Minimizing Cluster Under-use using a Control-Based Approach

Quentin Guilloteau

Master of Science in Informatics at Grenoble
Master Informatique
Specialization Digital Infrastructure

Minimizing Cluster Under-use using a Control-Based Approach

Quentin Guilloteau

June 25, 2020

Research project performed at INRIA/LIG

Under the supervision of:
Eric RUTTEN (CTRL-A) & Olivier RICHARD (DATAMOVE)

Defended before a jury composed of:
Thomas ROPARS (External Expert)
Martin HEUSSE (Examiner)
Bruno RAFFIN (President)

June 2020

Abstract

High Performance Computing (HPC) systems have become increasingly more complex in the last decade. Their behaviour concerning their performance and power consumption make them less predictable. This unpredictability requires more and more runtime management. We believe that the field of Control Theory can bring a new approch to the administration of such complex systems. This project integrates an autonomic approach to the CiGri middleware, a grid application to manage large sets of multi-parametric tasks.

This work presents two distinct contributions. In a first time, we present a solution to minimize cluster under-utilization while controlling the load of the fileserver. We then investigate the case of reproducibility for our studied distributed system.

Acknowledgement

I would like to thank all the people who contributed directlty and indirectly to my work:

- My supervisors, Eric Rutten and Olivier Richard, for all the advise and precious guidance during this project.
- Bogdan Robu and Bashir Ibrahim for the very helpful insights on control theory
- Members of the CTRL-A team, especially the others M2 interns, Manal and Lucie, for the much appreciated coffee breaks
- and finally my family for the much needed long distance support during these historic times

Résumé

Les systèmes HPC (High Performance Computing) sont devenus de plus en plus complexes ces dernières années. Leur comportements par rapport à leur performances et consommation d'énergie, les rendent inpredictibles, ce qui nécessite davantage de management dit "en ligne". Nous croyons que le domaine de la théorie du contrôle peut apporter une nouvelle approche à la gestion de tels systèmes. Ce projet intègre une approche autonomique à CiGri, une application de grille de calculs gérant de larges ensembles de tâches multi-paramétriques.

Ce rapport présente deux contributions distinctes. Dans un premier temps, nous présentons une solution pour réduire la sous-utilisation d'un cluster tout en controllant la charge du fileserver. Ensuite, nous nous intéressons au charactère reproductible du système que nous étudions.

Contents

— 1 —
Introduction

1.1 High Performance Computing (HPC)

As sience continues to go forward, scientists from all fields have more and more complex computing workflows to complete. It can be for example earthquake simulations, climate predictions or astronomy rendering. These workflows are computational and data heavy, and cannot be processed on a single machine in a reasonable execution time.

This issue led to the development of distributed systems focused on performance. The idea is to cluster multiple machines through network to allow the user to perform parallel/distributed computations to accelerate their applications. These systems use high end components such as high speed network, fast ans large storage, and high perfomance processors to satisfy users' requests. Such a system is called High Performance Computing.

HPC systems have generally the following architecture:

- A frontend, allowing the users to make reservations of resources
- A server, managing the cluster
- A set of computing nodes, for the users to run their workflows

1.2 Problem

However, HPC systems have become more complex to manage due to the increases in the number of machines and in usage. The variations in their behaviour (e.g. in performance or in power consumption) make them harder to fully predict.

For example the performance of an application executing read and write operations on a fileserver, depends on the number of users querying this server. There is the same issue with the network of the cluster. If the network cannot handle large workload, the applications of the users will lose in performance. This can also have impacts on the energy consumption of the system, as the node CPUs are idle while the network is being used.

Thus, this situation requires runtime management which is usually done by humans.

In the previous example, we might want to schedule less users' jobs into the cluster resources to ensure that the fileserver is not overloaded. As it can be seen, such systems

are too large to carry on with being manually administraded and shall be automated to avoid slow or error-prone manipulations by humans.

Managing such phenomena shall be done online based on real-time measurements performed during execution. Control theory can bring a solution to this problem.

1.3 Contribution

This work builds on previous internships on the project. Past works have been more focused on a control approach of the problem as the previous interns were control theorists. During this internship, we looked into the computer science side of the project.

The contribution of this work is double. We present a new approach to control minimize the under-utilization of a cluster, while regulating the load of the fileserver. We also investigate the transition from a deployed approach with multiple machines to a contrainer approach on a single machine, and what this change means for the validity of the previous works and models.

1.4 Outline

The state of the art is presented in Chapter 2. The following Chapters contain the main contributions of this work. In Chapter 3, we present a new controller focused on regulating the load on the fileserver. Chapter 4 then presents a container based approach for the studied experimental setup in order to improve reproducibility, before concluding in Chapter 5.

— 2 —
State-of-the-Art

2.1 High Performance Computing

2.1.1 Resource Management in HPC systems

In the context of this work, we will focus on the Grid5000[1] system. Grid5000 is a large-scale and versatile test-bed for experimental computional research. It is composed of several clusters of machines around France (Grenoble, Nancy, Lille, Lyon, Rennes, Sophia). To submit a job into a machine, a user has first to open a SSH connection to one of the sites. Then she can submit her job to the scheduler. The scheduler used on the Grid5000 clusters is *OAR*[8]. Her job is then placed into the scheduler queue, which will assign it to resources on the cluster in a FIFO (First In First Out) fashion.

However, sometimes there might not be enough users on the cluster to use all the available resources, leaving some idle, which is a shortfall.

2.1.2 Usage of Idle Resources

The idea to use the idle resources of machines has be applied to personal computers. Projects like SETI@Home[2][6] have made this technology more popular. As of June 2020, the project averages a floating point operations rate of 72 TetraFLOPS[3] with more than 120.000 hosts [4]. The same team developed in 2004 the BIONC project [7]. It aims at providing a desktop grid infrastructure that can be used for different desktop grid applications.

In the actual context of COVID19, we can also cite the Folding@Home project[3]. The goal of this project is to harvest idle computational power for finding cures to diseases by running simulations of protein dynamics. The Folding@Home took on the Coronavirus with the support of thousands of volunteers giving their idle CPU times [2].

In the context of grid computing, as users reserve resources, it is possible that some resources are unsued at some point due to a small number of jobs submitted or to the fact that these resources have a configuration that does not meet the users' requests. For instance, if the user needs a GPU for her jobs and that every resources disposing of a

[1] `https://www.grid5000.fr/w/Grid5000:Home`
[2] Search for ExtraTerrestrial Intelligence
[3] Floating Point Operations Per Second

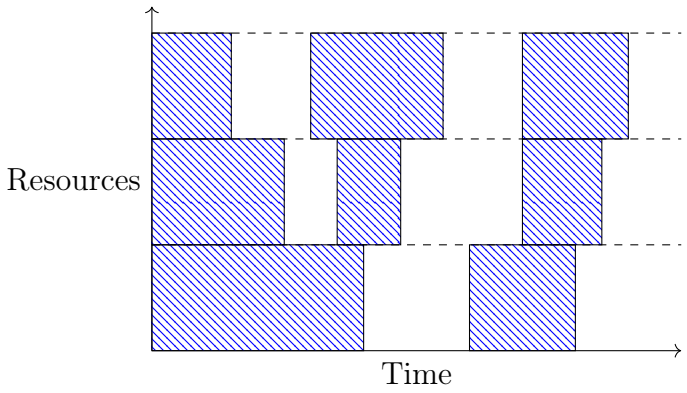

Figure 2.1: Example of usage of Resources in Grid Computing

GPU are being used, the user would have to wait, even if there are some idle resources with no GPU.

In this case, there is a bad usage of the resources. Accordingly, ways shall be found to make these idle resources more profitable. Figure 2.1 portrays an example of the use of resources. Each job running is represented by a hatched rectangle. When the resources are free, it is left blank. Even if this is an artificial example, we can sense that there is an under-utilization of the resources.

OurGrid[9] is one example. Small laboratories often do not have access to large cluster of machines to run their experiments. They mostly have a small number of machines. The goal of OurGrid is double: (i) to link machines from the small laboratories and (ii) to allow users to profit from idle resources of other machines if there are any resources idle. By doing so, they manage to scale up their application, but also to reduce the number of idle resources on registered machines.

2.2 Autonomic Computing

2.2.1 Presentation

As previously exposed, the field of Computer Science is a relatively young domain of application of Control Theory. Kephart and Chess [16] introduce the notion of Autonomic Computing as systems that can manage themselves given high-level objectives from administrators. They define four aspects of self-management in autonomic computing:

- Self-configuration: automated configuration follows high-level policies. The rest of the system adjusts automatically and seamlessly
- Self-optimization: the components and the system continually seek opportunities to improve their own performance and efficiency
- Self-healing: the system automatically detects and repairs software and hardware problems
- Self-protection: the system automatically defends against malicious attacks and cascading failures.

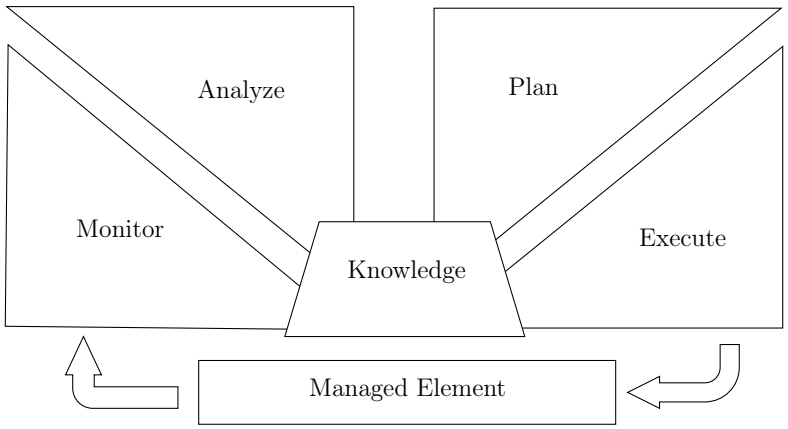

Figure 2.2: MAPE-K Loop

2.2.2 The MAPE-K Loop

The MAPE-K Loop represents the autonomic manager of an autonomic element [18]. A graphical representation of this loop can be found in Figure 2.2. The MAPE-K model is composed of five different parts:

- **M**onitor: In this phase, the sensors keep track of the fluctuations from the measured variables. Based on the changes, the Monitor must give notice to the next phase, Analyse phase, with relevant events based on the requeirements.
- **A**nalyse: Based on the requieriments and the notifications from the Monitor, the Analyse phase concludes if a change of Plan is required. If it is, the Analyser will notify the Planner with the new requirements. This might happen if the Quality of Service of the system is violated.
- **P**lan: Once the notification from the Analyser received, the Planner will select the adequate strategy based on the new requirements. The Planner will query the shared Knowledge. Once the strategy chosen, the control actions are computed and sent to the Executor.
- **E**xecute: After receiving the control actions by the Planner, the Executor will interprete each operation and execute them on the system.
- **K**nowledge: The Knowledge is shared among all the different phases. It can be used, for instance, to verify that a given state respects the requirements or to select the proper stategy based on the analysis.

2.2.3 Properties of Closed Loop Structures

Closed loop strutures, as opposed to open loop structures that do not take into account the feedback of the system, guarantee some properties [22]:

- Stability: A system is defined as stable if its output remains bounded stipulated that the inputs and disturbances also stay bounded. This is an essential property

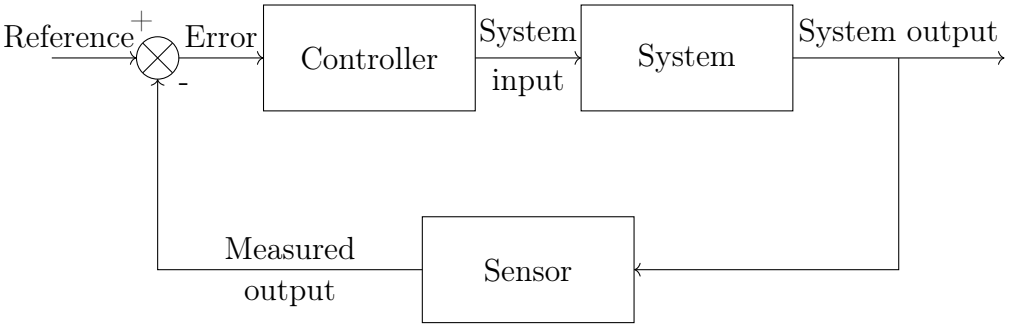

Figure 2.3: Example of feedback loop

to ensure that the controlled system behaves close to the requirements of the administrator. Well defined and tuned controller can even stabilize naturally unstable systems.

- Robustness: A fast rate of Monitoring compared to the dynamics of the controlled system, can offer a quick response to errors. Indeed, if an error is detected, the control loop can analyse, plan and execute a correction to rectify enough the trajectory of the system. Thanks to robustness, tracking objectives can be successfully performed over a large range of input trajectories without need for on-line re-tuning

2.2.4 Control Theory

Control Theory deals with the control of dynamical systems. The main tool of control theorists is the feedback loop. The concept behind such a loop is better explained by an example. Consider a house heating system, composed of three components:

- a temperature sensor
- a controller
- a heating system

Imagine that you want the temperature in your house to be 21°C (the reference value or set point). The sensor measures of the current temperature of the house. The controller compares the measured temperature with the reference value, and forwards an adapted input to the heating system according to the difference between the reference value and the measured temperature (i.e. the error). Periodically, the sensor measures the temperature to keep feeding the controller with the current status of the system. At some point, the house will be hot enough and the heating system shall shut down. But, as the heating system is shut down, the house may cool down. This system is a simple example of feedback loop, where the sensor will interface the heating system to regulate the temperature in the house. A general feedback loop is depicted in Figure 2.3.

There are different types of controllers used in feedback loops. They all have their advantages and drawbacks, which are discussed in Section A.1.3.

Control theorists have used feedback loops in many fields, but only started recently to apply them to Computer Science.

