Methods and tools for challenging experiments on Grid'5000 : a use case on electromagnetic hybrid simulation

Cristian Ruiz

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Présentée par

Cristian RUIZ

Thèse dirigée par **Olivier Richard**
et codirigée par **Thierry Monteil**

Préparée au sein du **LIG, Laboratoire d'Informatique de Grenoble**
et de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Methods and Tools for Challenging experiments on Grid'5000: a use case on electromagnetic hybrid simulation

Thèse soutenue publiquement le **15 décembre 2015**,
devant le jury composé de :

M. Emmanuel Jeannot
Directeur de recheche à INRIA, Président
M. Frederic Desprez
Directeur de recherche à INRIA, Rapporteur
Mme. Kate Keahey
Scientist and Senior Fellow à Argonne National laboratory, Rapporteur
M. Yves Denneulin
Professeur à Grenoble INP, Examinateur
M. Olivier Richard
Maitre de conference, LIG, Directeur de thèse
M. Thierry Monteil
Maitre de conference, LAAS-CNRS, Co-Directeur de thèse

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my family. They were always supporting me, creating the perfect conditions for me to be able to succeed in this endeavor. Without their support this thesis would have not be possible.

My highest appreciation to Christiane Tron-Siaud who shared my happiness, my sadness and made the impossible to help me out. I would like to thank Erick Meneses and Carlos Jaime Barrios for their support and friendship that were important to endure the hard moments and specially for giving me the idea of pursuing my studies in this field.

This thesis would have not been possible without the guidance, availability of my advisor Olivier Richard who was always there to discuss and give me valuable feedback. I really enjoy working with him and I really appreciated his sense of humor. I would like to thank Thierry Monteil my other advisor for his invaluable help during this thesis who, in spite of, the distance was present when I needed him.

I would like to thank Tomasz Buchert, Lucas Nussbaum, Mihai Alexandru, Joseph Emeras who contributed directly to this thesis by co-authoring some research papers. A special thanks goes to Salem Harrache, Michael Mercier, Pierre Neyron and Bruno Bzeznik who contributed to this thesis by helping me out with technical issues and by creating an excellent atmosphere for working. I would like also to thank INRIA for funding this thesis.

Finally, nothing would have meaning if I did not have the support of my friends who always were there to give me a hand.

Abstract

In the field of Distributed Systems and High Performance Computing experimental validation is heavily used against an analytic approach. The latter is not feasible any more due to the complexity of those systems in terms of software and hardware. Therefore, researchers have to face many challenges when conducting their experiments, making the process costly and time consuming. Although world scale platforms exist and virtualization technologies enable to multiplex hardware, experiments are most of the time limited in size given the difficulty to perform them at large scale. The level of technical skills required for setting up an appropriate experimental environment is rising with the always increasing complexity of software stacks and hardware nowadays. This in turn provokes that researchers in the pressure to publish and present their results use *ad hoc* methodologies. Hence, experiments are difficult to track and preserve, preventing future reproduction.

A variety of tools have been proposed to address this complexity at experimenting. They were motivated by the need to provide and encourage a sounder experimental process, however, those tools primary addressed much simpler scenarios such as single machine or client/server. In the context of Distributed Systems and High Performance Computing, the objective of this thesis is to make complex experiments, easier to perform, to control, to repeat and to archive.

In this thesis we propose two tools for conducting experiments that demand a complex software stack and large scale. The first tool is called *Expo* that enables to efficiently control the dynamic part of an experiment which means all the experiment workflow, monitoring of tasks, and collection of results. *Expo* features a description language that makes the set up of an experiment with distributed systems less painful. Comparison against other approaches, scalability tests and use cases are shown in this thesis which demonstrate the advantage of our approach. The second tool is called *Kameleon* which addresses the static part of an experiment, meaning the software stack and its configuration. *Kameleon* is a software appliance builder that enables to describe and control all the process of construction of a software stack for experimentation. The main contribution of *Kameleon* is to make easier the setup of complex software stacks and guarantee its post reconstruction.

