host system possesses 8 cpu cores, the nodes will have a maximum of 8 cpu available. This will have implications when trying to run workloads recorded on real systems: either we get to find a workload that complies with the host system capacity (which is very unlikely), or we adapt the workload so the jobs requirements do not exceed the host capabilities (see section 4.1.2).

4.1.2 Studied workloads

We consider three workloads, representing three different situations. The first two are simplistic and very controlled, and the last one depicts a more realistic case. In all cases the required resources are only quantified in cpu only to simplify the study. Note that Batkube does support memory requets, we just do not wish to add this other layer of complexity to our experiments.

- A *burst* workload, consisting in an important amount of jobs submitted at once. 200 delays with duration 170s and requesting 1 cpu are submitted at the origin.
- A *spaced* workload, where jobs of the same nature are submitted at regular intervals. 200 delays with duration 170s, and requesting 1 cpu are submitted every 10s.
- A *realistic* workload, which is a trace extracted from the Karlsruhe Institue of Technology ForHLR II System. It is composed of 49 jobs including both long running jobs (truncated up to an hour) and groups of smaller jobs often submitted simultaneously, spanning over 10 hours.

The first two workloads are straight forward and could be generated with the use of a plain text editor (understand `vim` and its macros). The third workload required more processing to be obtained.

Standard Workload Format processing

Batsim provides a tool to translate SWF files to its own json definition. It also works as a workload preprocessor, although we want to process SWF files very specifically to suit our needs which is why a custom script was implemented.

First, a trace in standard workload format (swf) was obtained on a web archive[1]. The chosen workload was `KIT-FH2-2016-1.swf` (we give the file name for the reader to find the workload on the archives) because it is the most recent and is relatively lightweight. It is composed of 114,355 jobs spanning over 19 months. Secondly, `evalys`[2] allowed us to extract a subset of this workload lasting for a given period of time and with a given mean utilization of the resources. We chose a period of 10h with 80% utilization of the resources so as to keep reasonable experiment durations – Later on we experiment with larger workloads to test out Batkube's limits in terms of scalability. The third step is translating this extracted workload to a `json` file that can be read by Batsim, which is done with a script written in Go.

After extracting this subset, we are left off with a workload containing jobs spanning up to 45h and using up to 24048 cpu (or cpu cores), which is undoable at our scale on our emulated cluster. We need to trim job durations as well as cpu usage, as we are limited in cpu by the host machine. This is done during the translation to the `json` format. The durations are trimmed down to a maximum of one hour and the cpu usages are normalized so the maximum amount of cpu requested equals the amount of cpu available per node on the host machine. Otherwise, the job would not be schedulable which would not present much interest. We end up with a trace composed of 39 jobs spanning over 10 hours, with a maximum job length of one hour and cpu usage ranging from 0 (excluded) to 5.9 (even though the host machine had 6 cores, Kubernetes scheduler did not allow for a job to be scheduled on a node it would use 100% resources of).

4.1.3 Studied platforms

The platform used for the first two workloads, *burst* and *spaced* is composed of 16 nodes each heaving one cpu. For the *realistic* workload however, we use a single node composed of six cores for the following reasons.

First, the host machines where the experiments were conducted had six cpu cores available. This means that if we want to be able to run an emulated cluster equivalent to this platform we can't exceed six cpu per node. We use the maximum amount of available cores in order so as not to obtain too low values when normalizing the resource requests on the jobs. Indeed, Kubernetes only allows for a precision of 1 milicpu, so any value bellow that is not considered a significant number. Normalizing on six cpus instead of one allows us to get more significant number. Also, only one node gives us a satisfactory overall resource usage: with more than one node one resource is almost always available making the scheduling operations trivial.

4.2 Study of the simulator parameters

The simulator has a few parameters that impact the simulation speed and accuracy. The objective is to study the effects of these parameters on the simulation to better understand the scheduler behavior when running in coordination with Batkube.

The objective here is to fine tune the parameters in order to find a compromise between accuracy and scalability. We want to know which combination lead us to the most stable results, while keeping simulation

[1] https://www.cs.huji.ac.il/labs/parallel/workload/logs.html

[2] https://github.com/oar-team/evalys

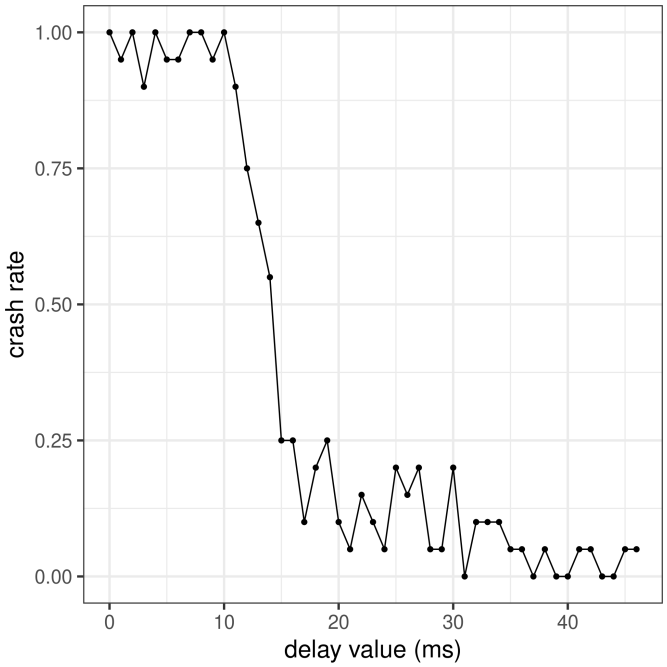


Figure 4.2 – Crash rate of the simulations against minimum delay.

time as low as possible. In these simulations, the scheduling time is very low and was neglected. This would not be the case with more complicated workloads involving complex scheduling mechanisms.

The parameters are:

- The *minimum delay* we have to spend waiting for the scheduler.
- The *timeout* value when waiting for scheduler decisions.
- The *maximum simulation time step*, which is the maximum amount of time Batsim is allowed to jump forward in time.

We first study these parameters one by one by fixing the other parameters to some other value, then we study what effects these parameters have in respect to one another, and finally we conduct scalability experiments to test Batkube's stability and performances on large workloads.

4.2.1 Minimum delay

Earlier in the development of batkube we noticed that not leaving enough time to the scheduler each cycle lead it to crashes and deadlocks, ultimately failing the simulation. This time is independent from any decision making we would receive from it which is why it is called *minimum wait delay* instead of a plain *timeout* - which is in fact another parameter we will study later.

For each workload, we compute the crash rate every 5ms, from 0ms to 50ms. Each point is made by running the simulation 15 times and recording the exit code as well as the simulation time. The other parameters are: `timeout=20ms`; `max-simulation-timestep=20s`. As we will see later those do not offer acceptable simulation results but they allow us to run prompt simulations, as accuracy do not concern us here.