2.3 Reproducibility

2.3.1 Motivation

Computer Science is a young science. Other fields of Science have strong methodological standards. We could argue that Computer Science is growing too fast to define a standard that will not be quickly obsolete.

In 2015, Collberg et al. [10] made a study where they took papers that were submitted in conferences and journals with results that were backed up by code. Out of those 402 papers, only for 54% of them Collberg et al. were able to build the code or the original authors stated that their code would build with reasonable effort. This means that 46% of these papers were not reproducible. The non reproducible character could result from different causes:

- The code was not provided
- The code failed to build
- The experiment requires special hardware

Even if Reproducibility can be reduced to making experiements repeatable, it is more importantly making science verifiable and reusable. This also improves the confidence in the results.

2.3.2 Reproducibility in Computer Science

Mercier[19] defines five good pratices to follow for reproducible experiments.

1. Use a long-term, publicly available, properly organized, version control repository
2. Use reusable open source software and proven simulator
3. Provide environment
4. Provide experiment design and workflow
5. Provide inputs and results

We will focus here on the second and third practices.

An invariable experimental environment is crucial to ensure reproducibility. Slight changes in the environment variables could lead to noticible differences in the results [20]. This is why several tools have been developed to help computer scientists experiement on the exact same setup.

Nix

Courtes and Wurmus[11] underline that functional package managers, like *Guix* [12] or *Nix* [13], are good candidates to share complex environements among users. We will focus on *Nix* in the following, but both *Nix* and *Guix* have similar approaches.

Nix is a purely functional package manager introduced by Dolstra et al. [13]. In *Nix*, each package is defined as a deterministic function with no side effect, and wittren in

the *Nix* Expression Domain Specific Language (DSL). Using such functions as package definition answers the issue of Dependency hell. Dependency hell can be defined as the frustration of having to install application packages which have dependencies on specific versions of other packages which might end up by having multiple different versions of the same library/software on a single system possibly leading to some issues [1]. Indeed, a *Nix* derivations requires the user to specifiy the versions of each dependency of the package. *Nix* will then build the specified versions of the dependencies and linked them to the installed software.

When building a package, *Nix* will produce files called "derivations", and will generate a hash of all the inputs to the build (version of dependencies, source code, etc). *Nix* then stores the packages in the *Nix* store, where each package has its own readonly subdirectories, making it impossible to alter a package. Thus, if a user wants to build the same package with a difference version of a compiler for example, *Nix* will create a new derivation with a different hash i.e. a new package.

In a *Nix* package, all the dependencies must be defined as a *Nix* package. This has the advantage of having a clear definition of all the dependencies of a software. The drawback is that a user wanting to build a package on her own using a dependency without a *Nix* expression will have to write it herself.

Here is an example *Nix* expression building the GNU `hello` package[4].

```
{ stdenv, fetchurl, perl }:

stdenv.mkDerivation {
  name = "hello-2.1.1";
  builder = ./builder.sh;
  src = fetchurl {
   url = ftp://ftp.nluug.nl/pub/gnu/hello/hello-2.1.1.tar.gz;
sha256 = "1md7jsfd8pa45z73bz1kszpp01yw6x5ljkjk2hx7wl800any6465";
  };
  inherit perl;
}
```

The main steps of the building of this package are the following:

1. *Nix* will download the source code using the given url
2. it will compare the `sha256` hash of the download source code with the hash in the expression and fail the build if they differ
3. `Nix` then call the builder file to build the package

The first line of the expression states that this is a function expecting three parameters:

- `stdenv` which provide the standard environment tools to build a package (e.g. C/C++ compiler)
- `fetchurl` to download the source code of the package

[4] `https://www.gnu.org/software/hello/`

- the `perl` interpreter to execute the code of the package

NixOS[5] is a Linux distribution built on top of the *Nix* package manager. It aims at improving the state-of-the-art in system configuration and management. In *NixOS*, the entire OS (i.e. kernel, applications, system packages, configuration files ...) is built by the *Nix* package manager from a description in the *Nix* language. New configurations cannot overwrite previous configurations, making it simple to roll back to an older configuration of the system.

Kameleon

Variations of the Operating System (OS) can introduce unreproducible experiments. The Kameleon tool [21] generates reproducible customized OS. It provides a full software stack from the bootloader to the application layer. The description of the OS image is written in the YAML[6] description language and can thus be easily modified and more importantly, shared.

2.3.3 Reproducibility in Distributed Systems

When dealing with distributed systems, the complexity to keep the experiments reproducible increases as the number of machines increases.

To evaluate the performance of a distributed application, we could run it on a supercomputer. However, this approach makes it difficult to reproduce, as not everyone has access to a supercomputer.

Grid5000 and Kameleon

Deploying a distributed system in a reproducible fashion can be done using tools like Kameleon (see Section 2.3.2). However, the main drawback of this approach is that a user needs several machines in order to reproduce the experiment.

Arion

Arion[7] is a tool for building and running applications that consists of multiple docker containers using *NixOS* modules. *NixOS* modules are *Nix* expressions defining system services with their configuration and their dependencies. It has special support for Docker images that are built with *Nix*, for a smooth development experience and improved performance.

Users can define a *NixOS* module for each of the different machines that they would like to deploy. *Arion* will then create the `Dockerfile` associated to the modules, and then coordinate them using `docker-compose`[8].

By doing so, we are able to experiment on a simulated distributed systems using a single machine.

[5] `https://nixos.org/`
[6] `https://yaml.org/`
[7] `https://docs.hercules-ci.com/arion/`
[8] `https://docs.docker.com/compose/`

This approach also has the benefit of greatly reducing the development cycle time. Indeed, approaches like Kameleon on Grid5000, presented in Section 2.3.3, do take time to develop with. The generation of the OS image, as well the time to deploy the image to the nodes can take dozens of minutes. The *Arion* approach, as it is using containers, takes around one minute to boot up the entire system.

NixOps

The *NixOps* project[9] proposes to deploy *NixOS* machines over a network or the cloud. *NixOS* machines can be described using the *Nix* DSL which ensures the tracability of the build.

2.4 The CiGri Middleware

2.4.1 Presentation

CiGri[10] (CIMENT Grid) is a lightweight grid middleware application designed to exploit unused resources of a cluster [15]. It runs on top of a set of *OAR*[8] clusters in order to manage large sets of multi-parametric tasks, also called bag-of-tasks. Figure 2.4 depicts the global system.

Bag-of-tasks is defined as a collection of independent parametric tasks which can be executed in any order. One example could be Monte-Carlo experiments where the user needs to run a large set of experiments.

CiGri interfaces the *OAR* server. *CiGri* jobs are viewed by *OAR* as *best-effort* jobs.

Best-effort Jobs *Best-effort* jobs are jobs with a lower priority than a regular cluster user's job. If a regular user of the cluster needs the resource where a *best-effort* job is running, then the later will be stopped and the cluster user will get the resource. *Best-effort* jobs will only get scheduled on idle cluster nodes, as opposed to higher priority jobs that can interrupt lower priority jobs in order to run on a specific resource. Moreover, these jobs are killed when a regular job recently submitted needs the nodes used by a *best-effort* job. By default, there is no checkpointing or automatic restart of *best-effort* jobs when killed.

2.4.2 Bag-of-tasks Execution and Limitations

Algorithm

The current job submission algorithm implemented in *CiGri* is based on a tap mechanism.

CiGri will submit a number of jobs to the scheduler (open the tap) and will wait until all these jobs have finished (close the tap) to submit a new subset of jobs (open the tap again).

[9] `https://github.com/NixOS/nixops`

[10] `http://ciment.univ-grenoble-alpes.fr/cigri/dokuwiki/doku.php`

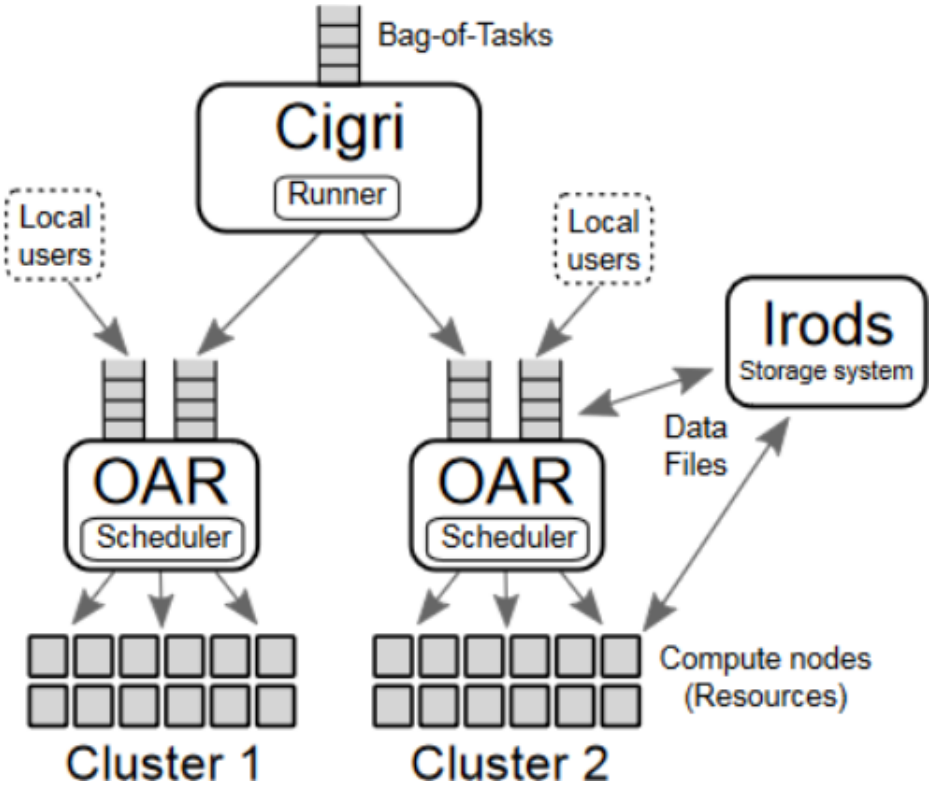

Figure 2.4: Graphical Representation of the System

The algorithm implemented is *CiGri* is presented below:

Algorithm 1: CiGri Current Job Submission

```
rate = 3;
increase_factor = 1.5;
while jobs left in bag-of-tasks do
    if no running jobs then
        launch rate jobs;
        rate = min(rate × increase_factor, 100);
    end
    while jobs running > 0 do
        sleep until timeout;
    end
end
```

Resource Utilization

The current solution implemented in *CiGri* has the drawback that once it has submitted a set of jobs, it will wait for the completion of all those jobs before submitting another set of jobs. We can think of a situation where this solution can lead to an under-utilization of the resources.

Let us have a cluster of 100 resources, and for the sake of the example, an infinite bag-of-tasks. Let us imagine that there are no users using the cluster. The *CiGri* algorithm

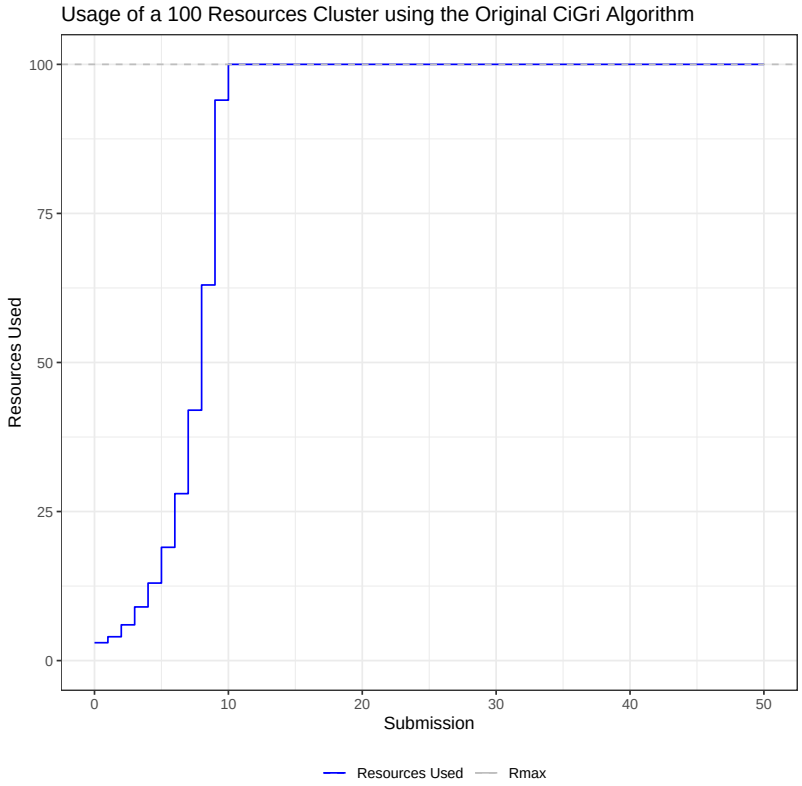

Figure 2.5: Example of situation where the original CiGri algorithm under-performs

will start by submiting 3 jobs, then once those jobs are terminated, will submit 4 new jobs ($3 \times 1.5 = 4.5$ and $\lfloor 4.5 \rfloor = 4$). If we note T the execution time of one job, and if we consider that the execution times are simular across the bag-of-tasks, then it will take $10T$ to reach full utilization of the cluster.

Figure 2.5 shows the utilization of the cluster in this scenario.

During the rising phase of the algorithm (i.e. before reaching the maximum number of resources), there have been 660 resources left idle and 340 used resources, which represents a usage of 38%.

One can argue that for a large number of jobs in the bag-of-tasks, the impact of the unused resources is diminished.

Table 2.1 presents the number of jobs required for the total utilization of the cluster to be a certain percentage. As we can see, in order to reduce the impact of the under-utilization of the rising phase, we would need to have a large number of tasks in the bag-of-tasks.

One can also argue that a solution would be to change the intial values of the $rate$ and $increase\_factor$ in order to reduce the under-utilization and thus have a more aggresive submission mechanism. The issue with this strategy would be that *CiGri* could overload the scheduler with a large submission if the cluster is being highly requested by regular users increasing the response time of the *OAR* scheduler [8].