Résumé

Dans le domaine des systèmes distribués et du calcul haute performance, la validation expérimentale est de plus en plus utilisé par rapport aux approches analytiques. En effet, celles-ci sont de moins en moins réalisables à cause de la complexité grandissante de ces systèmes à la fois au niveau logiciel et matériel. Les chercheurs doivent donc faire face à de nombreux challenges lors de la réalisation de leurs expériences rendant le processus coûteux en ressource et en temps. Bien que de larges plateformes parallèles et technologies de virtualisation existent, les expérimentations sont, pour la plupart du temps, limitées en taille. La difficulté de passer une expérimentation à l'échelle représente un des grands facteurs limitant. Le niveau technique nécessaire pour mettre en place un environnement expérimentale approprié ne cesse d'augmenter pour suivre les évolutions des outils logiciels et matériels de plus en plus complexes. Par conséquent, les chercheurs sont tentés d'utiliser des méthodes ad-hoc pour présenter des résultats plus rapidement et pouvoir publier. Il devient alors difficile d'obtenir des informations sur ces expérimentations et encore plus de les reproduire.

Une palette d'outils ont été proposés pour traiter cette complexité lors des expérimentations. Ces outils sont motivés par le besoin de fournir et d'encourager des méthodes expérimentales plus construites. Cependant, ces outils se concentrent principalement sur des scénarios très simple n'utilisant par exemple qu'un seul noeud ou client/serveur. Dans le contexte des systèmes distribués et du calcul haute performance, l'objectif de cette thèse est de faciliter la création d'expériences, de leur contrôle, répétition et archivage.

Dans cette thèse nous proposons deux outils pour mener des expérimentations nécessitant une pile logicielle complexe ainsi qu'un grand nombre de ressources matérielles. Le premier outil est Expo. Il permet de contrôler efficacement la partie dynamique d'une expérimentation, c'est à dire l'enchaînement des tests expérimentaux, la surveillance des taches et la collecte des résultats. Expo dispose d'un langage de description qui permet de mettre en place une expérience dans un contexte distribué avec nettement moins de difficultés. Contrairement aux autres approches, des tests de passage à l'échelle et scénarios d'usage sont présentés afin de démontrer les avantages de notre approche. Le second outil est appelé Kameleon. Il traite les aspects statiques d'une expérience, c'est à dire la pile logicielle et sa configuration. Kameleon est un logiciel qui permet de décrire et contrôler toutes les étapes de construction d'un environnement logiciel destiné aux expérimentations. La principale contribution de Kamelon est de faciliter la construction d'environnements logiciels complexes ainsi que de garantir de futur reconstructions.

Part I

Introduction

Chapter 1

Introduction

Beware of bugs in the above code; I have only proved it correct, not tried it. – Don Knuth

If I have seen further it is by standing on the shoulders of giants - Isaac Newton -

Natural sciences have created instruments [1] and develop methodologies [92] for carrying out a more sound experimental process that follows the scientific method and assure that the results can be validated. Nowadays, computers are the support for scientific discoveries in natural sciences which spans areas from particle physics to astronomy and cosmology. Computers are mostly used for performing data analysis and carrying out simulations [2]. In view of the increasing complexity of this data-driven process, computational scientific workflows have been adopted as a tool for improving and automating the experimentation activity [90]. They cover different phases of the science process: hypothesis formation, experiment design, execution, and data analysis. Recently computational scientific workflows and data provenance techniques have received special attention [36] due the need for *Reproducible research* that make a call for results reproducibility, sharing and knowledge re-use in the scientific community. Likewise, research based mainly on data analysis and simulation of natural phenomena such as image processing, geophysics, bioinformatics, signal processing, neuroscience, etc have been creating a set of tools [37, 54, 43, 100] that help to achieve reproducibility of their results.

A tendency can be observed for improving the experimental methodologies when using computers at the service of science and we should expect the same for pure computer science. *Distributed systems* in general and *High performance computing* in particular rely heavily on experimentation, given that it is difficult to study those systems using an analytic approach [121, 59, 66]. Unfortunately, there is a lack of methodologies and tools to conduct experiments with distributed systems as expressed in [70], making experimenters use *ad hoc* approaches that are hardly reproducible. This can be explained by the fact that there exist more challenges when our object of study is the same computer system and experiment results and research conclusions are dependent on the most minimal detail of the software and hardware stack.