As we can see on figure 4.2, the crash rate decreases dramatically as soon as the minimum delay reaches a certain threshold, which here is 10ms. This crashing issue though was resolved with an update of the scheduler : the success rate flattens out at 100% – or around 100%. Then, with earlier versions of the scheduler, the user may have to adjust the minimum delay in order to run simulation smoothly.

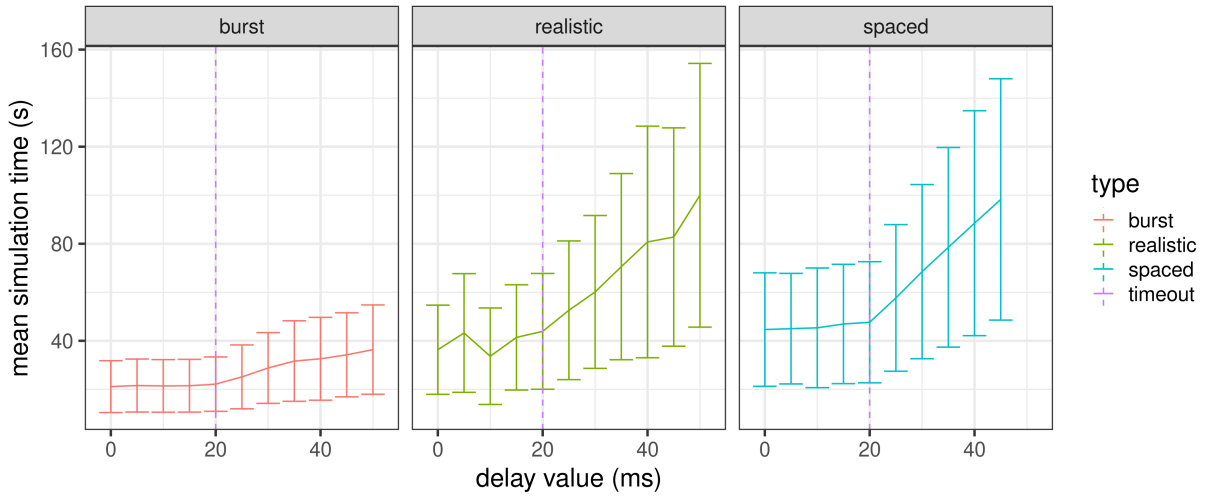


Figure 4.3 – Mean duration of the simulations (in case of success) against minimum delay. Error bars show confidence intervals at 95%.

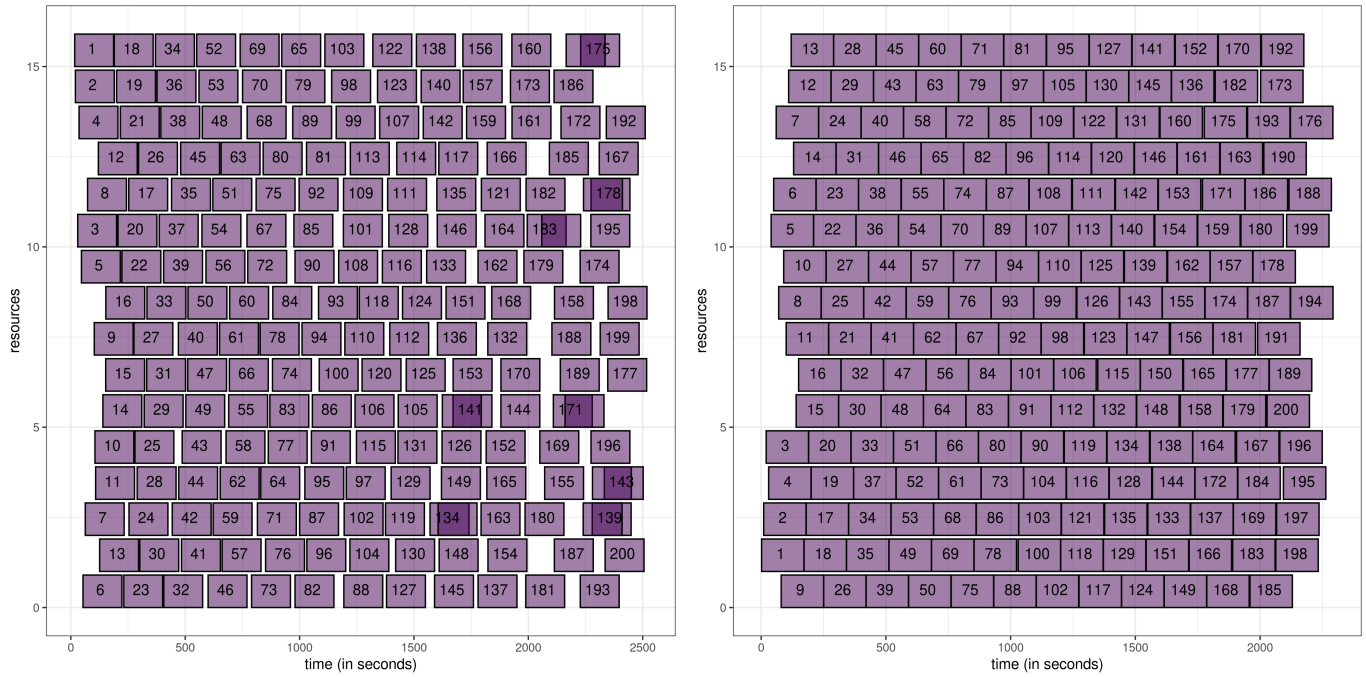
We also observe on figure 4.3 – which was made with an updated scheduler – a prompt increase in simulation time from delay value 20ms. This is due to the fact that the *timeout* value is 20ms, which is reached most of the time because the vast majority of the calls to the scheduler do not result in a decision making. After this value, we notice a direct correlation between *minimum delay* increase and simulation time increase. It follows that the best choice for the *minimum delay* now is zero, and we will use this value for the rest of the experiments.

4.2.2 Timeout

This value is the maximum amount of time we leave for the scheduler to react. A `timeout value` not large enough may lead to inaccuracies in the simulation: for example, if the scheduler needs 30ms to make a decision upon reception of a message, and the value of the timeout is 20ms, Batkube will receive the decision on the next cycle which may happen several dozens of seconds later (depending on the *maximum simulation time step* value). This phenomenon appears very clearly in figure 4.4 which represents a simulation run on the spaced workload (other parameters are `min-delay=0` and `max-simulation-timestep=1000s` to reduce simulation time). Additionally, this graph shows us that low timeout values induce incorrect behavior from the scheduler such as over allocation of resources: darker areas show jobs overlapping on the same resource, even though all resource have a capacity of one cpu and jobs request one full cpu. On the other hand, a *timeout value* too large will induce longer simulation times. Indeed, once the simulator was given enough time to process a message, any time following is spent idling. We want to measure which `timeout value` is just enough for the scheduler to be able to make a response without spending any time idling.