Another example of under-utilization from *CiGri* would be in the situation where there are only a few jobs left from the previous submission to be executed before the new submission. If we imagine a situation where the usage of the cluster is high and a

Total Utilization	Number of jobs required	Exec. Time	Optimal Exec. Time
50%	724	$14T$	$7T$
75%	2124	$28T$	$21T$
90%	6124	$68T$	$61.2T$
95%	12824	$135T$	$128.3T$
99%	66524	$672T$	$665.3T$

Table 2.1: Utilization of a 100 resources cluster with the original CiGri submission algorithm

few moment later decreases drastically (e.g. lunch break), then the current submission algorithm of *CiGri* would wait for the tasks from its previous submission to be executed to submit again, even if the number of jobs left is inferior to the number of idle resources.

2.4.3 Previous Works

The proof of concept of the control-based approach for *CiGri* was done by Stahl et al. in [23]. It was using a simple Proportional controller (see Section A.1.3) to improve the utilization of the number of resources of the cluster.

In later work, the writing phase of the fileserver was studied by Yabo et al. in [25]. A model predictive controller (see Section A.1.3) was developped to improve the utilization with the dual objective of maximizing the utilization of the cluster while regulating the load of the fileserver.

The latest work [5] focused on the scalability of the previous approaches when increasing the number of nodes in the system.

2.4.4 Reproducibility of Experiments

This research project is done in collaboration with people from Gipsa Lab[11], whose core expertise is not computer science. However, running an experiment requires some computer science knowledge and more importantly, troubleshooting skills in a complex environement.

It would be foolish to ask control theorists to understand and master all the tools used to run an experiment.

Section B.1 of the Appendix presents the steps to run an experiment. The experimental workflow can be very discouraging and intimidating for non-computer scientists. Besides, the number of commands to write makes it very error prone, even for experienced users.

This is an area of the project where we could improve in both usability and reproducibility, in the sense of changing only the experimentator and carrying the same experiment as someone else.

[11] `http://www.gipsa-lab.grenoble-inp.fr/`

2.5 Conclusion

Given these observations, we strongly feel that the state of the art can be improve in two areas. First, we can build on the previous works on *CiGri* to propose a new controller regulating the number of jobs to send to the cluster based on the the load of the fileserver. Then, the reproducible trait of the *CiGri* experiments can be reinforced.

— 3 —
Development of a Load Driven Controller

This internship has been done in collaboration with people from Gipsa Lab[1], who are specialized in the field of control theory. During this work, I have been working with an intern from this laboratoty.

Thus the development of the controller used in this work was a back and forth discussion between the two parties.

In this Section, we start by modelling the system and then design a new controller with the load of the fileserver as unique objective.

3.1 Modeling

In this Section, we will present our model for the system.

Table 3.1 and Figure 3.1 summarise the different notations used. The notations are then explained in Section 3.1.1.

3.1.1 Notations

Maximum number of running jobs (r_{max})

The number of jobs running on the cluster is limited by several factors. First, the number of resources. The hardware of the machine can also be a reason for the scheduler to not

[1] `http://www.gipsa-lab.grenoble-inp.fr/index.php`

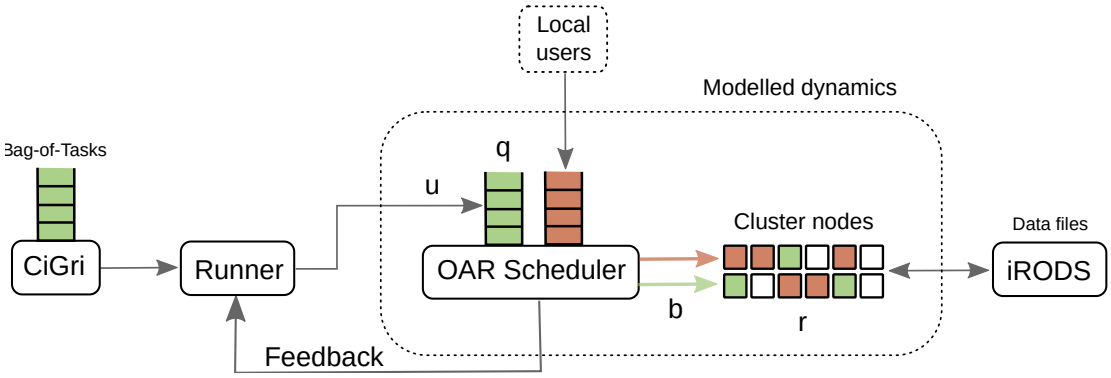

Figure 3.1: Graphical Representation of the System with the Feedback Loop

Notation	Description	Unit
u_t	Jobs sent by *CiGri* to *OAR* at time t	$jobs$
q_t	Jobs in the waiting queue at time t	$jobs$
b_t	Jobs allocated by *OAR* at time t	$jobs$
r_t	Jobs running in the cluster at time t	$jobs$
r_{max}	Maximum number of running jobs	$jobs$
p	Job processing rate	s^{-1}
Δt	Sampling time	s

Table 3.1: Notations

use all the resources of the cluster. Indeed, *CiGri* and *OAR* offer the opportunity to the users to specify the hardware they want their applications to run on. Thus, it can happen that r_{max} is inferior to the total number of idle resources.

Jobs sent by *CiGri* to *OAR* (u_t)

This quantity is the number of jobs that *CiGri* submits to *OAR* at time t. This is the only quantity that we can control.

Jobs in the waiting queue (q_t)

The quantity q_t represents the number of *CiGri* jobs being in the waiting queue of the *OAR* scheduler at time t

Jobs allocated by *OAR* (b_t)

Periodically, *OAR* takes jobs from its waiting queue and schedules them onto the resources. This mechanism is outside of the scope of this work. However, there are some constraints on the value b_t. Indeed, it is not possible to allocate at time t more jobs than there are in the waiting queue. We thus have: $0 \leq b_t \leq q_t$.

Jobs running in the cluster (r_t)

The number of running jobs at time t is constrainted by the maximum number of running jobs allowed (r_{max}). Thus, $0 \leq r_t \leq r_{max}$.

Job processing rate (p)

The value p represents the number of jobs that can be executed (and terminated) per second. This also includes the time for the scheduler to allocate (and de-allocate) the job. However, execution times in scientific workflows, and especially bag-of-tasks jobs, have minor variability among jobs. Thus, we will assume that the processing rate p remains constant during the execution of the campaign.

Sampling Time (Δt)

The sampling time is the time interval between two potential *CiGri* submission. Small values could lead to an overload of the cluster, while larger values can lead to an under-utilization of the resources due to a lack of reactivity. In this work, we took $\Delta t = 30s$ as it has been proven in previous work to be adequate [25].

3.1.2 Model

Yabo et al. [25] presented a queueing model for the number of jobs in each state (see Section A.2). However, their model did not correctly model the jobs terminating. Indeed, the proposed model made the assumption that a job that has been running for half of its total execution time was 50% terminated. The issue with this thinking is that if the model submits 100 jobs to the *OAR* scheduler, managing a cluster of 100 resources, and that the jobs take T seconds to complete, then at time $0.5T$, 50 jobs would have been completed and thus the model could submit 50 new jobs as it thinks there are 50 idle resources.

We thus propose a new model where the amount of jobs terminated at time t is exaclty the number of jobs sent to the cluster at time $t - T$, where T is the time to complete one job.

The equations describing the model are the following and are summarized in Figure 3.2:

$$\begin{cases} q_{t+\Delta t} = q_t + u_t - b_t \\ r_{t+\Delta t} = r_t + b_t - b_{t-d} \end{cases} \tag{3.1}$$

with $d = \frac{1}{p}$ and

$$b_t = \begin{cases} r_{max} - r_t & \text{if } q_t \geq r_{max} - r_t \\ q_t & \text{otherwise} \end{cases} \tag{3.2}$$

Concerning the number of jobs allocated by *OAR* at time t (b_t), we assume that the scheduler will allocate as many jobs as there are idle resources ($r_{max} - r_t$). If there are not enough jobs for all the resources, we suppose that it will allocate all the waiting jobs (q_t) onto the resources.

The value d is the delay between the submission of the jobs and the termination of the jobs. In this model, we assume that if a job takes $\frac{1}{p}$ seconds to complete one job on one resource, then all the jobs from the same submission (and from the same bag-of-tasks) will start at the same moment and will take the same time to complete.

Note that this value of d would be for a continuous time model, but we are working with discrete time in our case. We can adapt the value of d to discrete time: $d = \lceil \frac{1}{p\Delta t} \rceil \Delta t$

Another important remark is that the model depends on the value of p, the processing time of a job. However, the value can differ from job to job, with sometimes a standard deviation of several seconds. These variations make it difficult to have a perfect fit with the experimental results.

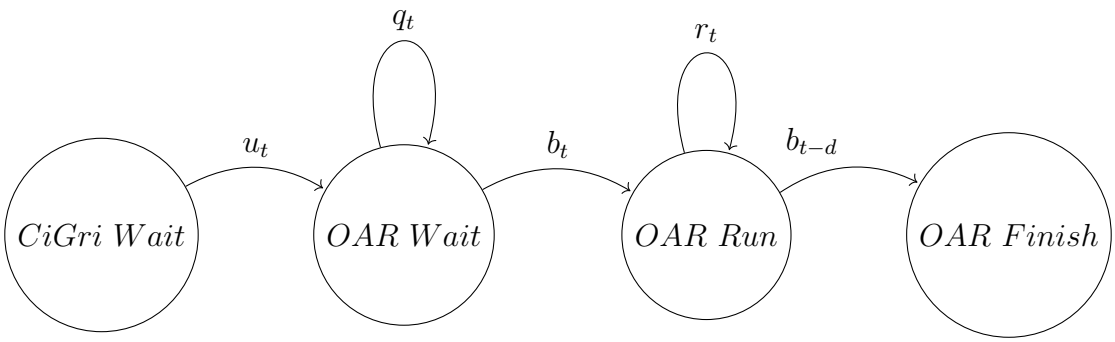

Figure 3.2: *CiGri-OAR* Workflow Model

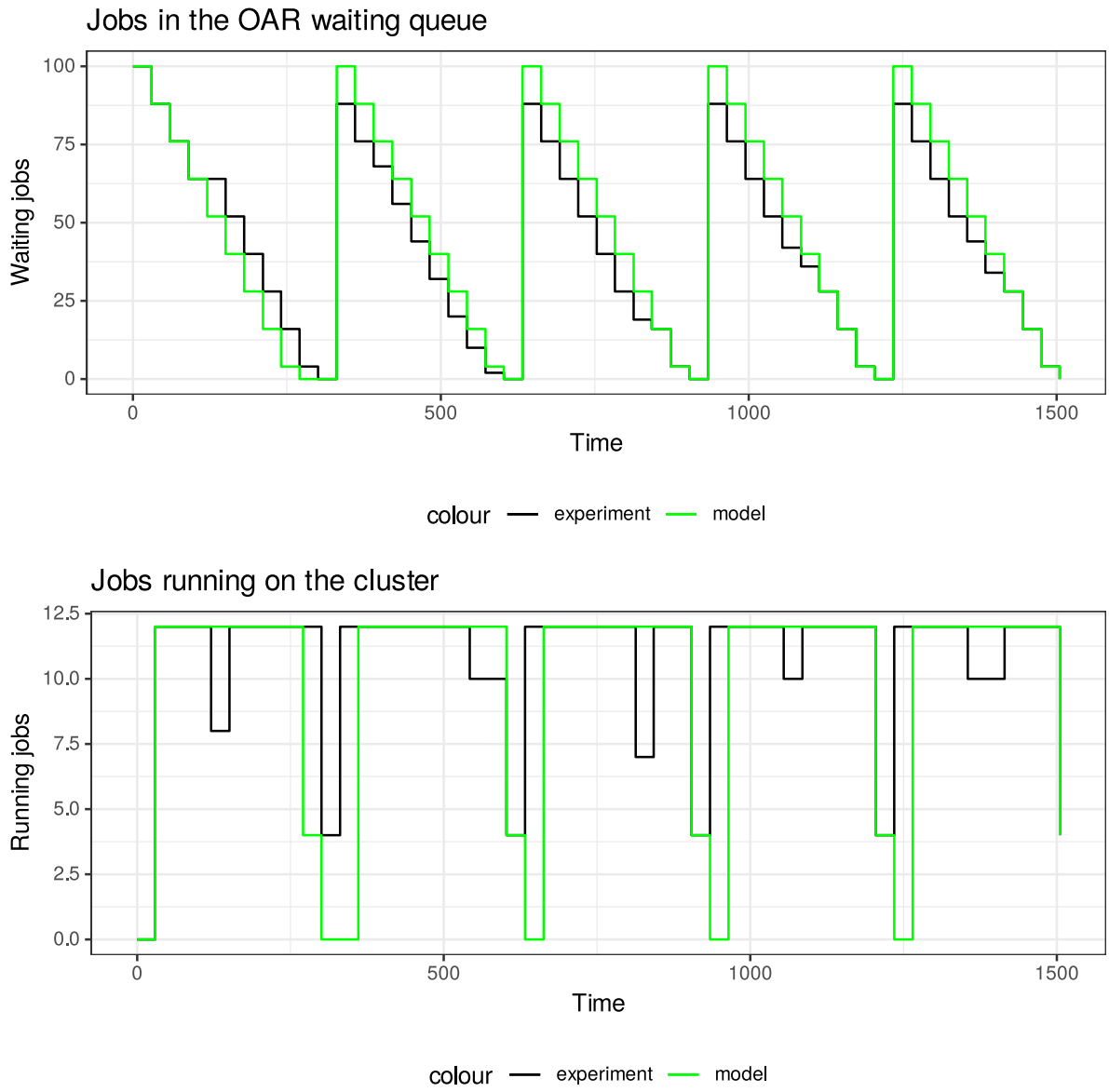

Figure 3.3: Model Validation: Comparison between Experiment and Model

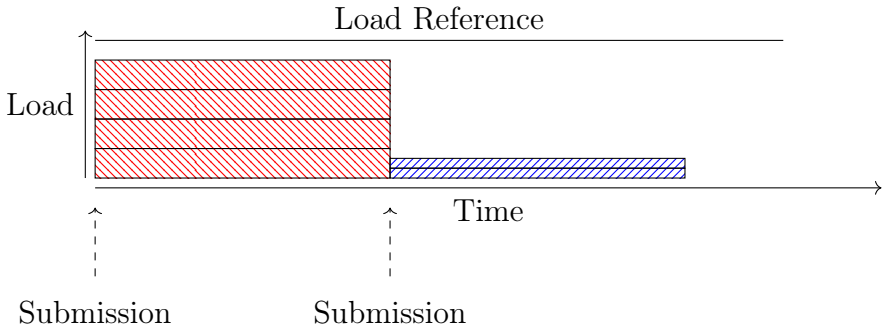

Figure 3.4: Submissions with jobs from single campaign

3.1.3 Validation

Figure 3.3 shows the behaviour of the system (in black) and the ouput of the model (in green).