In [32] the process of repeating an experiment was carefully studied and among the many conclusions drawn, the difficulty of repeating published results was highly relevant. There could be many reasons that hamper the *Reproducibility/Repetability* of experiments presented in a paper. For example, the buildability of artifacts, a recent study [30] found that roughly only 25% of publications in ACM conferences and journals can be built. Another reason is the measurement bias. In [93] it was shown that seemingly changes in the experimental setup such as Linux environment size can influence the apparent performance of applications. The low quality of experiments in *Distributed systems* and *High performance computing* could be explained by the constant and fast evolution of computer hardware and software.

[1] The Large Hadron Collider (LHC), so far the biggest scientific instrument build by humans.
[2] Which is normally called in-silico science

Testbeds have been created to study different kinds of distributed systems by offering controlled conditions. Thanks to the evolution of *virtualization*, resource sharing has been possible enabling to build planet scale testbeds [103] that expose real network conditions. Different forms of *emulation* have made possible to achieve large scale while offering more controlled conditions [130]. Other testbeds enable the whole software stack to be reconfigured [25]. In short, the decrease in the price of off-the-shelf hardware and the evolution of *virtualization* and *emulation* technologies have provoked that testbeds grow in size and possibilities for the user, making them more complex to manage and difficult to take full advantage of them.

The conduction of experiments with distributed systems presents many challenges. First, the increasing number of software layers and their configuration. Second, the complex architecture and hardware options now present. Third, the scale of distributed systems which could go from a simple network of caching servers to a big computational cluster with thousands of nodes. Those challenges make the task of designing, description, setup, management, results collection, etc, very complex. In order to ease the experimentation processes, make it less expensive and assure the quality of the experiment (which comprehends two important properties like *Reproducibiliy* and *Repeatability*), each testbed have endorsed the development of tools that help the users with the process of experimentation. Those tools address the experimentation cycle differently offering important features such as failure handling and large deployment [5], manage of the whole experimental cycle with distributed systems and workload generation [126], versioning system to allow researchers move forward and backward through their experimentation process [47], abstractions to manage the increasing number of nodes [124], instrumentation facilities for applications [107], etc. Cloud based testbeds have motivated the apparition of generic APIs for scripting experiments [10, 67] that enable the use of all kinds of language constructs, such as loops, exception blocks, etc. More recently, a workflow approach inspired in the domain of business process management is envisioned as a new alternative to manage large scale experiments [20].

There has been an evolution on the description language going from inflexible markup languages like XML to the now widely used scripting languages such as Ruby [3] and Python [4]. The scalability has been addressed by improving mechanisms to control experiments and federate multiple testbeds. The right level of abstraction is still missing, making descriptions too verbose or with a high learning curve. Repeatability of experiments (which has been a driving force for those tools) seems far from achieved. Software stacks used for distributed systems have become very complex. They are composed of different interrelated layers that are in a constant change. Therefore, the setup of an experiment is not guarantee to be repeatable. This thesis proposes two tools targeted at making easy mainly the setup and execution of experiments with distributed systems. Nowadays, the number of testbeds that enable to control the whole software stack has risen. Either by adopting *cloud computing* technologies [51] or provisioning systems on real hardware [25]. We take advantage of the previous fact and propose an appliance builder to build, track and preserve the software stack used in an experiment, avoiding when possible the dependency on external sources. For management and automation of the experimental workflow with distributed systems, an experiment management tool is proposed that relies on a lightweight architecture and provides to the user a domain specific language that brings an appropriate level of abstraction, lowering the learning curve, providing conciseness and an efficient mapping to the platform.

1.1 Experimental cycle

In order to better explain the challenges encountered when conducting experiments with distributed systems and to make clear the contributions of this thesis, it is explained first the experimental cycle that is normally followed.