We run each workload with a `timeout value` ranging from 0ms to 100ms, –except for the realistic workload because simulation times were already consequent from 50ms – with a step of 1ms. Each time we measure the duration of the simulation as well as the makespan and the mean waiting time. The latter two will enable us to compare the results against the emulated results in order to estimate the accuracy of the simulation. The other parameters are set to: `min-delay=0ms`, `max-simulation-timestep=20s`

As we expected, a *timeout value* too low results in the scheduler missing a few cycles each time it wants to communicate a decision making, thus increasing the makespan and mean waiting time. As the *timeout* increases, it reaches a point where the scheduler consistently sends decisions in the same cycle as the one where it has received the message that triggered the decision making. After this point though the curves keep decreasing ever so slightly, showing that the gaps keep receding afterwards. However, the gain in accuracy is shallow and considering that there is, again, a direct correlation between the *timeout value* and the simulation time, it is desirable to keep this value at the limit where the results start to stabilize. In this



(a) Timeout value: 5ms

(b) Timeout value: 50ms

Figure 4.4 – A low timeout value results in the apparition of delays in the scheduling process. Both gantt charts were made with the *spaced* workload.

case, according to figure 4.5, `timeout-value=50ms` seems like a decent compromise between accuracy and scalability.

We observe slightly different behaviors with the realistic workload. First, the results stabilize earlier than the other workloads as we can see on the first graph of figure 4.5b. Also, the mean waiting time and simulation time values are much more scattered than the other two workloads: it seems like two curves appear on the last two graphs. Since the simulation is not deterministic, and that the number of jobs in the realistic workload is fairly low, a slight difference in scheduling decisions has a notable impact on the simulation.

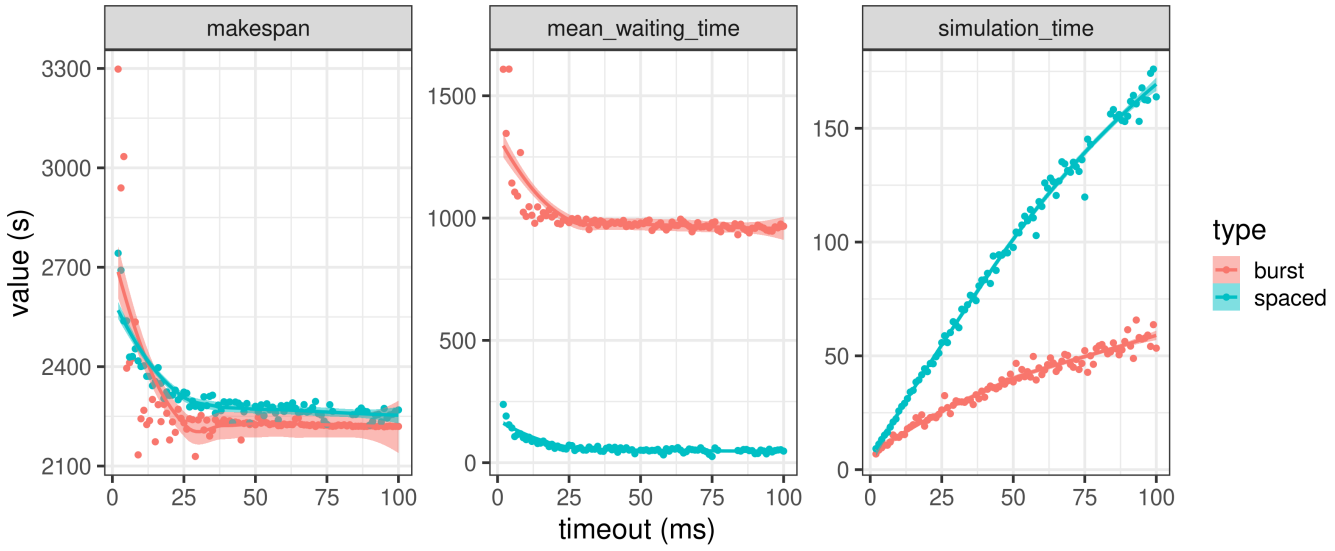
4.2.3 Maximum simulation time step

Having a high maximum time step value will allow Batsim to jump forward further in time. This may result in skipping scheduler decisions that could have been made in the mean time, delaying them to when Batsim decides to wake up. We expect increasing this value to have an analogous effect to the timeout value: higher simulation speed, but also decreased accuracy due to gaps (delays) in the decision process.

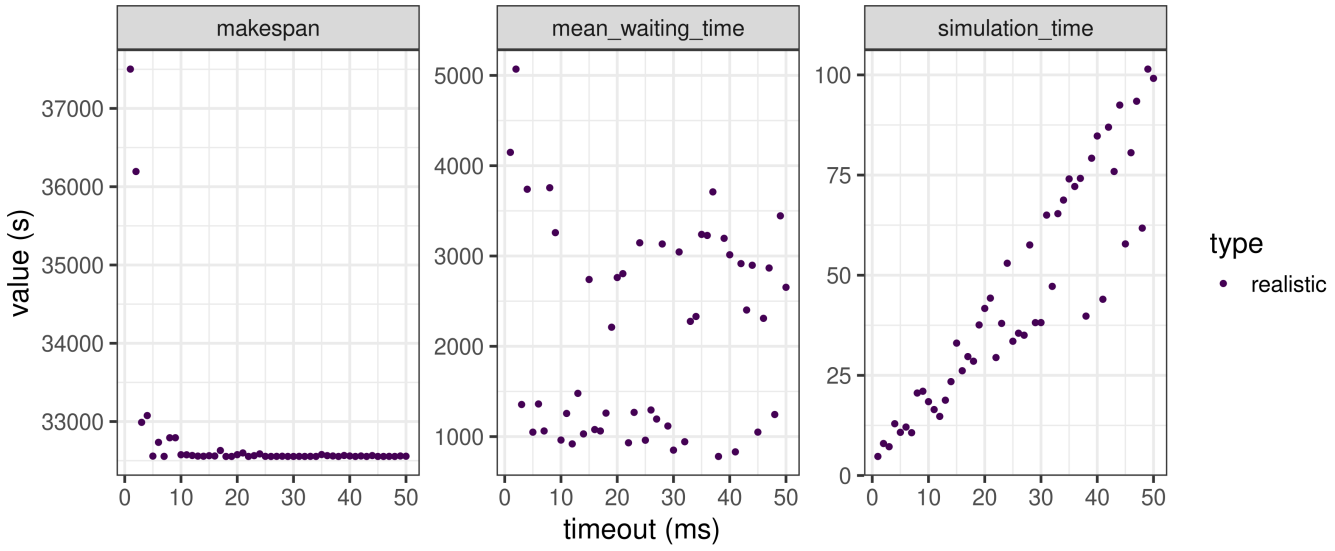
To experiment with the maximum time step effect on the results we obtain, we run each workload with different values of `max-simulation-timestep` following a logarithmic scale. The other parameters are fixed to `min-delay=0s`, `timeout=50ms`. Also, the `base-simulation-timestep` was lowered to 10ms in order to test lower values of the maximum timestep (compared to the previous 100ms).