There are several takeaways from Figure 3.3.

The first one is that the model does not fit perfectly the experiment for the waiting jobs (top graph). This could be due to several things. Maybe the jobs, for some reason, took longer to complete than their mean processing time. It could be due to the *OAR* scheduler taking longer to allocate and deallocate the jobs to and from the resources.

We can also see on the graph depicting running jobs that there are some "drops" in the experiments. This might be due to the timing of the measurement. Indeed, if we query the *OAR* scheduler for the number of running jobs when it is allocating (or deallocating) jobs, we will receive an slightly inaccurate value.

3.2 Presentation of the Problem

In this Section, we will look at a new controller whose main focus is to regulate the load of the cluster.

Originally, when *CiGri* sends jobs to the *OAR* scheduler, those jobs are part of the same campaign. This means that every of those jobs has a similar behaviour, and thus produces a similar impact on the load.

But some campaigns have jobs that have more IO operations than other campaigns, we define these campaigns as *IO heavy*.

Let us imagine that we have two campaigns submitted to *CiGri*, one with high IO load jobs and the other with light IO load jobs.

Figure 3.4 represents a situation using submissions composed of jobs from the same campaign. If we suppose that we are able to regulate perfectly the number of jobs submitted for the red campaign to keep the load of the cluster under the reference load, there will be a "gap" between the actual load of the cluster and the reference load. We could exploit this "gap" by submitting jobs that have a smaller impact on the load (blue jobs). Of course, we would first make sure that we have enough resources in the cluster.

Figure 3.5 shows a situation where we submit a set of jobs coming from different campaigns to improve the running time, the number of resources used while keeping the load under the reference value.

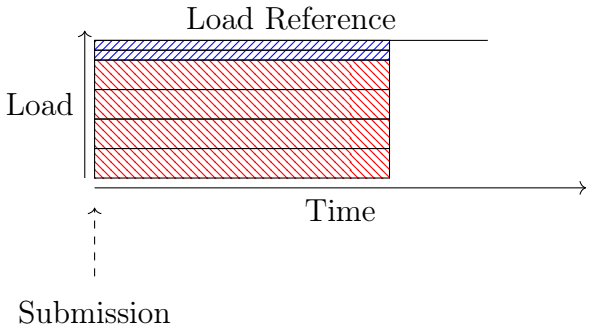

Figure 3.5: Submission with jobs from several campaigns

3.3 Proposed Solution

One solution to maximize the set of jobs to send while keeping the load of the cluster not exceeding the reference would be to measure the impact of each job on the load and deduce an optimal set of jobs to send. However, measuring the impact on the load of a single job is not an easy thing to do. Indeed, as the jobs from *CiGri* are running on a cluster of machines that are also being used by regular users, a measurement would be erroneous, thus leading to a unefficient optimization.

Measuring the impact on the load of a job is thus impossible. The solution that we came up with implies the user to specify in which category the jobs from her campaign are. In the campaign file, represented as a JSON[2] object, the user would define the IO heaviness of the campaign. Here follows a reduced example of a campaign file for a campaign with jobs carrying a high IO load.

```
{
  "name": "example heavy campaign",
  "resources": "resource_id=1",
  "exec_file": "sh io_heavy_job.sh",

  "heaviness": true,

  "clusters": {
    "cluster_0": {
      "type": "best-effort",
      "walltime": "300"
    }
  },
  "params": [
    ...
  ]
}
```

[2] https://www.json.org/

The campaign is considered IO heavy if it has jobs performing IO operations on the fileserver. And IO light campaign has jobs which perform no IO operations on the fileserver.

We now suppose that in each set of jobs submitted we have jobs from two distinct campaigns: a IO heavy campaign and a IO light campaign.

The parameters that we can alter are:

- The total number of jobs submitted (i.e. number of jobs from IO heavy campaign + number of jobs from IO light campaign)
- The proportion of jobs from each campaign in the set of jobs to submit

3.4 Determining the impact of each strategy

Let f_h be the load of one IO heavy job and let f_l be the load of one IO light job. Please note that measuring those values in pratice is delicate. Indeed, as there might be some perturbations on the cluster (e.g. other users), we cannot simply measure the load and deduce the values of f_h and f_l. However, in practice, we do not really need to know the exact value of f_h and f_l. What we need to know is which of the two campaigns has heavier jobs. Thus, instead, as a first solution, we decided to ask the user the IO weight of the job in her campaign: either heavy or light. We realize that such a classification is tricky and subjective to the user. We hope to find a accurate way to measure the heaviness of the jobs in future works.

Let b be the number of jobs sent from *CiGri* to *OAR*. We will suppose here that all these b jobs are sent together from *OAR* to the resources together.

Let $p \in [0, 1]$ be the percentage of IO heavy jobs in b.

We can model the load of the submitted set of jobs as:

$$f(p, b) = (p \times f_h + (1 - p) \times f_l) \times b \tag{3.3}$$

Let us suppose that there are n_h heavy jobs in the buffer ($n_h = p \times b$). We want to increase n_h by δn. There are two ways to achieve this: either by increasing the number of jobs in the buffer (b), or by increasing the percentage of IO heavy jobs (p).

3.4.1 Increasing the size of the buffer

Let us fix the value of p.

So to go from $n_h = pb$ to $n_h + \delta n$, we have to increase the b to b', with b' as follow:

$$b' = \frac{\delta n}{p} + b \tag{3.4}$$

Let us apply f on (p, b'):

$$f(p, b') = (p \times f_h + (1 - p) \times f_l) \times \left(b + \frac{\delta n}{p}\right) \tag{3.5}$$

And we get:

$$f(p, b') = f(p, b) + \delta n f_h + \frac{1 - p}{p} \delta n f_l \tag{3.6}$$

So, in order to increase the number of IO heavy jobs by δn by only changing the buffer size (b), we increase the load by $\delta n f_h + \frac{1-p}{p}\delta n f_l$.

As the proportion is kept constant, increasing the number of IO heavy jobs leads also to an increase in the number in the number of IO light jobs.

3.4.2 Increasing the percentage of IO heavy jobs

Let us fix the value of b.

So to go from $n_h = pb$ to $n_h + \delta n$, we have to increase the p to p', with p' as follow:

$$p' = \frac{\delta n}{b} + p \tag{3.7}$$

Let us apply f on (p', b):

$$f(p', b) = \left(\left(\frac{\delta n}{b} + p\right) \times f_h + \left(1 - \left(\frac{\delta n}{b} + p\right)\right) \times f_l\right) \times b \tag{3.8}$$

And we get:

$$f(p', b) = f(p, b) + \delta n \left(f_h - f_l\right) \tag{3.9}$$

So, in order to increase the number of IO heavy jobs by δn by only changing the percentage (p), we increase the load by $\delta n \left(f_h - f_l\right)$.

As the number of jobs is kept constant, an increase in the number of IO heavy jobs leads to a decrease in the number of IO light jobs.

3.4.3 Comparing strategies

Let us make the difference between both strategies.

$$\Delta f = f(p, b') - f(p', b) = \frac{\delta n}{p} f_l \tag{3.10}$$

This means that changing the buffer size has a bigger impact on load.

So, if we want to regulate the load of the system precisely, we should do these actions in the following order:

1. Change the buffer size
2. Change the percentage of IO heavy jobs

This result was expected, as increasing the number of total jobs submitted also increases the number of IO light jobs which, even if smaller than the IO heavy jobs, have an impact on the load.

3.5 Proposed Controller

As we saw in the previous section, changing the number of jobs sent to *OAR*, while keeping the percentage fixed, has a greater impact on the load than changing the percentage and keeping the number of jobs fixed. We will thus design a controller using this observation.

The contoller will have two modes. Each mode will run individually. We will decide which mode to run depending on the error represented as the absolute value of the difference between the reference load value and the current load of the cluster.

The modes of the controller depend on the error:

1. When the error is high, we need to change the number of jobs sent to *OAR* as it has the most impact.

2. When the error is low, we need to change the percentage of IO heavy jobs sent to *OAR*.

The idea behind this controller could be summed up as "big step, small step". We will first control the number of jobs sent to *OAR* (big step). Then, when we are "close" to the reference value, we change the proportion of IO heavy jobs among the jobs sent to *OAR* (small step).

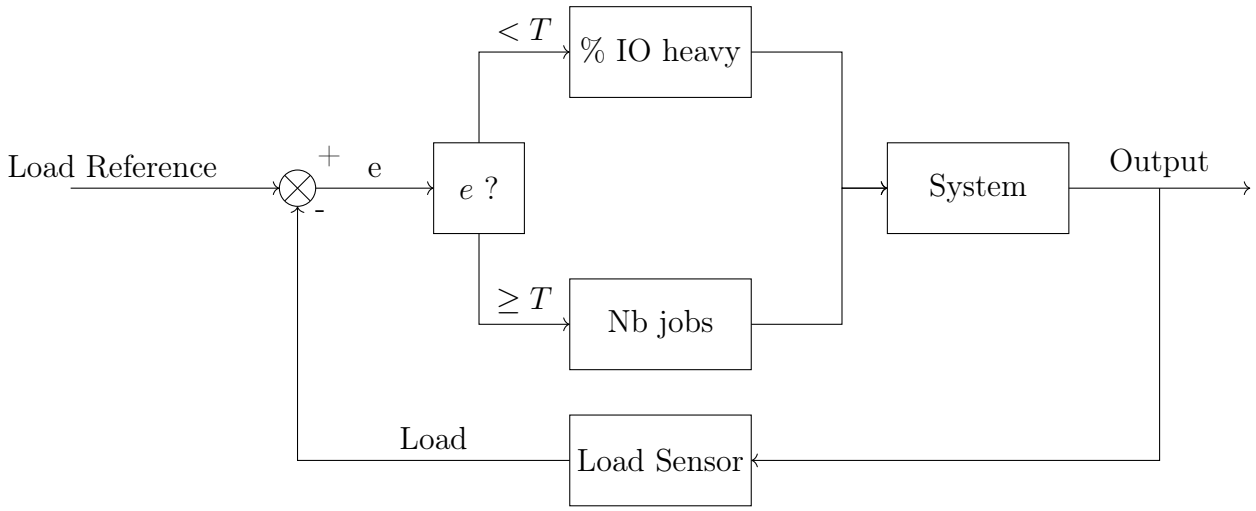

Figure 3.6: Feedback loop

Figure 3.6 depicts the overall idea of the controller. Note that the objective of this approach is to get the load of the fileserver within a given interval and to keep it inside.

As a first approach, the two modes of the controller are regulated using a Proportional controller (see Section A.1.3). A Proportional controller can be expressed by the following equation:

$$u(t) = k_p e(t) \tag{3.11}$$

where k_p is called the gain, $e(t)$ is the error at time t and $u(t)$ is the output of the controller at time t.

In our case, the error is the difference between the reference for the load and the value return by `loadavg`.

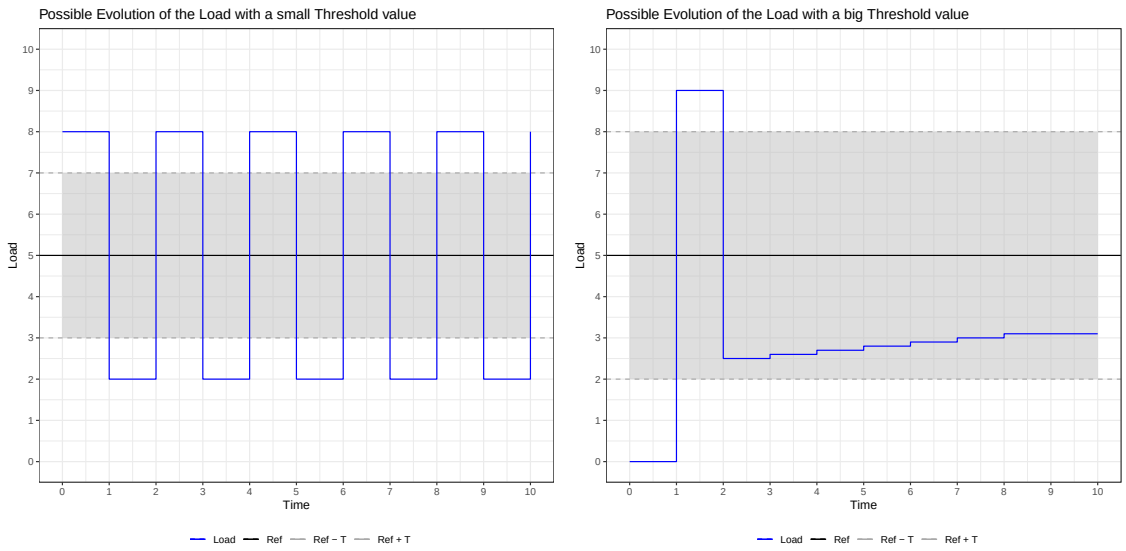

Figure 3.7: Graphical Representation of the potential Behaviour of the Load if the Threshold is too small (left) or too big (right)

Mode 1: Number of jobs sent For the first mode of the controller, the output should be the number of jobs to send. We will note k_1 the proportional gain for this mode.