[3] `https://www.ruby-lang.org`
[4] `https://www.python.org/`

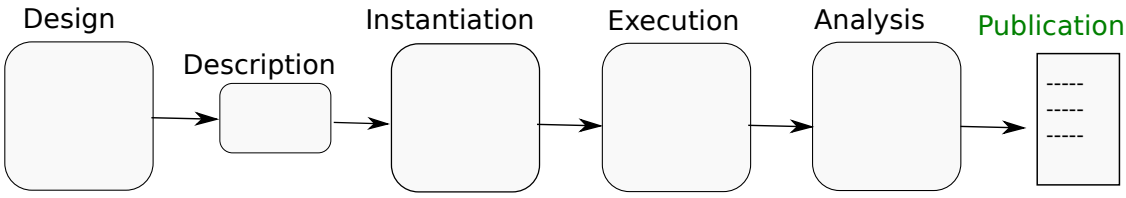

Figure 1.1: Experiment cycle with distributed systems

1.1.1 Design

Here, the experimenter decides how his/her experiment environment is going to be conformed and what actions need to be executed during the experiment. It is decided as well the measures and how those will be collected in order to have the appropriate data for answering the question that has driven the experiment. The following are some questions answered in this phase: What platform to use? how many nodes? how many different roles and how they will be mapped into the chosen nodes? what version of software to use? should it be applied some kind of workload? what measures to do and with which frequency? how many times the individual tests have to be repeated?, etc. The output of this process of decision is the experiment description.

Challenges

The goal of the description of the experiment is to have enough details of the experimentation process in order to be able to re-create or at least trace back the experiment (its provenance). Therefore, this description normally details:

- All the different software with their respective versions.
- The required computer resources and their characteristics.
- The different actions that have to be carried out (e.g., execution of an application with certain parameters)
- The number of times that is to be repeated.
- The analysis steps that are to be performed.

The challenges here is to find an appropriate way to describe an experiment that is comprehensible with a low learning curve. We have to remark that when dealing with distributed systems, the experimental scenario is complex, comprising many variables (i.e., nodes, roles, software, workload, etc).

1.1.2 Instantiation

In this stage all the experiment requirements in software and hardware are mapped into the infrastructure. First, the machines that match the experiment requirements are allocated. Then, all the necessary software is loaded into the chosen machines (provisioning) and finally the configuration of all the software stack takes place (contextualization). Software can be instrumented if needed.

Challenges

The challenges here is to find an efficient mechanism for resource discovery, to track all the information related with the software and hardware used (environment capturing) and to assure that the hardware is correctly configured.

1.1.3 Execution

In this phase, all the actions that the experimenter has planned within the experiment are carried out. The experimenter monitors the state of the experiment in order to detect errors and follow its progress.

Challenges

When dealing with distributed infrastructures there is a necessity of scaling the experiment and controlling large number of nodes. There should be a good orchestration of the experiment that enables to perform tasks at a given time, execute operations efficiently, monitor and collect results. This is done most of the time with the goal of reducing costs. Another important challenge is the capture of the platform state which could have important influence on the results of an experiment.

1.1.4 Analysis

It deals with the transformation of the raw data obtained by running the experiments in useful information and conclusions. This will be included in publications as tables and plots.

Challenges

One of the challenges is to make the process of transformation of the raw data explicit in order to be able to reproduce it without the need of re-executing the experiment.