As expected, the simulation time decreases drastically when the maximum timestep increases. Still, this value reaches a minimum eventhough the maximum timestep keeps increasing. This happens because Batsim events are only so far appart in the simulation, and Batsim will always wake up before the maximum timestep is reached.

Surprisingly, we observe that the increase in the maximum simulation timestep has no noticeable effect on the other metrics for the burst and realistic workloads. The makespan and mean waiting time flatten out regardless of the increase in the timestep for the burst workload on figure 4.6a. Regarding the realistic



(a)

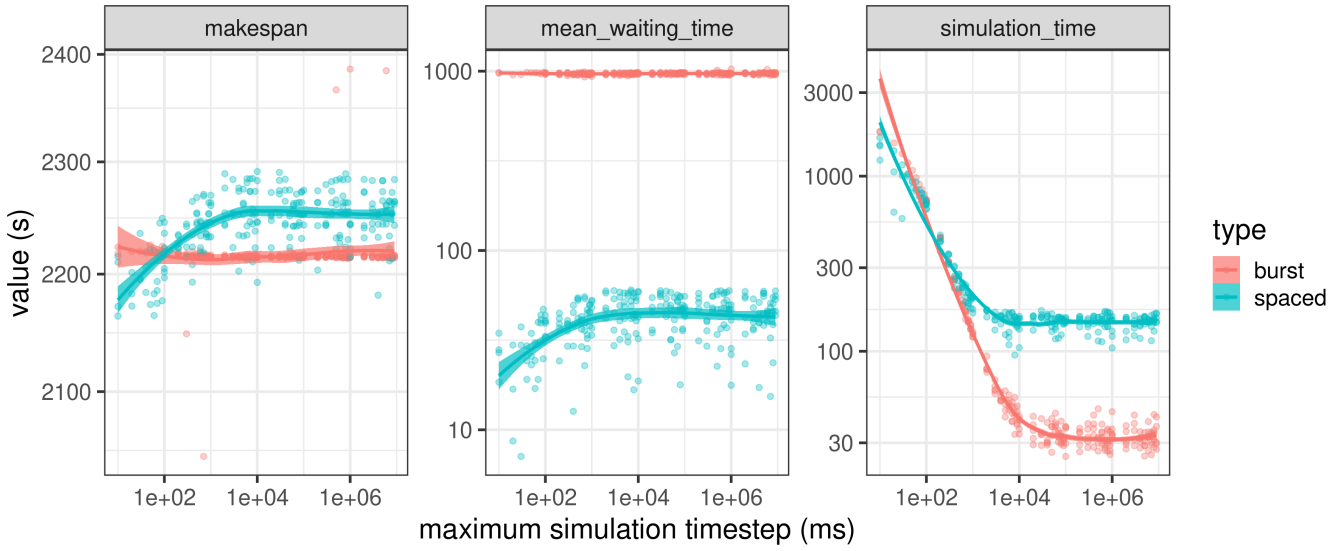


(b)

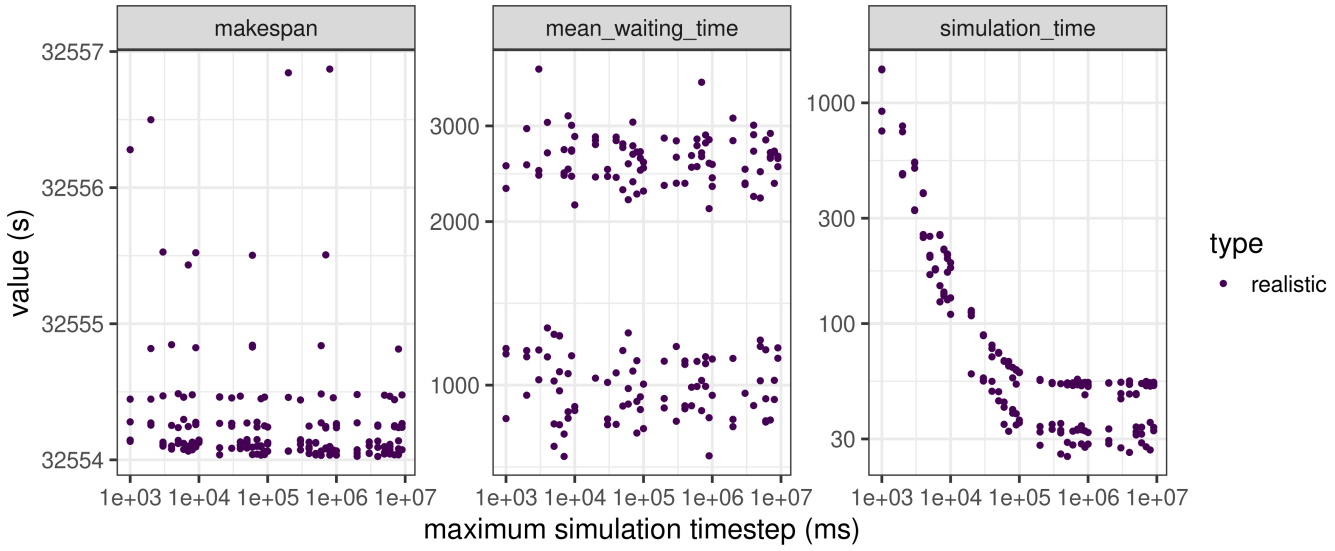
Figure 4.5 – Effect of the timeout value on the simulation.

workload, we still observe on figure 4.6b the different trajectories we noticed in the last experiment, however the distribution does not change according to the maximum timestep either.

The spaced workload however is impacted by the increase in the timestep. Plotting a Gantt chart at different values of the simulation timestep allow us to visualize why this happens, and it appears that lower values of the maximum timestep induce more errors in the simulation. We believe that this is caused by the drastic increase in the number of exchanges with the scheduler, which in turn increases the likelihood of generating errors in the scheduler. We conclude that having a low maximum timestep is never desirable, as it increases both simulation time and scheduler errors. Furthermore, a timestep value as high as 1000s in this case corresponds to no limits on Batsim jumps in time so one could argue that this parameter can simply be discarded. While it is the case here with these simple workloads and the absence of jobs preemption, it might have an impact when the scheduler is susceptible to make scheduling decisions on jobs that are already scheduled.