Mode 2: Percentage of IO heavy jobs For the second mode of the controller, the output should be the percentage of IO heavy jobs in the submission. We will note k_2 the proportional gain for this mode.

3.6 Choice of the Parameters

3.6.1 Threshold

In Figure 3.6, we make the choice of the mode to use depending on the error. We compare the absolute value of the error to a threshold T. We remind that the error is defined as the distance between the reference value (R) and the actual value.

Small Threshold Value

If T is too small, then it will be more difficult to get into the mode regulating the percentage of IO heavy jobs. This is due to the fact that we will only regulate using the number of jobs submitted. The left plot of Figure 3.7 presents a simplified scenario where the load oscillates with values outside of the target interval.

Big Threshold Value

If T is too large, then it might get "stuck" in the mode regulating the percentage of IO heavy jobs. This could lead to reach a non optimal stationary state. The right plot of Figure 3.7 presents a simplified example where the controller manages to keep the load

inside the target interval, but the percentage of IO heavy jobs can already be at 100% and thus making it impossible for the controller to get closer to the reference.

Choice of the Threshold value

For the remaining of this study we choose a threshold value of 1 ($T = 1$) as experiments showed not to be too small nor to big.

3.6.2 Managing the load

Let us suppose that for (p, b) the load is $f(p, b) = f$. We want to change the load by δf. There are two ways to achieve this: either by changing the number of jobs in the buffer (b) or by changing the percentage of IO heavy jobs (p).

Changing the size of the buffer

We want to find b' such that $f(p, b') = f(p, b) + \delta f$.

We get:

$$b' = \frac{\delta f}{p f_h + (1-p) f_l} + b = \left(1 + \frac{\delta f}{f}\right) b \tag{3.12}$$

Changing the percentage of IO heavy jobs

We want to find p' such that $f(p', b) = f(p, b) + \delta f$.

We get:

$$p' = \frac{\delta f}{b(f_h - f_l)} + p \tag{3.13}$$

3.6.3 Upper bound on the parameters

From the results of the Section 3.6.2, we can extract upper bound values for the parameters of the controllers.

Controller on the Number of Jobs

We are in the situation where the error $|e| \geq T$. We would like the change in the number of jobs to result in an increase, or decrease, of at least one job.

With this information, we can write:

$$\begin{cases} k_1 \geq \frac{1}{T} & \text{if } \ 1 \leq k_1 T \leq |k_1 e| \\ k_1 \leq \frac{1}{T} & \text{if } \ k_1 T \leq 1 \leq |k_1 e| \end{cases} \tag{3.14}$$

We could then choose $k_1 = \frac{1}{T}$.

To be a little more conservative, we took $k_1 = 0.5$.

Controller on the Percentage of IO heavy jobs

As we are increasing or decreasing the percentage of IO heavy jobs, we want the proportional coefficient of the controller to be such that a change in the percentage (δp) changes the number of jobs (δb) sent from the heavy IO campaign.

$$(p + \delta p)b = pb + \delta b \implies \delta p = \frac{\delta b}{b} \geq \frac{1}{b} \tag{3.15}$$

However, we are in the situation where the error e is smaller than T in absolute values. Thus,

$$\frac{1}{b} \leq |k_2 e| \leq k_2 T \implies k_2 \geq \frac{1}{bT} \tag{3.16}$$

For the remaining of this work we took $k_2 = 0.1$ as we consider that the number of jobs sent (b) will be greater than 10 when regulating in the second mode.

3.7 Evaluation

3.7.1 Experimental Setup

To test the controller described in Section 3.5, we used the following setup:

- One *CiGri* Server
- One *OAR* Server
- One Fileserver
- One Cluster of 100 resources

The experiment was done on the Grisou Cluster from Grid5000. Each node of this cluster has two Intel Xeon E5-2630 v3 CPU with eight cores per CPU and 128 GB of memory.

The experiment consisted in submitting simultaneously two different campaigns to *CiGri* containing 1000 jobs each. The first campaign contains IO heavy jobs. These jobs sleep for 30 seconds then write to the fileserver a file of 10Mb. The jobs from the other campaign, the light IO campaign, only sleep for 30 seconds. We regulated the load of the fileserver around the value 3, with a threshold of 1.

3.7.2 Results

Figure 3.8 presents the results of the experiment using this new controller.

We can see a first phase where the number of total jobs is rising, with the proportion of IO heavy jobs remains constant at 50%. Around 700 seconds, the load enters the interval where the control switches mode into the modification of the percentage of IO heavy jobs sent. The percentage decreases until reaching 0% around 1300 seconds. At this point, the load is in the lower half of the interval. Around 1500 seconds, the load gets lower than the lower bound of the control interval, the controller thus increases the number of jobs sent. This lead to an increase of the load, that will be met with a decrease

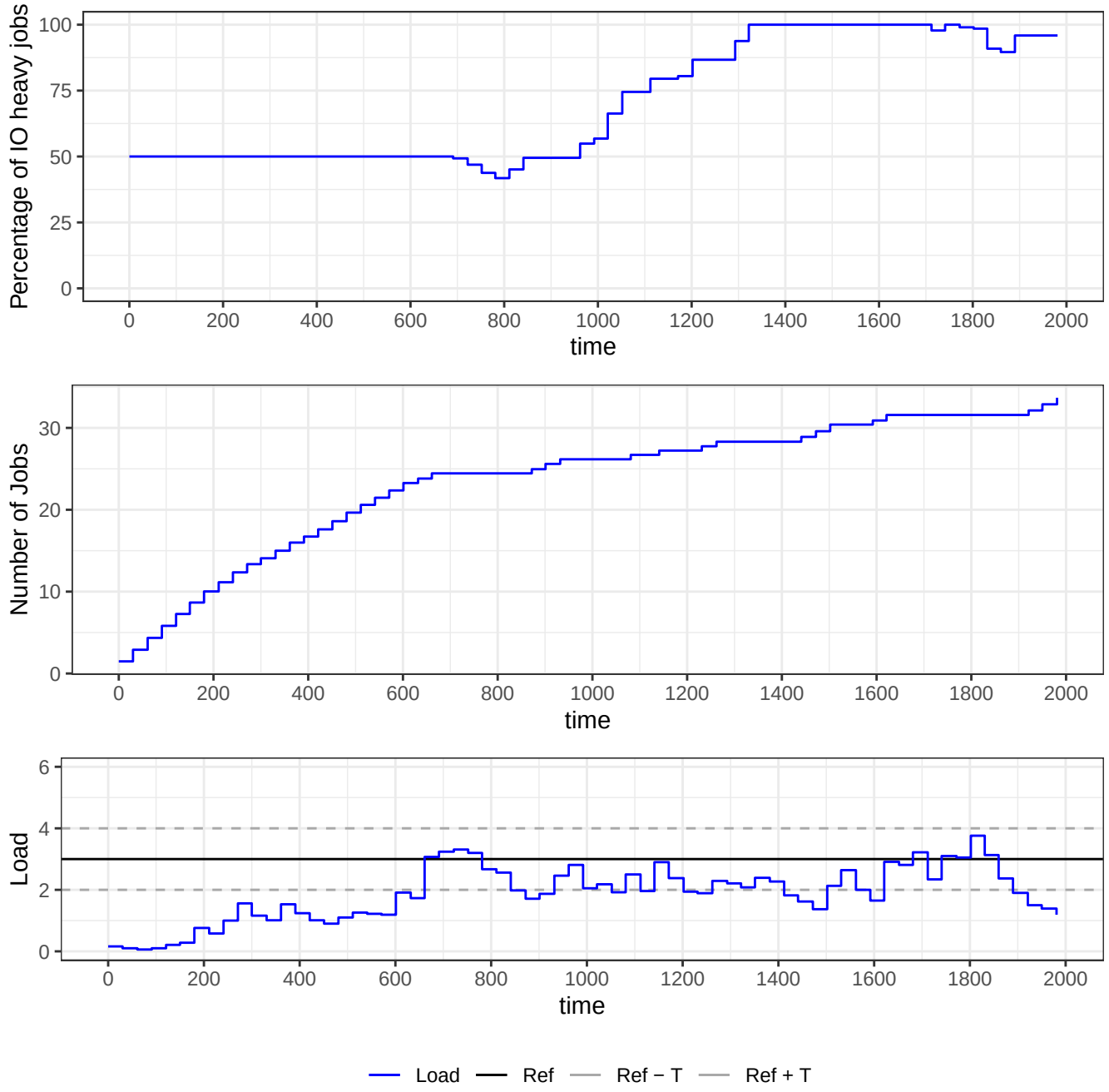

Figure 3.8: Experimental Result of the Controller

of the percentage of IO heavy jobs around 1700 seconds. Around 1800 seconds the IO heavy campaign is terminated (no jobs left), and we can see the load decreasing. Note that Figure 3.8 only shows when both the IO heavy and IO light campaigns are running.

Previous work [25] defined a Model Predictive Contoller (MPC, see Section A.1.3) regulating the number of jobs sent by *CiGri* with respect to the load of the fileserver. When comparing the controller proposed in this Chapter to the MPC controller, we find that the new controller performed better. Indeed, the presented controller completed the IO heavy campaign and the IO light campaign in 39 minutes, for a cluster utilisation 25%. Whereas, the MPC controller failed to complete any of the campaigns under 3 hours. It was averaging 6 jobs per submission, leading to an estimated cluster utilisation of 6%.

We define the cluster utilisation, with the notations seen in Section 3.1.1, as follows:

$$U = \frac{1}{t_f - t_0} \sum_{t=t_0}^{t_f} \frac{r_t \Delta t}{r_{max}} \tag{3.17}$$

Note that the constraint on the load (in our case keeping the load of the fileserver around a value of 3) makes it impossible to reach 100% cluster utilisation. Different reference values for the load would of course lead to different values of the cluster utilisation.

3.8 Conclusion

In this Section we presented a proof of concept for a controller focusing on regulating the load of the fileserver. We introduced a mechanism to improve the granularity of the control by submitting jobs from several campaigns with different IO loads.

This work can be improved by using more complex controllers like a PID controller (see Section A.1.3) or a MPC controller (see Section A.1.3). We could also think of others ways to alternate between the two modes of the controller. We could, for example, change the mode of control (number of jobs sent and percentage of IO heavy jobs) at different frequencies: every 2 minutes we control on the number of jobs sent and in between we control the percentage of IO heavy jobs.

— 4 —

Towards Reproducibility

4.1 Motivation

Experimenting with our system requires several machines to represent the studied architecture (see Figure 4.1). We are fortunate to have access to Grid5000, but this might not be the case for everyone. We are thus currently transitioning the deployed system to a container approach using *Arion* and Docker that can be executed on a single machine (see Figure 4.2). However, this transition brings some incertitudes concerning the work previously done. For instance, we can wonder if the models developed for the deployed system still stand in a containerized architecture.

In this section we investigate the impact of the container approach on the project. We will also present solutions to help non computer scientists convey experiments with reproducible workflows.

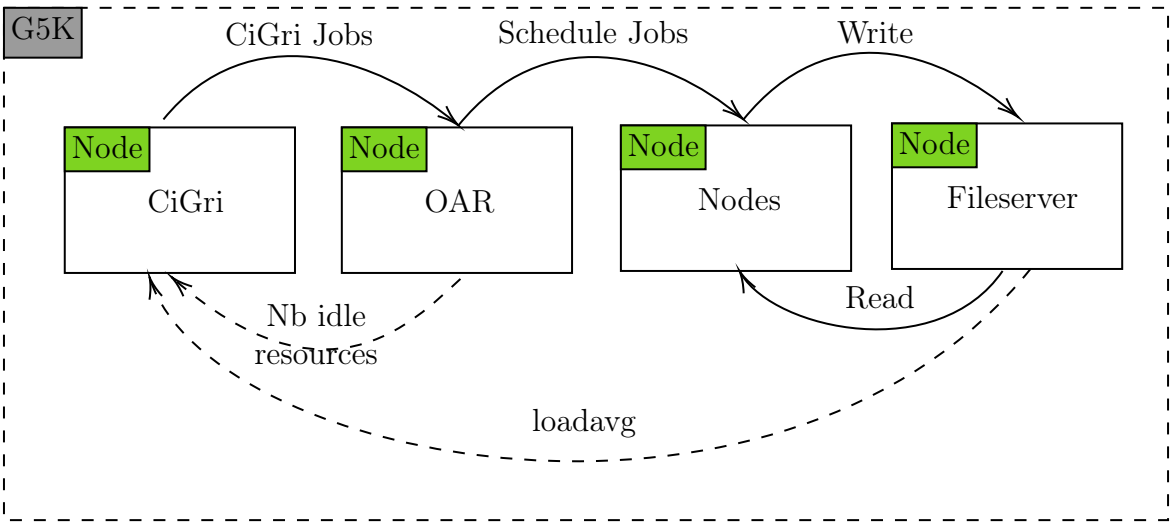

Figure 4.1: Simplified Representation of the System in a deployed environment for a single Cluster

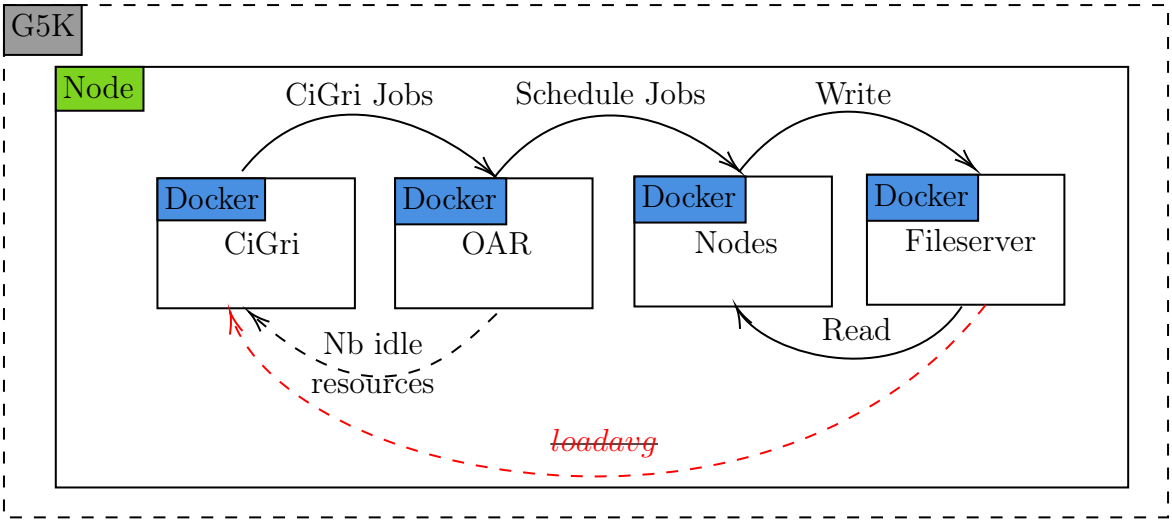

Figure 4.2: Simplified Representation of the System with a container approach for a single Cluster

4.2 *Arion* Configuration

We present in this section a portion of the *Nix* expression for *Arion* to generate the `docker-compose` file.