1.2 Contributions

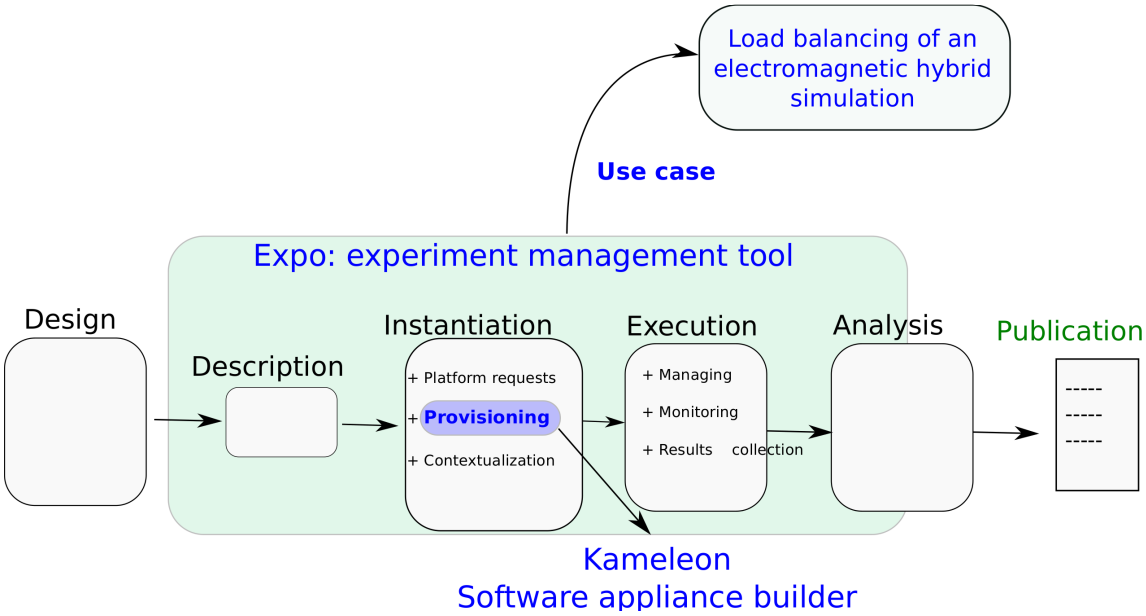

Figure 1.2: Experiment cycle proposed in this thesis

This thesis presents two tools aimed at improving the experimentation activity with distributed systems. The tools proposed, seek for rendering the process less costly, making the experimenter more efficient and improving the quality of the experiments with distributed systems. The experimental cycle is managed paying special attention to the provisioning of the experiments. Provisioning is an important part of the process of experimentation and it constantly generates issues,

making the whole process error-prone and time consuming. Experimenters could lack the appropriate computer engineering skills necessary to deal with the complexity of the software stack. For the previous reasons we opted for addressing *Provisioning* with a different tool. Additionally, in this thesis we have identified the concept of *reconstruct-ability* which we believe is essential for guaranteeing the revisability, modifiability and post-reconstruction of software artifacts employed in an experiment. This represents a step further towards experiments reproducibility with distributed systems.

The contributions of this thesis are threefold:

- A survey of experimental management tools.
- An experimental management tool for distributed systems that covers the whole experiment cycle (i.e., Design, Instantiation, Execution and Analysis).
- An appliance builder that deals with complex software stacks required for the experiments (i.e., Provisioning of experiments).

1.2.1 Survey of experimental management tools

This thesis presents a survey of the existing experimental management tools for distributed systems. Given the emergence of new tools for managing experiments with distributed systems and a significant number of publications dedicated to them, we decided to carry out an extensive literature review which led us with the following results:

- Definitions and common vocabulary.
- List of features that enables to evaluate the current experiment management tools proposed by different testbeds.
- Impact analysis of publications.

This survey could be used as a framework for evaluating existing experiment management tools. It was done in tightly collaboration with Tomasz Buchert Ph.D student in the *AlGorille* team, at *LORIA* (Nancy). This survey produced the following publication:

- Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45(0):1 – 12, 2015

1.2.2 Experiment management tool

In this thesis presents work on *Expo*. It is an experiment management engine that automates the whole experiment cycle with distributed systems. It provides a flexible description language based on two main abstractions: *ResourceSet* and *Tasks* that help the experimenter to manage large amount of nodes efficiently and specify complicated workflows for the execution part. This tool has already been proposed and presented in [125, 124]. During this thesis *Expo* has been extended, its architecture has suffered a total redesign, their abstractions have been refined and new functionalities have been added. Comparisons with existing tools were done and new use cases were found. The work with *Expo* has produced the following publications:

- Cristian Ruiz, Olivier Richard, Brice Videau, and Iegorov Oleg. Managing Large Scale Experiments in Distributed Testbeds. In *Proceedings of the 11th IASTED International Conference*, pages 628–636. IASTED, ACTA Press, feb 2013
- Cristian Ruiz, Mihai Alenxandru, Olivier Richard, Thierry Monteil, and Herve Aubert. Platform calibration for load balancing of large simulations: TLM case. In *CCGrid 2014 – The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, Illinois, USA, 2014

1.2.3 Experimental software environment

> *It should be reasonable to expect experimental setup to be reproducible. Specifically, if the infrastructure setup and the software installation and configuration can be performed in a reproducible manner then scientists are much more enabled at replicating or extending the experiment in question [84]*

This thesis presents the work on *Kameleon* that has mainly two goals: (1) make the setup of complex software stacks easier for the average user, (2) make software artifacts reconstruct-able which means they could be examined, modified and reconstructed at any time (post-experiment). It addresses a widespread problem in publications [30] and in the daily research life [57] which is the buildability of the software environment. The constant and rapid change in the different software components used nowadays, make difficult to track them and put them together to work. As a result, few experiment setups can be reused and experimenters spend a lot of time trying to build their environment for experimentation. *Kameleon* is an appliance builder already proposed in [49], during this thesis the tool was re-conceptualize and new syntax and functionalities were added. All was driven by the requirements for building complex software stacks for *Distributed systems* and *High Performance computing* research. A persistent cache mechanism was proposed and implemented that enables to preserve the software stack over time (which means it can be rebuilt at any time). This work produced the following publication:

- Cristian Ruiz, Olivier Richard, and Joseph Emeras. Reproducible software appliances for experimentation. In *Proceedings of the 9th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, Guangzhou, China, 2014
- Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. Reconstructable software appliances with kameleon. *SIGOPS Oper. Syst. Rev.*, 49(1):80–89, January 2015

1.3 Thesis organization

The thesis is divided into three parts:

- *Part I*: Introduces all the necessary terminology in order to position our contributions in the field of experimentation with distributed systems. Chapter 2 presents a survey of experimentation tools for distributed systems. It shows all the state of the art related with the tools conceived for helping users with the processes of experimentation.
- *Part II*: Presents *Expo* an experimentation tool for distributed systems. Chapter 3 shows the new concepts and design changes added during this thesis as well as an evaluation against others experiment management tools. Chapter 4 presents a use case of *Expo* that helps to deploy applications efficiently by performing a load balancing.
- *Part III*: Presents *Kameleon* an appliance builder for complex software stacks. In Chapter 5 the concept of reconstruct-ability is presented along with *Kameleon* architecture, syntax, concepts and a comparison with the most widely known appliance builders used in cloud computing. Chapter 6 is dedicated to the problematic of preserving a software stack over time.

Chapter 2

Overview of experiment management in computer science

In the field of large-scale distributed systems, experimentation is particularly difficult. The studied systems are complex, often nondeterministic and unreliable, software is plagued with bugs, whereas the experiment workflows are unclear and hard to reproduce. These obstacles led many independent researchers to design tools to control their experiments, boost productivity and improve quality of scientific results.

Despite much research in the domain of distributed systems experiment management, the current fragmentation of efforts asks for a general analysis. We therefore propose to build a framework to uncover missing functionality of these tools, enable meaningful comparisons between them and find recommendations for future improvements and research.

The contribution in this chapter is twofold. First, we provide an extensive list of features offered by general-purpose experiment management tools dedicated to distributed systems research on real platforms. We then use it to assess existing solutions and compare them, outlining possible future paths for improvements.

Considering the complexity of experimenting with distributed systems, there exist a plethora of specialized tools that address specific parts of the experimentation process. We conclude our study of general-purpose experiment management tools with a presentation of the state of the art of those complementary tools that are a valuable help for researchers when experimenting with distributed systems. The contents of this chapter were published in a paper [21] that I co-authored with Tomasz Buchert Ph.D student in the *AlGorille* team, at *LORIA* (Nancy).

2.1 Introduction

Distributed systems are among the most complex objects ever built by humans, as they are composed of thousands of systems that collaborate together. They also have a central role in today's society, supporting many scientific advances (scientific & high-performance computing, simulation, Big Data, etc.), and serving as the basis for the infrastructure of popular services such as Google or Facebook. Their role and popularity makes them the target of numerous research studies in areas such as scheduling, cost evaluation, fault tolerance, trust, scalability, energy consumption, etc.

Given the size and complexity of distributed systems, it is often unfeasible to carry out analytic studies, and researchers generally use an empirical approach relying on experimentation: despite being built by humans, distributed systems are studied as if they were natural objects, with methods similar to those used in biology or physics.

One can distinguish four main methodologies for experimentation on distributed systems [59]:

- *in-situ*: a real application is tested on a real platform.

- *simulation*: a model of an application is tested on a model of the platform.
- *emulation*: a real application is tested using a model of the platform.
- *benchmarking*: a model of an application is used to evaluate a real platform.

Each methodology has its advantages and disadvantages. For example, results obtained during simulation are (usually) completely reproducible. On the other hand, as the platform is a model of the reality, the results may not apply in a general sense, as the model could lack some unnoticed but important features. It is important to remark as well that all those methodologies complement each other and choosing between them depends on the level of realism we want to achieved in our experiments. In this chapter we focus on experiments based on *in-situ* and *emulation* methodologies.