(a)



(b)

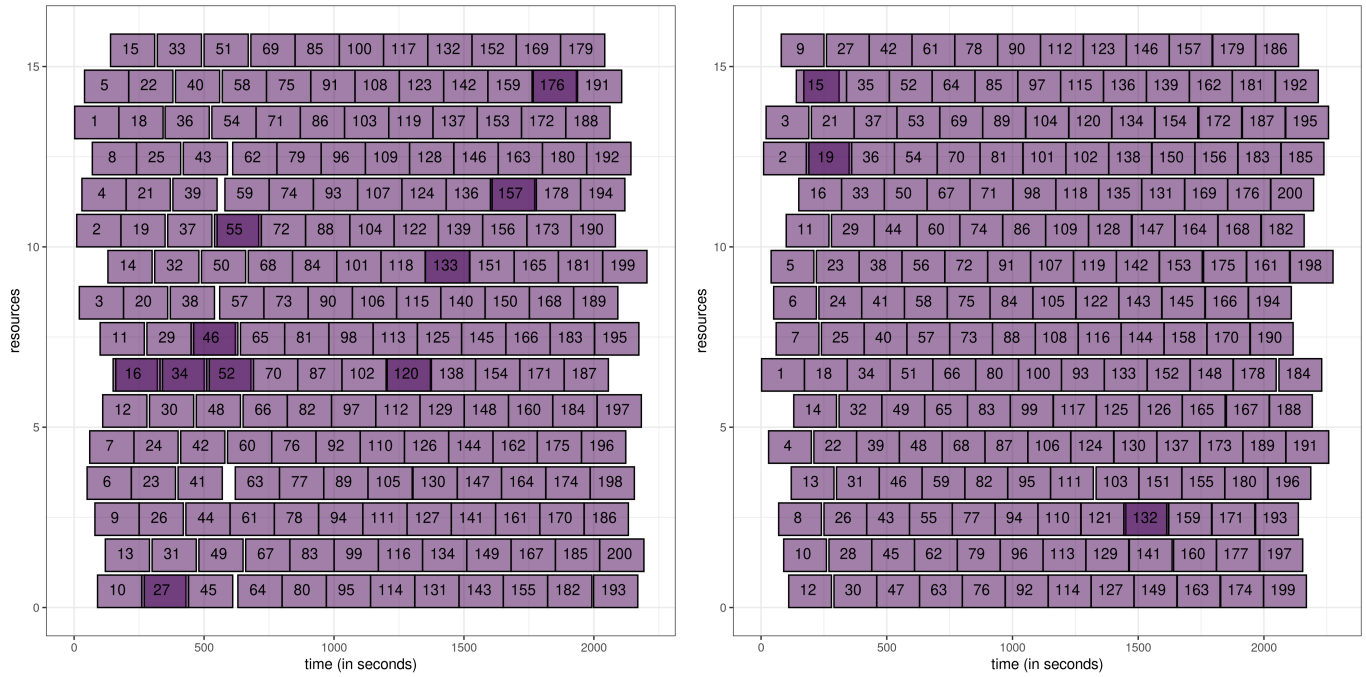
Figure 4.6 – Effect of the maximum simulation timestep on the simulation

4.3 Deviation of the simulation with reality

Thanks to our emulated cluster we were able to run those same workloads on real clusters that presented the same characteristics as the platforms we used to run our simulations. The *burst* and *spaced* workloads were run on 16 nodes each having one allocatable cpu, and the *realistic* workload was run on a single node composed of 6 allocatable cpus. The simulated workloads were run with the following parameters : `min-delay=0ms`, `timeout-value=50ms`, `max-simulation-timestep=1000s`. Each emulated workload was run 10 times (except for the realistic workload which was only run once) and simulated workloads were run 20 times each. Here are the deviations we observed between the simulation and the emulation.

As we saw earlier the scheduler makes errors in the simulations by allocating pods on nodes that are already full. Also, the model is incomplete and the simulator does not account for the time Kubernetes takes to pull the containers from the Internet. For these reasons we except simulated makespan and mean waiting time to be lower than the emulated ones.

We observe a neat difference between simulated and emulated makespans for the burst and spaced workloads as expected: both times, the simulation finishes earlier than the emulation. The mean waiting



(a) Maximum simulation timestep value: 30ms

(b) Maximum simulation timestep value: 1000s

Figure 4.7

	makespan				**mean waiting time**			
	emulated		simulated		emulated		simulated	
	μ	σ	μ	σ	μ	σ	μ	σ
burst	2467	28.3	2215 (-252)	0.508	1077	10.6	970 (-107)	12.6
spaced	2468	5.14	2257 (-211)	16.9	146	1.67	48.1 (-97.9)	9.44
realistic	32556	-	32555 (-1)	1.30	2884	-	2020 (-864)	950

Table 4.1 – Difference between emulated and simulated values. All values are in seconds. The difference with emulated results is indicating with the simulated means.

times of the burst workload seem to follow this same principle as the simulated values are slightly lower than emulated ones, although it doesn't seem to be the case for the spaced workload. Indeed, the scheduler performed drastically better in the simulated scenario. We don't know the cause of this. As for the realistic workload, the simulated metrics correspond exactly to the ones measured in the emulated cluster. At first glance it seems that the waiting times do not correspond, however we noticed during our experiments that the scheduler seemed to follow two differents scenarios for this workload leading to two different values of the mean waiting time. This is especially obvious on figure 4.6b where we notice two groups on the far right graph. Considering this, we can make the assumption that the real waiting times values either $\mu + \sigma$ or $\mu - \sigma$, that is to say, in this case, 2970 or 1070. We can then consider 2970 and 2884 to be part of the same group since they are fairly close to each other and assume that the simulation was accurate. Running more experiments with the emulated realistic workload would give us better insights, unfortunately we did not have enough time on our hands. In that regard, we would have benefited from a realistic workload composed of more jobs and spanning over a shorter period of time. This is an overlooked part of our experimentations.

4.4 Discussion and future work

First, it is important to note some fundamental differences between grid computing and cluster computing, differences because of which we had to put aside large components of SimGrid models. Parallel computing

relies on the ability to schedule a task on multiple nodes. The different processes communicate with each others thanks to protocols such as the *Message Passing Interface (MPI)*. With Kubernetes however one cannot schedule one pod on multiple nodes because it is not meant to run highly parallel tasks. Furthermore, it would break the paradigm of encapsulation of containers within nodes which is part of what make Kubernetes a powerful tool for cluster administration. Therefore, we are forced to ignore one of the major features of SimGrid which is the simulation of parallel tasks and MPI applications (SimGrid has an entire module for the simulation of MPI applications named SMPI [10]).