The complete listing can be found in Section B.2.

4.2.1 Definition of the global *NixOS* Configuration

We first define the common configuration for all the containers. We define a volumes (as defined by Docker) and some environment packages (e.g. ruby, openssh).

```
service.volumes = [ "${builtins.getEnv "PWD"}/..:/srv" ];
service.capabilities = { SYS_ADMIN = true; }; # for nfs
service.useHostStore = true;

nixos.useSystemd = true;
nixos.configuration = {
    networking.firewall.enable = false;

    users.users.user1 = {isNormalUser = true;};
    users.users.user2 = {isNormalUser = true;};

    environment.systemPackages = with pkgs; [ nfs-utils socat wget ruby openssh ];
    imports = lib.attrValues pkgs.nur.repos.kapack.modules;
};
```

It will be the building block for all the nodes.

4.2.2 Definition of the Nodes

We can now define the different nodes on top of the common configuration. We give in this Section two examples: the *Nix* expressions for *CiGri* and for the fileserver.

CiGri Node

The *CiGri* node needs to define a database, as well as a server. We can also define services ourselves with a custom script. Note that the packages required for the execution of our service are specified.

```
services.cigri = addCommon {
  service.hostname="cigri";
  nixos.configuration = {
    services.cigri = {
      dbserver.enable = true;
      client.enable = true;
      database = {
        host = "cigri";
        passwordFile = "/srv/common/cigri-dbpassword";
      };
      server = {
        enable = true;
        web.enable = true;
        host = "cigri";
        logfile = "/tmp/cigri.log";
        };
      };
    services.my-startup = {
      enable = true;
      path = with pkgs; [ nur.repos.kapack.cigri sudo postgresql ];
      script = ''...'';
    };
  };
};
```

Fileserver

Another example of *Nix* expression defining a node. In this case, we define the fileserver node by simply adding a NFS server on top of the common configuration.

```
services.fileserver = addCommon {
  service.hostname="fileserver";
  nixos.configuration = {
    services.nfs.server.enable = true;
    services.nfs.server.exports = ''/srv/shared *(rw,sync,no_subtree_check,no_root_sq
  };
};
```

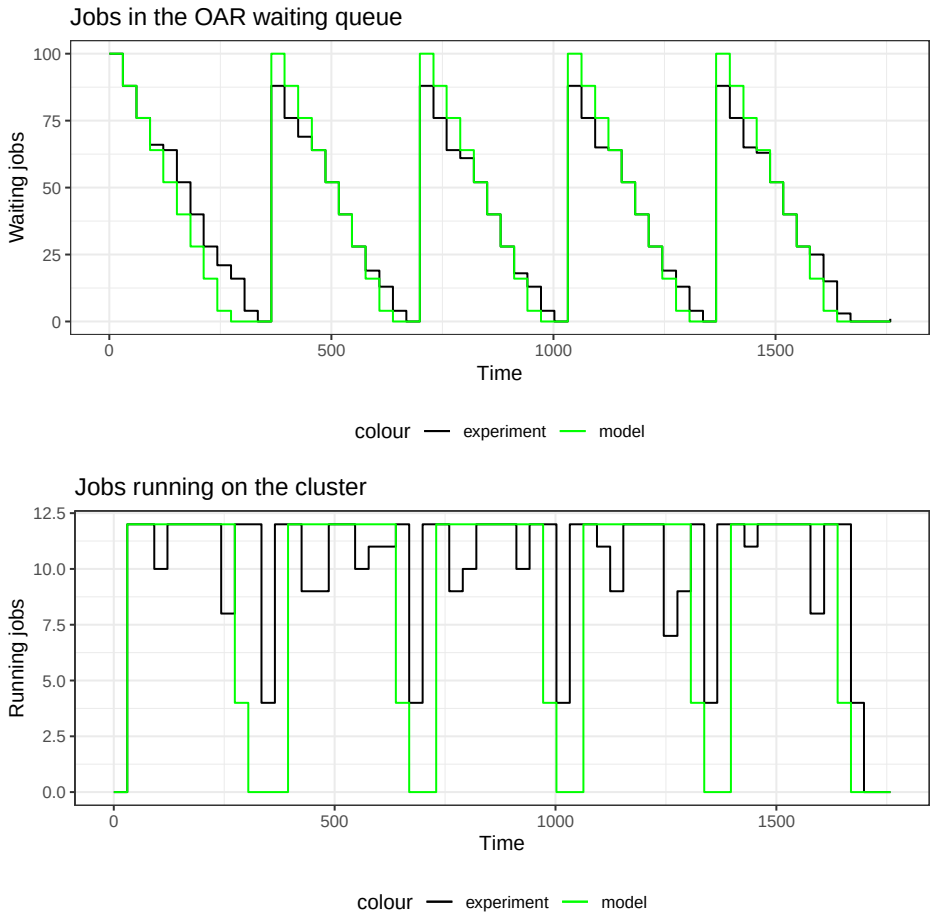

Figure 4.3: Validation of the job model for the container approach

4.3 Container Approach: Job Model

The first thing to control is that the model for the number of jobs presented in Section 3.1.2 still stands with the container approach.

Figure 4.3 shows a plot comparing the output of the model for the number of jobs with the experimental results.

We can see that the model does fit. Thus the container approach does not affect the model for the number of jobs.

However, if we compare Figure 4.3 with Figure 3.3, we can see that the execution time with the container approach is longer than with the deployed approach.

We suppose that this is due to the larger number of processes running on a single machine slowing the experiment down. Indeed, when using the container approach, all the processes from all the different nodes are being executed on a single machine.

4.4 Container Approach: `loadavg`

During this internship, we have been working on controllers using the output of `/proc/loadvg` as sensor. When we are experimenting on Grid5000 with multiple machines, the sensor

reads the intended value: the `loadavg` of the fileserver. However, when we are using *Arion* and containers, now the value returned by `/proc/loadavg` does not reflect the load of the fileserver, but of the host system. Thus, as we are running several containers on the same host machine, we cannot keep looking directly at `loadavg` as a sensor usable in both experimental setups (Grid5000 and *Arion*) (see Figure 4.2).

The objective of this section is to have a mean to compare the load in a deployed system and the load in containers.

4.4.1 Definition of the `loadavg`

The `loadavg` value represents the average number of jobs in the running queue or waiting for disk.

```
$ cat /proc/loadavg
0.20 0.29 0.70 1/1043 13302
```

We are only interested at the first number returned. It represents the load over the last minute.

The `loadavg` metric is computed as an exponential weighted sum on the number of jobs [14].

$$f_i = f_{i-1}\left(1 - e^T\right) + n_i e^{-T} \tag{4.1}$$

with, f_i the load at time i, n_i the number of jobs at time i and T the period of the average.

The advantage of this metric is that there is a sense of "overload". When the value returned by `loadavg` is greater than the number of cores on the machine, we can consider that the system is overloaded.

4.4.2 Comparison `loadavg` / Model

The objective of this Section is to see if it is possible to have an approximatation value of the load average by computing the metric "by hand".

In this experiment, we measured the `loadavg` during an experiment as well as the number of jobs in the CPU run queue and jobs waiting for CPU. We then compute the estimated `loadavg` value using only the number of jobs, and compare it to the actual value of the `loadavg`.

The results can be seen on Figure 4.4.

As we can see, the model returns a decent fit to the experimental values.

The reasons why we do not have a perfect match between the two curves could be:

- The measurements of the number of running and waiting processes might not be done at the same time as `loadavg`, thus resulting in different values.
- The previous history of `loadavg` is not taken into account. In Equation 4.1, we define f_0 as 0, which cannot be the case for the actual `loadavg`. As the load takes into account the previous value of the load, even with a light weight, this can introduces a slight deviation during the period of the experiment.

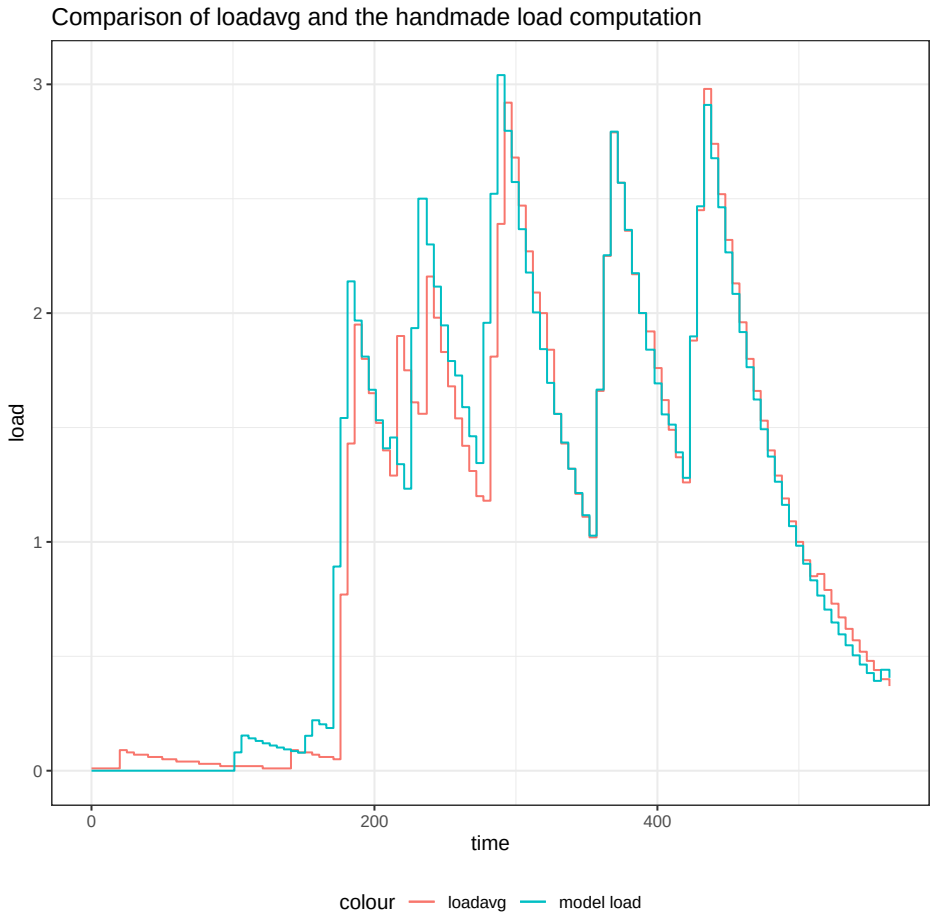

Figure 4.4: Comparison `loadavg` and model

4.4.3 Computing `loadavg` in a container

As explained previously, we cannot use the metric `loadavg` when using *Arion* as the value returned by `loadavg` will be the load of the host, not the container.

There have been some projects[1] trying to implement a container-based `loadavg`. Unfortunately, we did not manage to make it work in our case[2].

In order to compute the `loadavg` of the fileserver, we will focus only on the processes using the NFS server as there should be the only processes running on on the fileserver during the experiments. However, the current implementation of the NFS server for the container approach is not done in user-space. Thus the processes using the NFS server are not visible from inside the container, but they are visible from the host. We decided to only monitor the `nfsd` processes (processes from the NFS server) on the host to compute the load of the container.

Figure 4.5 shows the evolution of the load during the same campaign using a deployed system on Grid5000 (top) and using a single Grid5000 machine with *Arion* (bottom).

The first thing that we can see, is that there is the same number of peaks in the load.

[1] `https://github.com/lxc/lxcfs/pull/237`

[2] `/proc/loadavg` was returning an empty file

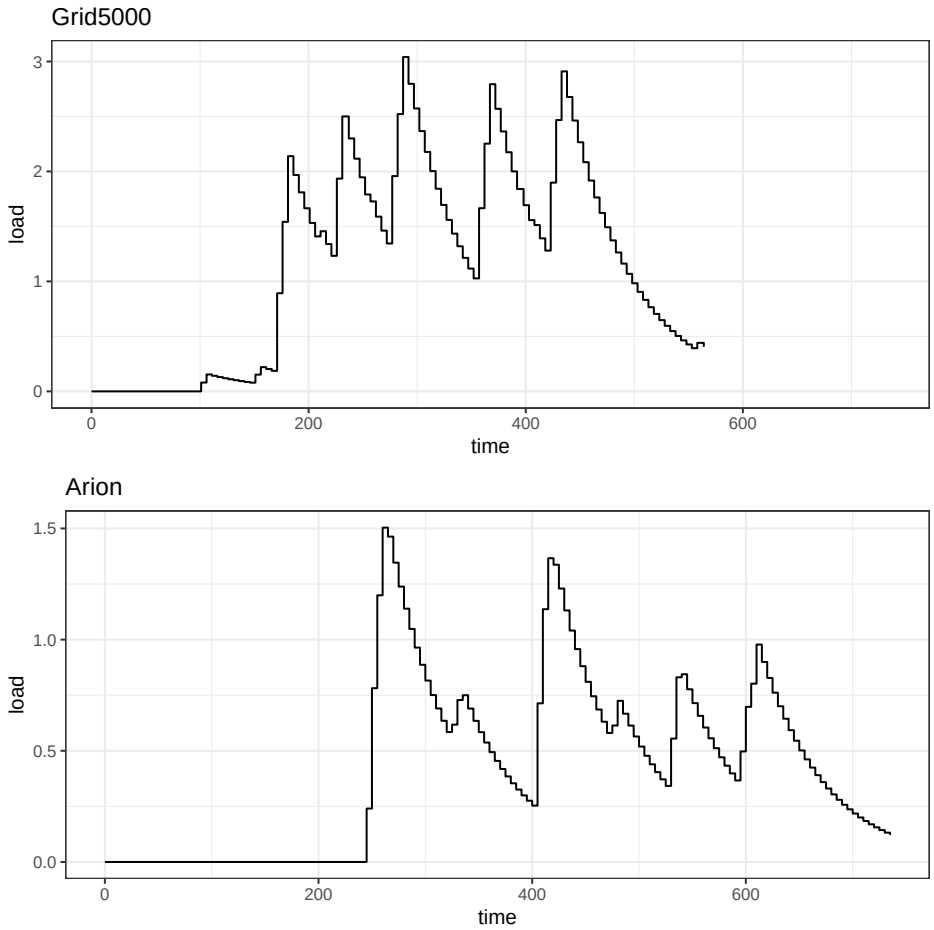

Figure 4.5: Comparison of `loadavg` using a deployed enviroment (top) and using *Arion* with containers (bottom)

This represents the periods where the jobs are writing to the fileserver.

Next, we can see that the peaks are not of the same amplitude relative to each other. This can be explained by the fact that the *OAR* scheduler is undeterministic, and thus this could result to different writing times for the jobs, leading to a different management of the processes by the fileserver and thus a different load values.

Finally, we can see that the load values for *Arion* are smaller than the load values on a deployed system.

We suppose that this difference is due to the fact that for each Grid5000 machine there are other NFS servers. Indeed, the `home` directory of the user, for example, is a mounted with NFS to the node given to the user. Thus, there could be perturbations on the load value, whereas on systems using *Arion*, there are no perturbations caused by other NFS servers.

4.5 Reproducible Experimental Workflow

The steps required to run an experiment on Grid5000 are presented in Section B.1. These steps can be intimidating for non computer scientists, like people from Gipsa Lab

who contributed on this project. With the objective to simplify the process of running experiments and making experiments reproducible, we present in this Section some of the work done to reach this goal.

4.5.1 Experiment Script

The first solution that we can think of is to write script encapsulating the necessary commands (see Section B.1).

This script would have to be executed on the Grid5000 frontend as it would be difficult to manage nested SSH connections from a personal computer.

The advantage of scripting the experimental process is that the user does not have to write as many commands herself.

We thus developed a script managing all the steps presented previously. The script would:

1. Reserve and deploy the nodes
2. Configure the *CiGri* node according to the experiment
3. Submit the campaign to *CiGri*
4. Sleep until the campaign is over
5. Copy the log files to the user storing directory
6. Release the Grid5000 resources

The main drawback of this approach is the lack of interactivity with the experiment. For example, *CiGri* provides a application to know the progress of the campaign. The user might want to know if this campaign is worth continuing if it is set to not finish before the end of the reservation.

4.5.2 Computational Documents

An alternative to using black box scripts and writing cryptic commands is *Literate Programming* [17]. The goal of *Literate Programming* is to explain to humans what we want the computer to do rather than just giving instructions to the machine.

Computational Documents, or runnable notebooks, are a form of Literate Programming. Nowadays, there are several tools for writing such documents. The most famous ones are Jupyter[3] (Python), RMarkdown[4] (R) and, the mighty, Org-mode[5]. Computational documents can contain the explanation of the experimental workflow described in formatted text, as well as the runnable workflow itself. It is also possible to generate images and graphs from the results of the notebook.

During this work we used exclusively Org-mode. We developed a computational document to run our different experiments. The notebook can connect using the SSH protocol to the Grid5000 frontend and execute the commands required for the experiment.

[3] `https://jupyter.org`
[4] `https://rmarkdown.rstudio.com`
[5] `https://orgmode.org/`

An interesting feature of Org-mode allows us to manage nested SSH session from the host of the user, making thus unnecessary to open the command line.

This approach is however not error-proof. Indeed, there might be some errors during the experiment, either with the experiment itself or a broken SSH pipe. In this case, the user would need to go troubleshooting using the commnand line. The nested SSH sessions, while allowing a simpler connection to the end nodes, do not always manage to return the output of commands executed on the nodes. To ensure a consistent output of the commands we might want use some "tricks", but with the cost of complexifiying the code of the notebook.

Another solution would be to implement the runnable notebook at the level of the Grid5000 frontend. But this would require the user to connect using SSH and manipulate a version of Emacs without GUI[6] (grid5000 does not have a X server), which might be intimidating. Besides, the user would still need to copy the files from Grid5000 to her personal machine using the `scp`[7] command.

4.6 Conclusion

The reproducibility aspect of the project is still at the early stages.

Concerning the container approach, we still need to find a good metric with valuable information that would transfer nicely between Grid5000 and *Arion*. It could be a variation of the `loadavg` metric, or a complety different one, like the ones supplied by `cgroups`. We also would like to be able to deploy the container approach on different machines using the *NixOps* project, presented in Section 2.3.3, with the idea of having a single definition of the system, using the *Nix* description language, and being able to experiment both on a single machine (*Arion*) and on multiple machines (*NixOps*).

Regarding the experimental workflows, there is room for improvement. It would be interesting to integrate version control (i.e. Git) to the runnable notebook as presented by Stanisic et al. [24].

[6]Graphical User Interface

[7]Secure CoPy

— 5 —
Conclusion

Regulating complex systems such as HPC is a difficult task. However, Control Theory provides interesting tools to approach this issue.

In this work, we have proposed a controller with a new approach to improve the granularity of the control on the load of the fileserver. This controller is a proof of concept to build on, but showed nevertheless a promising behaviour.

We also investigate the transition from a deployed system, on an infrastructure such as Grid5000, to a container based solution, with in mind the goal to improve the reproducibility of the experimentations. Besides, we developed some tools to help non computer scientists carrying out their own experiments.

We would like to also note that we did participated to the implementation of controllers proposed by control theorists from Gipsa Lab.

Future Work

A crucial task in the near future would be to complete the transition to a container approach with Arion. This would require to find a metric able to return a sense of overload working on both a deploy setup as well as on a container setup. The goal would then be to look into the NixOps project (see Section 2.3.3) to be able to deploy NixOS images of each node on different machine, with the idea of having a single description of the system for both Arion and NixOps.

The experiment workflows could be even smoother and more rigorous. Integrating a version control to the runnable notebook [24] would be a great feature for improved traceability of the experimental results.

The controller proposed during this work could be improved by trying more complex types of controllers.

In this work, we focused on the load and the fileserver, and indirectly the number of used resources, but with control theory and the autonomic computing approach, the perspectives are endless. We can think about controlling the number of jobs sent from CiGri with respect to the energy consumption of the cluster for example.

— A —
Appendix - Control

A.1 Designing a Controller

A.1.1 Choice of the Process Variable

The first step of designing a controller is to know what we want to control. The Process Variable (PV) corresponds to the metric we want to regulate and is the output of what is called the Plant (or the Process).

In our case, we want to regulate the number of jobs running on the cluster. But we also want to keep the load of the fileserver as low as possible. We thus have multiple objectives.

Control Theory provides several means to deal with multi-objectives controllers. One strategy could be to associate different weights to each objective, leading to prioritize some over the others.

The role of computer scientists during this step is to decide what we want to control. And if we wish to regulate several Process Variables, which ones should the controller prioritize.

For this work, we decided to go with the number of jobs on the cluster and the load of the fileserver as the Process Variables. And we decided to give a greater importance to the load on the fileserver. The reason behind this decision is because we do not want to affect the jobs of the regular users of the cluster. Indeed, an overloaded fileserver, is a slow fileserver. If a regular user's job requires to read, or write, a file on the server, it might affect the execution time of the job and not terminate before its deadline.

A.1.2 Choice of the Sensor

Once we know the Plant to control, we have to find a sensor for the Process Variable. The role of the sensor will be to yield a metric for the Plant.

In this work, we have two PV (number of jobs running on the cluster and the load of the fileserver). We thus need two sensors: one for each PV.

For the number of jobs running on the cluster, the OAR scheduler provides an API[1]. Our controller can thus send a request to the OAR API to get this metric.

Concerning the load of the fileserver, there are several possible sensors. We could choose the CPU percentage of the server or the percentage of memory used. However,

[1]Application Programming Interface

even if these metrics provide us with the current status of the fileserver, they do not allow for a forecasting of the state of the machine. As the controller will submit jobs to the scheduler based on the value returned by the sensor, once the jobs will start running on the cluster, the state of the fileserver will be completely different. We are therefore looking for a metric able to give the controller a hint on what will be the state of the fileserver in the future. The `loadavg` metric, described in Section 4.4.1, does respond to this description.

A.1.3 Choice of the Controller

Now that we have the processes to control and metrics given by sensors, we can choose the controller we want to regulate our system.

The choice of the controller depends on the fact that we have a model of our system or not.

If we do not have a model, then we can use Proportional Integral Derivative Controllers (PID Controllers). These controller do not require a model to give good results.

In our case, we do have a model, see Section 3.1.2. With this model, control theorist can now use more advanced controllers.

Proportional Controller

The Proportional Controller, or P Controller, is one of the simplest. It can be expressed by the following equation:

$$u(t) = K_p e(t) \tag{A.1}$$

where K_p is called the gain, $e(t)$ is the error at time t and $u(t)$ is the output of the controller.

Proportional-Integral Controller

The PI Contoller integrates a memory component. It can be modelled as:

$$u(t) = K_p e(t) + K_i \int_0^t e(x)dx \tag{A.2}$$

with K_i being the gain of the integral part.

The idea behind having an integral part is to have a more aggresive controller the more it has been wrong in the past, as the intergral term represents the cumulative error since time $t = 0$. Indeed, if the controller has been wrong in the past, the integral part will grow and thus have a bigger impact on the next values. A strategy could be to reinitilize the integral term once the error is "small enough" to stabilize the system.

Unfortunalety, in computer science, there is no notion of continuous time. Thus, we have to discretise the previous equation:

$$u(t_k) = K_p e(t_k) + K_i \sum_{i=0}^{k} e(t_i)\delta t \tag{A.3}$$

Proportional-Integral-Derivative Controller

The PID Controller adds a derivative term to the PI Controller. The interest is to predict the behaviour of the system and to anticipate it. It can be expressed as:

$$u(t) = K_p e(t) + K_i \int_0^t e(x)dx + K_d \frac{de}{dt}(t) \tag{A.4}$$

We can discretise the previous equation:

$$u(t_k) = K_p e(t_k) + K_i \sum_{i=0}^{k} e(t_i)\delta t + K_p \frac{e(t_k) - e(t_{k-1})}{\delta t} \tag{A.5}$$

Model Predictive Control

Principle A controller using the model predictive scheme will compute all possible input values and use the model of the system to keep the best one. To find the optimal values, the controller will use the model of the system to make prediction based on an inital input and the current state of the system. It will make several iterations of the model in order to make a better prediction for the future. The number of iterations is called the horizon, we will note it N in the following. Using a greater horizon will result in a better prediction and choice of the number of jobs to sumbit. However, the drawback to a great horizon is the computational time required to make the prediction. There is thus a trade-off between a great prediction and a reasonable computational time.

Maximize Cluster Usage We can guarantee maximizing the usage of the cluster by providing enough waiting jobs (q_t) to the OAR scheduler. We can achieve this by controlling with respect to a reference q_{ref}.

The cost function associated with the objective is defined as:

$$J_t^1(u) = \sum_{j=0}^{N} \left(q_{t+j\Delta t|t} - q_{ref}\right)^2 \tag{A.6}$$

where $q_{t+j\Delta t|t}$ represents the value returned by the model for the number of waiting jobs after j iterations of the model starting from the state q_t with the input u.

Fileserver Overload In a similar fashion as in the previous paragraph, we can define a cost function for the load of the fileserver. We need to define a reference to compare the current load of the system to. A rule-of-thumb of system administrators specifies that a system is overloaded if the `loadavg` is greater than the number of cores of the system. We thus define f_{ref} as the number of cores in the fileserver.

The cost function is thus defined as:

$$J_t^2(u) = \sum_{j=0}^{N} \left(f_{t+j\Delta t|t} - f_{ref}\right)^2 \tag{A.7}$$

where $f_{t+j\Delta t|t}$ represents the value returned by the model for the load after j iterations of the model starting from the state f_t with the input u.

Multi Objectives One solution to deal with the two cost functions is to take the minimum optimal value of both cost functions.

$$u_t = \min\left(\min_{u_1} J_t^1(u_1), \min_{u_2} J_t^2(u_2)\right) \tag{A.8}$$

A.1.4 Tweaking the Controller

Once the controller has been defined, we now have to adjust the different parameters in order to improve the control.

This is done by looking at the response of the system under an "open loop". The term "open loop" means that there is no feedback. The goal is to see how the system reacts to a given input. From these results, the control theorists can adapt the values of the different parameters to better suit the controller to the system.

In our case, we generate an impulse input. We make CiGri submit all the jobs of the campaign at once, and we look at the behaviour of the system (OAR scheduler + cluster + filerserver).

A.2 Model Yabo et al.

$$\begin{cases} q_{t+\Delta t} = q_t + u_t - b_t \\ r_{t+\Delta t} = r_t + b_t - p\Delta t r_t \end{cases} \tag{A.9}$$

— B —
Appendix - Reproducibility

B.1 Steps to run an Experiment

In this Section we present the steps to run an experiment on Grid5000 and the key moments.