Batkube, as a simulator, has limited features, however it is functional and proved that it is possible to adapt any Kubernetes scheduler to Batsim. It supports delay jobs that can request resources such as cpu or memory. In its current state the API supports the default Kubernetes scheduler, which is a very capable scheduler and is widely used in the industry. For now, it has several flows that need to be improved upon.

The API of Batkube needs further development in order to correctly support other Kubernetes schedulers. Partial or wrong implementations of the endpoints lead to numerous bugs and misbehavior of the scheduler throughout the development, and while the most critical issues have been resolved, some errors remain like the over allocation of resources.

The obligatory synchronization of the time between the scheduler and Batsim make this simulator not viable for large scale simulations. As we saw in the experiments, a simple workload with 200 delay jobs of 170 seconds submitted every 10 seconds already requires nearly two minutes of simulation. This is due to the frequent exchanges between Batsim and Batkube over the json interface, and the fact that we wait up to *timeout value* every exchange. This is why we did not conduct any scalability experiment. This kind of experiment would require more time, but also more resources. Indeed, we observed that the Kubernetes scheduler require more memory on long running experiments and hindered our capacity of running long simulations. Also, it would simply not be relevant because scalability is not our objective here.

Batkube simulation capabilities can be expanded on by implementing the rest of Batsim own capabilities. A single pod cannot be scheduled on several nodes but one node can still have a theoretically unlimited capacity in resources, and pods inside one same node can communicate with each others and work together to execute a parallel task[3]. With a bit of tweaking in the API one could model simple parallel tasks. Batsim also supports energy models which could become a critical metric in the near future, considering the climate urgency. Storage resoures can be added as well for a more complete simulation of an online service with a database. Finally, Batsim supports external events such as resource failures and other user defined events.

With a deeper understanding of the Kubernetes api-server internals one could support any other scheduler and compare them against each others. Also, Kubernetes allowing the use of multiple schedulers, studies could be conducted on their interaction with each other and how their combinations could improve the general scheduling performances on various system topologies. Extensive studies can be realized on schedulers competing with each others on the same resources, justifying the need for a simulator. Also, since the api-server is the central component of Kubernetes, any other tool of the k8s ecosystem could be supported given a more complete API. During the development we already made use of their command line interface, `kubectl`, to monitor the resources and we could imagine plugging some monitoring tools to the simulator such as `kubetop`. All this offer the perspective of a fully fledged Kubernetes simulator based on the sound models of SimGrid.

[3] https://kubernetes.io/docs/concepts/workloads/controllers/job/

Appendices

.1 Reproducing the experiments

All scripts can be found in the batkube-test[4] repository, under the `scripts` repository. The exact commands used to run Batkube, the scheduler and Batsim are described in the shell scripts. Running an experiment on an emulated cluster can be done with `real-clutser-experiment/main.go`, after starting a cluster with `scripts/startup-cluster.sh`. Both of these scripts are shipped with a help section. Finally, `swf-translate/swf-to-json.go` is the script used to process swf workloads. It also has a help section.

Batsim is run with option `enable-compute-sharing`: for a reason unknown, Kubernetes scheduler tends to over allocate resources in some cases (especially with smaller jobs) which makes Batsim crash if this option is disabled. Then, we must allow compute sharing even when it is not expected in order to capture the scheduler behavior as precisely as possible.

.2 Batsim events handled by Batkube

Here are all the Batsim events that are handled by Batkube.

From Batsim to the scheduler

SIMULATION_BEGINS contains information about the available resources in the cluster, with Batsim's configuration.

SIMULATION_ENDS is sent at the very end of the simulation: all jobs have finished, and no more jobs are left in the queues. Batsim exits on this message.

JOB_SUBMITTED notifies the scheduler that a new job has been submitted. It contains information about the job type, id and specifications. We only consider jobs of type *delay* to simplify the models. Delay jobs specifications boil down to the delay length, to which we add resource requests.

JOB_COMPLETED notifies the scheduler that a job has ended, specifying the reason for it. We only consider situations where all jobs complete correctly. Their state is then always COMPLETED_SUCCESSFULLY in our case.

REQUESTED_CALL is an awnser to a CALL_ME_LATER event sent by the scheduler.

From the scheduler to Batsim

CALL_ME_LATER is an incentive from the scheduler for Batsim to wake up at a certain timestamp. When the timestamp is reached in the simulation, Batsim will send a REQUESTED_CALL to the scheduler. In our case, this particular exchange will serve as the base for time synchronisation between the scheduler and Batsim.

EXECUTE_JOB is sent when the scheduler has made a decision. It contains the id of the job at stake and the id of the resources it has been scheduled to.

[4] github.com/oar-team/batkube-test

```
{
  "now": 1024.24,
  "events": [
    {
      "timestamp": 1000,
      "type": "EXECUTE_JOB",
      "data": {
        "job_id": "workload!job_1234",
        "alloc": "1 2 4-8",
      }
    },
    {
      "timestamp": 1012,
      "type": "EXECUTE_JOB",
      "data": {
        "job_id": "workload!job_1235",
        "alloc": "12-100",
      }
    }
  ]
}
```


Figure .8 – Example of a Batsim message

Bidirectional

NOTIFY is used to send some information to the other peer. In our case, we use the NOTIFY containing no_more_static_job_to_submit to determine if the simulation has ended: knowing that there are no more jobs susceptible to be scheduled allow us to fast forward to the end of the simulation, thus saving execution time.

.3 Batsim integration with Kubernetes: other options

Here are the options that were considered but not chosen for Batsim integration with the Kubernetes ecosystem.

In between the api and the kubelets

This is the lowest level option. We position the simulator so as to simulate just the infrastructure and avoid tampering with Kubernetes resource management, which is done in their API. This approach would allow us to effortlessly use any Kubernetes scheduler once their API is supported by Batkube, and potentially produce the most accurate results. However, interactions between the kubelets and the API are not documented because the typical user is not supposed to have to deal with this part of Kubernetes. This would hinder the development of Batkube because a reverse engineering process would be required beforehand to understand the intricacies of internal Kuberenetes exchanges.

```
{
    "nb_res": 1,
    "jobs": [
        {"id":"1", "subtime":0, "res": 1, "profile": "delay10"},
        {"id":"2", "subtime":3.4, "res": 1, "profile": "delay10"}
    ],
    "profiles": {
        "delay10": {
            "type": "delay",
            "delay": 10,
            "scheduler": "default",
            "cpu": "1.5",
            "memory": "500Mi"
        }
    }
}
```

Figure .9 – Example of a Batsim workload

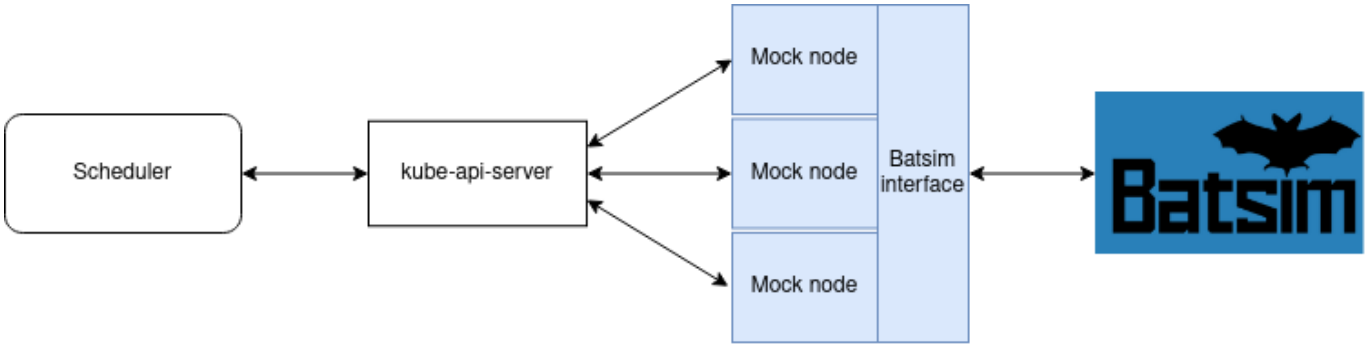


Figure .10 – Mocking the cluster itself.

Custom client-go

Most Kubernetes schedulers rely on client-go[5], which is a Go client for the api-server. It is a library implementing various tools to help schedulers converse with the API. By altering this client and patching schedulers so they use our client instead, we can make it exchange with Batsim instead of the API.

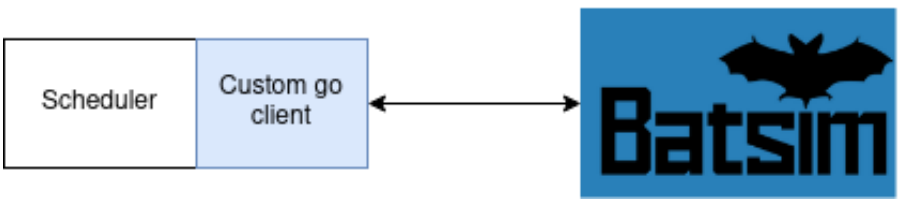


Figure .11 – Custom Go client to redirect scheduler communications to Batsim

Contrary to the kubelets, client-go is a user interface and therefore it is documented, facilitating reverse engineering of its source code. Still, it represents thousands of lines of code and altering it to our needs would not be an easy feat. The other drawback to this approach is that Batkube would only support schedulers written in Go and making use of client-go, although this should not be an issue as the only kubernetes scheduler we could find that does not rely on client-go is a toy scheduler written in bash [15].

[5]https://github.com/kubernetes/client-go

Partial reimplementation of the API

Re-implementing the API offers a middle ground between the low level and undocumented solution of the mock nodes, and the higher level and technically challenging solution of a client-go fork. Again, there are several options here.

A partial reimplementation of the API would save us the task of building a new API from the ground up. However this would imply digging deep into the api-server code in order to understand how the api is organized and what code we would have to alter. In the end, it is easier to simply build a new API, since there are tools to help us generate it from its specification.

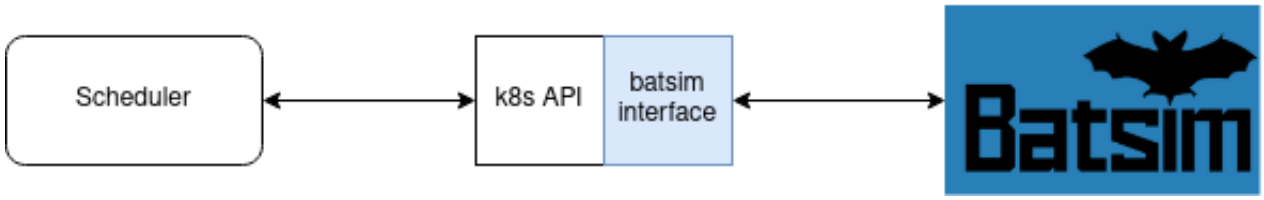


Figure .12 – Partial reimplementation of the api-server.

.4 Time interception: the C library approach

An attempt was first made to patch a custom C library, which is the lowest level solution. Going for the low level solution would truly redirect all calls to machine time which is something we can not guarantee with the second option, as we explain in section 3.5.2. This approach proved challenging due to circular dependency issues and was ultimately abandoned. We opted for the second option which consist of modifying Go source code, which requires some additional work to patch the schedulers but was actually easier to implement.

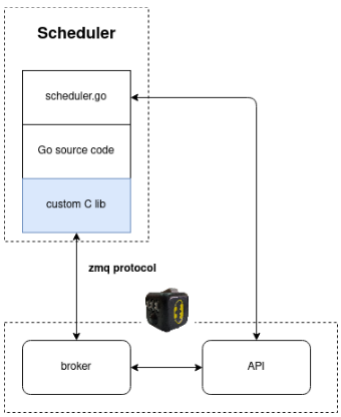


Figure .13 – Option A: patching the C library

Bibliography

[1] Anders Andrae. "Total consumer power consumption forecast". In: *Nordic Digital Business Summit* 10 (2017).

[2] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. "The datacenter as a computer: Designing warehouse-scale machines". In: *Synthesis Lectures on Computer Architecture* 13.3 (2018), pp. i–189.

[3] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. "OverSim: A scalable and flexible overlay framework for simulation and real network applications". In: *2009 IEEE Ninth International Conference on Peer-to-Peer Computing*. IEEE. 2009, pp. 87–88.

[4] A. M. Beltre et al. "Enabling HPC Workloads on Cloud Infrastructure Using Kubernetes Container Orchestration Mechanisms". In: *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*. 2019, pp. 11–20.

[5] J. Blazewicz, J.K. Lenstra, and A.H.G.Rinnooy Kan. "Scheduling subject to resource constraints: classification and complexity". In: *Discrete Applied Mathematics* 5.1 (1983), pp. 11 –24. ISSN: 0166-218X. DOI: https://doi.org/10.1016/0166-218X(83)90012-4. URL: http://www.sciencedirect.com/science/article/pii/0166218X83900124.

[6] Rajkumar Buyya and Manzur Murshed. "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing". In: *Concurrency and Computation: Practice and Experience* 14 (Nov. 2002). DOI: 10.1002/cpe.710.

[7] Rodrigo Calheiros et al. "CloudSim: A Novel Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services". In: (Apr. 2009).

[8] Y. Caniou and J. Gay. "Simbatch: An API for Simulating and Predicting the Performance of Parallel Resources Managed by Batch Systems". In: Apr. 2009, pp. 223–234. DOI: 10.1007/978-3-642-00955-6_27.