B.1.1 Step 1: Connect to the Grid5000 frontend

As the experiments are done on Grid5000, the user has to connect to the frontend of the grid to reserve the nodes.

This step introduces the user with the **SSH Protocol**[1] and thus, also to the command line. It might be daunting for a regular Windows user who is probably used to graphical interfaces.

B.1.2 Step 2: Make a reservation and deploy

Once on the frontend, the user needs to make a reservation to the OAR scheduler for the number of nodes required for the experiment. Then the user would need to deploy the correct image to the nodes to ensure the reproducibility behaviour of the experiment.

This step would require a sequence of commands using the CLI[2] that the user might not be at ease with. Besides, having an unexperienced user writing multiple commands by hand increases the probability of mistakes.

B.1.3 Step 3: Connect to the nodes and configure them

Once the nodes are deployed and running, the user now has to set up the nodes (mainly the CiGri node, and as `root`) for the experiment she would like to run.

This would require to connect, via SSH, to the nodes from the fontend. Then make the changes to the CiGri source code (by coping a file) and restart CiGri using the `systemctl`[3] command.

[1] Secure SHell Protocol

[2] Command Line Interface

[3] `https://man7.org/linux/man-pages/man1/systemctl.1.html`

It might also happen that some processes did not start properly. In this case, the user would have to investigate and restart the correct services using again the `systemctl` command.

B.1.4 Step 4: Run the experiment

Finally the user has step up all the system and now is ready to run the experiment.

In order to do so, she needs to connect again to the CiGri node (but not as `root`, which might be confusing). Once on the CiGri node, a final command starts the experiment.

B.1.5 Step 5: Gather the log of the experiment

When the experiment is done, the user would like to get back some of the log files generated. However, those might be in a directory on the node that is not in the user personal filesystem subtree (e.g. `/tmp`), and thus will not be accessible on the frontend. The user would have to copy the log files to a correct location.

Now that the log files are on the Grid5000 frontend, the user would need to get them to their personal computer to make the analysis. There are several ways to do this.

The first one would be to use a version control application like Git. The user would commit the log files to a repository and pull them on her personal computer. This solution requires a basic knowledge of version control, which is a lot to ask to a non computer scientist.

The second solution would be to use the `scp`[4] command. But this would mean one more command to write.

B.2 `arion-compose.nix` Listing

We show here the complete listing of the `arion-compose.nix` file which generate the configuration file for `docker-compose`.

```
{ pkgs, lib,... }:
let
  inherit (import ../common/ssh-keys.nix pkgs) snakeOilPrivateKey snakeOilPublicKey;
common = {
  service.volumes = [ "${builtins.getEnv "PWD"}/..:/srv" ];
  service.capabilities = { SYS_ADMIN = true; }; # for nfs
  service.useHostStore = true;

  nixos.useSystemd = true;
  nixos.runWrappersUnsafe = true;
  nixos.configuration = {
    networking.firewall.enable = false;
    boot.tmpOnTmpfs = true;

    users.users.user1 = {isNormalUser = true;};
```

[4]Secure CoPy

```
    users.users.user2 = {isNormalUser = true;};

    environment.systemPackages = with pkgs; [ nfs-utils socat wget ruby openssh ];
    imports = lib.attrValues pkgs.nur.repos.kapack.modules;

    # oar user's key files
    environment.etc."privkey.snakeoil" = { mode = "0600"; source = snakeOilPrivateKey
    environment.etc."pubkey.snakeoil" = { mode = "0600"; source = snakeOilPublicKey; }

    services.oar = {
      # oar db passwords
      database = {
        host = "server";
        passwordFile = "/srv/common/oar-dbpassword";
      };
      server.host = "server";
      privateKeyFile = "/etc/privkey.snakeoil";
      publicKeyFile = "/etc/pubkey.snakeoil";
    };
  };
};

addCommon = x: lib.recursiveUpdate x common;

 apacheHttpdWithIdent = pkgs.apacheHttpd.overrideAttrs (oldAttrs: rec {
  configureFlags = oldAttrs.configureFlags ++ [ "--enable-ident" ]; });

  defineNode = name: resources:
    addCommon {
      service.hostname=name;
      nixos.configuration = {
        services.oar.node = {
          enable = true;
          register = {
            enable = true;
            extraCommand = ''
              /srv/common/prepare_oar_cgroup.sh init
              mkdir -p /mnt/shared
              /run/current-system/sw/bin/mount -t nfs fileserver:/srv/shared /mnt/sha
            '';
            nbResources = resources;
          };
        };
      };
    };
in
```

```
{
  services.cigri = addCommon {
    service.hostname="cigri";
    nixos.configuration = {
      services.cigri = {
        dbserver.enable = true;
        client.enable = true;
        database = {
          host = "cigri";
          passwordFile = "/srv/common/cigri-dbpassword";
        };
        server = {
          enable = true;
          web.enable = true;
          host = "cigri";
          logfile = "/tmp/cigri.log";
        };
      };
      services.my-startup = {
        enable = true;
        path = with pkgs; [ nur.repos.kapack.cigri sudo postgresql ];
        script = ''
          # Waiting cigri database is ready
          until pg_isready -h cigri -p 5432 -U postgres
          do
            echo "Waiting for postgres"
            sleep 0.5;
          done

          until sudo -u postgres psql -lqt | cut -d \| -f 1 | grep -qw cigri
          do
            echo "Waiting for cigri db created"
            sleep 0.5
          done

          newcluster cluster_0  http://server/oarapi-unsecure/ none fakeuser fakepass
          systemctl restart cigri-server
        '';
      };
    };
  };

  services.frontend = addCommon {
    service.hostname="frontend";
    nixos.configuration = {
      services.oar.client.enable = true;
```

```
    };
  };

  services.server = addCommon {
    service.hostname="server";
    nixos.configuration = {
      environment.etc."oar/api-users" = {
        mode = "0644";
        text = ''
          user1:$apr1$yWaXLHPA$CeVYWXBqpPdN78e5FvbY3/
          user2:$apr1$qMikYseG$VL8nyeSSmxXNe3YDOiCwr1
        '';
      };
      services.oar.dbserver.enable = true;
      services.oar.server = {
        enable = true;
      };

      services.oar.web = {
        enable = true;
        extraConfig = ''
          #To support remote_ident custom header
          underscores_in_headers on;
          location ^~ /oarapi-unsecure/ {
            rewrite ^/oarapi-unsecure/?(.*)$ /$1 break;
            include ${pkgs.nginx}/conf/uwsgi_params;
            uwsgi_pass unix:/run/uwsgi/oarapi.sock;
          }
        '';
      };
    };
  };

  services.fileserver = addCommon {
    service.hostname="fileserver";
    nixos.configuration = {
      services.nfs.server.enable = true;
      services.nfs.server.exports = ''/srv/shared *(rw,sync,no_subtree_check,no_root_
    };
  };
  services.node1 = defineNode "node1" "100";
}
```

Bibliography

[1] Dependency Hell, . URL https://en.wikipedia.org/wiki/Dependency_hell.

[2] Folding@home takes up the fight against COVID-19 / 2019-nCoV - Folding@home, . URL https://foldingathome.org/2020/02/27/foldinghome-takes-up-the-fight-against-covid-19-2019-ncov/.

[3] Folding@home - Fighting disease with a world wide distributed super computer., . URL https://foldingathome.org/.

[4] SETI@Home - Detailed stats | BOINCstats/BAM!, . URL https://www.boincstats.com/stats/0/project/detail/.

[5] Abdul-Hafeez Ali, Eric Rutten, Bogdan Robu, and Olivier Richard. Controlling Computer Cluster Overload: Intelligent Control Using Nonsupervised Learning. page 38.

[6] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, November 2002. ISSN 00010782. doi: 10.1145/581571.581573. URL http://portal.acm.org/citation.cfm?doid=581571.581573.

[7] D.P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Fifth IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Pittsburgh, PA, USA, 2004. IEEE. ISBN 978-0-7695-2256-2. doi: 10.1109/GRID.2004.14. URL http://ieeexplore.ieee.org/document/1382809/.

[8] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounie, P. Neyron, and O. Richard. A batch scheduler with high level components. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, pages 776–783 Vol. 2, Cardiff, Wales, UK, 2005. IEEE. ISBN 978-0-7803-9074-4. doi: 10.1109/CCGRID.2005.1558641. URL http://ieeexplore.ieee.org/document/1558641/.

[9] Walfredo Cirne, Francisco Brasileiro, Nazareno Andrade, Lauro B. Costa, Alisson Andrade, Reynaldo Novaes, and Miranda Mowbray. Labs of the World, Unite!!! *Journal of Grid Computing*, 4(3):225–246, September 2006. ISSN 1570-7873, 1572-9184. doi: 10.1007/s10723-006-9040-x. URL http://link.springer.com/10.1007/s10723-006-9040-x.

[10] Christian Collberg, Todd Proebsting, and Alex M Warren. Repeatability and Benefaction in Computer Systems Research - A Study and a Modest Proposal. page 68, 2015.

[11] Ludovic Courtes and Ricardo Wurmus. Reproducible and User-Controlled Software Environments in HPC with Guix. In Sascha Hunold, Alexandru Costan, Domingo Giménez, Alexandru Iosup, Laura Ricci, Maria Engracia Gomez Requena, Vittorio Scarano, Ana Lucia Varbanescu, Stephen L. Scott, Stefan Lankes, Josef Weidendorfer, and Michael Alexander, editors, *Euro-Par 2015: Parallel Processing Workshops*, volume 9523, pages 579–591. Springer International Publishing, Cham, 2015. ISBN 978-3-319-27307-5 978-3-319-27308-2. doi: 10.1007/978-3-319-27308-2_47. URL http://link.springer.com/10.1007/978-3-319-27308-2_47.

[12] Ludovic Courtès. Functional Package Management with Guix. *arXiv:1305.4584 [cs]*, May 2013. URL http://arxiv.org/abs/1305.4584. arXiv: 1305.4584.

[13] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A Safe and Policy-Free System for Software Deployment. page 14, 2004.

[14] Domenico Ferrari and Songnian Zhou. An Empirical Investigation of Load Indices For Load Balancing Applications. *Proceedings of Performance '87*, the 12th International Symposium on Computer Performance Modeling(Measurement, and Evaluation):515–528, 1988. URL https://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/CSD-87-353.pdf.

[15] Yiannis Georgiou, Olivier Richard, and Nicolas Capit. Evaluations of the Lightweight Grid CIGRI upon the Grid5000 Platform. In *Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*, pages 279–286, Bangalore, India, 2007. IEEE. ISBN 978-0-7695-3064-2. doi: 10.1109/E-SCIENCE.2007.32. URL http://ieeexplore.ieee.org/document/4426898/.

[16] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36:41–50, January 2003. URL http://pages.cs.wisc.edu/swift/classes/cs736-fa06/papers/autonomic-computing.pdf.

[17] Donald Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, September 1984. doi: https://doi.org/10.1093/comjnl/27.2.97. URL http://www.literateprogramming.com/knuthweb.pdf.

[18] Marin Litoiu, Mary Shaw, Gabriel Tamura, Norha M. Villegas, Hausi A. Müller, Holger Giese, Romain Rouvoy, and Eric Rutten. What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems? In Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, editors, *Software Engineering for Self-Adaptive Systems III. Assurances*, volume 9640, pages 90–134. Springer International Publishing, Cham, 2017. ISBN 978-3-319-74182-6 978-3-319-74183-3. doi: 10.1007/978-3-319-74183-3_4. URL http://link.springer.com/10.1007/978-3-319-74183-3_4.

[19] Michael Mercier. *Contribution to High Performance Computing and Big Data Infrastructure Convergence*. PhD Thesis, Universite Grenoble Alpes, 2019.

[20] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! page 12.

[21] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. Reconstructable Software Appliances with Kameleon. *ACM SIGOPS Operating Systems Review*, 49(1):80–89, January 2015. ISSN 0163-5980. doi: 10.1145/2723872.2723883. URL `https://dl.acm.org/doi/10.1145/2723872.2723883`.

[22] Eric Rutten, Nicolas Marchand, and Daniel Simon. Feedback Control as MAPE-K Loop in Autonomic Computing. In Rogério de Lemos, David Garlan, Carlo Ghezzi, and Holger Giese, editors, *Software Engineering for Self-Adaptive Systems III. Assurances*, volume 9640, pages 349–373. Springer International Publishing, Cham, 2017. ISBN 978-3-319-74182-6 978-3-319-74183-3. doi: 10.1007/978-3-319-74183-3_12. URL `http://link.springer.com/10.1007/978-3-319-74183-3_12`.

[23] Emmanuel Stahl, Augustin Yabo, Olivier Richard, Bruno Bzeznik, Bogdan Robu, and Eric Rutten. Towards a control-theory approach for minimizing unsued grid resources. *AI-Science'18 - workshop on Autonomous Infrastructure for Science, in conjunction with the ACM HPDC 2018*, pages 1–8, June 2018. doi: 10.1145/3217197.3217201. URL `https://hal.archives-ouvertes.fr/hal-01823787/`.

[24] Luka Stanisic, Arnaud Legrand, and Vincent Danjean. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *ACM SIGOPS Operating Systems Review*, 49(1):61–70, January 2015. ISSN 0163-5980. doi: 10.1145/2723872.2723881. URL `https://dl.acm.org/doi/10.1145/2723872.2723881`.

[25] Agustin Gabriel Yabo, Bogdan Robu, Olivier Richard, Bruno Bzeznik, and Eric Rutten. A control-theory approach for cluster autonomic management: maximizing usage while avoiding overload. In *2019 IEEE Conference on Control Technology and Applications (CCTA)*, pages 189–195, Hong Kong, China, August 2019. IEEE. ISBN 978-1-72812-767-5. doi: 10.1109/CCTA.2019.8920473. URL `https://ieeexplore.ieee.org/document/8920473/`.