[9] Henri Casanova et al. "Developing Accurate and Scalable Simulators of Production Workflow Management Systems with WRENCH". In: *Future Generation Computer Systems* 112 (2020), pp. 162–175. DOI: 10.1016/j.future.2020.05.030.

[10] Henri Casanova et al. "Versatile, Scalable, and Accurate Simulation of Distributed Applications and Platforms". In: *Journal of Parallel and Distributed Computing* 74.10 (June 2014), pp. 2899–2917. URL: http://hal.inria.fr/hal-01017319.

[11] Bruno Donassolo et al. "Fast and Scalable Simulation of Volunteer Computing Systems using Simgrid". In: June 2010, pp. 605–612. DOI: 10.1145/1851476.1851565.

[12] Pierre-François Dutot et al. "Batsim: a Realistic Language-Independent Resources and Jobs Management Systems Simulator". In: *20th Workshop on Job Scheduling Strategies for Parallel Processing*. Chicago, United States, May 2016. URL: https://hal.archives-ouvertes.fr/hal-01333471.

[13] David Flores. “SimBA: A discrete-event simulator for performance prediction of volunteer computing projects”. In: *ETD Collection for University of Texas, El Paso* (Jan. 2006). DOI: 10.1145/1188455.1188629.

[14] Cristian Galleguillos, Zeynep Kiziltan, and Alessio Netti. “AccaSim: An HPC Simulator for Workload Management”. In: *High Performance Computing*. Ed. by Esteban Mocskos and Sergio Nesmachnow. Cham: Springer International Publishing, 2018, pp. 169–184. ISBN: 978-3-319-73353-1.

[15] Justin Garrison. *bashScheduler*. URL: https://github.com/rothgar/bashScheduler (visited on 08/18/2020).

[16] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. “LogGOPSim - simulating large-scale applications in the LogGOPS model”. In: vol. 10. Jan. 2010, pp. 597–604. DOI: 10.1145/1851476.1851564.

[17] Dalibor Klusáček and Šimon Tóth. “On Interactions among Scheduling Policies: Finding Efficient Queue Setup Using High-Resolution Simulations”. In: *Euro-Par 2014 Parallel Processing*. Ed. by Fernando Silva, Inês Dutra, and Vítor Santos Costa. Cham: Springer International Publishing, 2014, pp. 138–149. ISBN: 978-3-319-09873-9.

[18] Dalibor Klusáček and Hana Rudová. “Alea 2 - Job scheduling simulator”. In: *SIMUTools 2010 - 3rd International ICST Conference on Simulation Tools and Techniques* (Jan. 2010), p. 61. DOI: 10.4108/ICST.SIMUTOOLS2010.8722.

[19] Derrick Kondo. *SimBOINC: A simulator for desktop grids and volunteer computing systems*. 2007.

[20] *kube-batch*. URL: https://github.com/kubernetes-sigs/kube-batch (visited on 08/18/2020).

[21] Mikyoung Lee, Sungho Shin, and Sa-Kwang Song. “Design on distributed deep learning platform with big data”. In: (2017).

[22] Arnaud Legrand. “Scheduling for large scale distributed computing systems: approaches and performance evaluation issues”. PhD thesis. 2015.

[23] N. Liu et al. “YARNsim: Simulating Hadoop YARN”. In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015, pp. 637–646.

[24] Xin Liu, Huaxia Xia, and Andrew Chien. “Validating and Scaling the MicroGrid: A Scientific Instrument for Grid Dynamics”. In: *J. Grid Comput.* 2 (June 2004), pp. 141–161. DOI: 10.1007/s10723-004-4200-3.

[25] Alberto Montresor and Márk Jelasity. “PeerSim: A Scalable P2P Simulator”. In: *Proc. of the 9th Int. Conference on Peer-to-Peer (P2P’09)*. Seattle, WA, Sept. 2009, pp. 99–100.

[26] *OAR next generation*. URL: http://oar.imag.fr/start (visited on 08/18/2020).

[27] Simon Ostermann, Radu Prodan, and Thomas Fahringer. “Dynamic Cloud provisioning for scientific Grid workflows”. In: Nov. 2010, pp. 97 –104. DOI: 10.1109/GRID.2010.5697953.

[28] Peter Brucker, Sigrid Kunst. *Complexity results for scheduling problems*. June 29, 2009. URL: http://www2.informatik.uni-osnabrueck.de/knust/class/ (visited on 06/10/2020).

[29] Millian Poquet. “Simulation approach for resource management”. Theses. Université Grenoble Alpes, Dec. 2017. URL: https://tel.archives-ouvertes.fr/tel-01757245.

[30] 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 - New York, NY, USA, 2017, 9–10. ISBN: 9781450351690. DOI: 10.1145/3154842.3154845. URL: https://doi.org/10.1145/3154842.3154845.

[31] Nikolay Simakov et al. “A Slurm Simulator: Implementation and Parametric Analysis”. In: Jan. 2018, pp. 197–217. ISBN: 978-3-319-72970-1. DOI: 10.1007/978-3-319-72971-8_10.

[32] Mustafa Tikir et al. “Psins: An Open Source Event Tracer and Execution Simulator for MPI Applications”. In: Aug. 2009, pp. 135–148. DOI: 10.1007/978-3-642-03869-3_16.

[33] Boris Tvaroska. “Deep Learning Lifecycle Management with Kubernetes, REST, and Python”. In: Santa Clara, CA: USENIX Association, May 2019.

[34] J. D. Ullman. “NP-Complete Scheduling Problems”. In: *J. Comput. Syst. Sci.* 10.3 (June 1975), 384–393. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(75)80008-0. URL: https://doi.org/10.1016/S0022-0000(75)80008-0.

[35] Pedro Velho et al. “On the Validity of Flow-Level Tcp Network Models for Grid and Cloud Simulations”. In: *ACM Trans. Model. Comput. Simul.* 23.4 (Dec. 2013). ISSN: 1049-3301. DOI: 10.1145/2517448. URL: https://doi.org/10.1145/2517448.

[36] Hidehito Yabuuchi. *k8s-cluster-simulator*. URL: https://github.com/pfnet-research/k8s-cluster-simulator (visited on 08/16/2020).

[37] Yuan Chen. *A Toolkit for Simulating Kubernetes Scheduling at Scale*. URL: https://static.sched.com/hosted_files/kccncna19/97/KubeCon_JoySim.pdf. Nov. 2019.

[38] Gengbin Zheng, Gunavardhan Kakulapati, and L.V. Kale. “BigSim: a parallel simulator for performance prediction of extremely large parallel machines”. In: May 2004, pp. 78–. ISBN: 0-7695-2132-0. DOI: 10.1109/IPDPS.2004.1303013.