Because of the actual size of the available testbeds and of the complexity of the different software layers, a lot of time is required to set up and perform experiments. Scientists are confronted with low-level tasks that they are not familiar with, making the validation of current and next generation of distributed systems a complex task. In order to lower the burden in setting up an experiment, different testbeds and experiment management tools have appeared. The last decade has seen more interest in the latter, mainly influenced by the needs of particular testbeds and other problems found in the process of experimentation such as reproducibility, replicability, automation, ease of execution, scalability, etc. Additionally, the existing number of papers oriented toward such tools asks for a classification in order to uncover their capabilities and limitations. Hence, *experiment management tools* are the main object of study in this chapter. We propose a set of features that improve the experimentation process in various ways at each step (design, deployment, running the main experiment and related activities, and data and result management). This list can be used to carry out a fair comparison of tools used for conducting experiments, as well as a guideline when choosing a tool that suits certain needs.

The rest of chapter is structured as follows. In Section 2.2 existing methods and approaches to experimentation with distributed systems are presented. Then, in Section 2.3, a set of features offered by existing experimentation tools is constructed and each element is carefully and precisely explained. In Section 2.4, we present a list of tools helping with research in distributed systems. Each tool is shortly presented and its features explained. Our additional observations and ideas are presented in Section 2.5. Finally, in Section 2.8 we conclude our work and discuss future work.

2.2 Context and terminology

This section introduces some definitions that will be used throughout this chapter, as well as the context where our object of study plays its role.

2.2.1 Definitions

For our purposes, an *experiment* is a set of actions carried out to test (confirm, falsify) a particular hypothesis. There are three elements involved in the process: a *laboratory* (the place where one experiments), an *investigator* (the one who experiments) and an *apparatus* (the object used to measure). If an experiment can be run with a different laboratory, investigator and apparatus, and still produce the same conclusions, one says that it is *reproducible*. This is in contrast with *replicability* which requires the same results while keeping these three elements unchanged. The terms *reproducibility* and *replicability* (*replayability*) produce a lot of confusion and discrepancies as they are often used to describe different ideas and goals. The above definitions are compatible with the definitions given in [44], although we do not share such a negative view about replicability as the authors. Being a "poor cousin" of reproducibility, replicability is nevertheless essential to the verification of results and code reusability as expressed in [37].

Finally, let us introduce a last piece of terminology and define the object of study in this chapter. An *experimentation tool* or an *experiment management tool* (for research in distributed

systems) is a piece of software that helps with the following main steps during the process of experimenting:

- design – by ensuring reproducibility or replicability, providing unambiguous description of an experiment, and making the experiment more comprehensible,
- deployment – by giving efficient ways to distribute files (e.g., scripts, binaries, source code, input data, operating system images, etc.), automating the process of installation and configuration, ensuring that everything needed to run the experiment is where it has to be,
- running the experiment itself – by giving an efficient way to control and interact with the nodes, monitoring the infrastructure and the experiment and signaling problems (e.g., failure of nodes),
- collection of results – by providing means to get and store results of the experiment.

Furthermore, it addresses experimentation in its full sense and it is normally conceived with one of the following purposes described fully in the next section:

- ease of experimenting,
- replicability,
- reproduciblity,
- controlling and exploring parameter space.

In this study we narrow the object of study even more by considering only *general-purpose* experiment management tools (i.e., tools that can express arbitrary experimental processes) and only ones that experiment with real applications (i.e., *in-situ* and *emulation* methodologies). The former restriction excludes many tools with predefined experimental workflows whereas the latter excludes, among others, simulators (see Section2.6).

2.2.2 Motivations for experimentation tools

As described before, there exist many tools that strive to ease experimentation with distributed systems. These tools are the main object of study in this article and as such they are described thoroughly in Section 2.4. Here, however, we discuss the main driving forces that are behind the emergence of experimentation tools.

Ease of experimenting

The first motivation, and the main one, for creating experimentation tools is helping with the scientific process of experimenting and making the experimenter more productive. By providing well designed tools that abstract and outsource tedious yet already solved tasks, the development cycle can be shortened, while becoming more rigorous and targeted. Moreover, it may become more productive as the scientist may obtain additional insights and feedback that would not be available otherwise. The ease of experimenting can indirectly help to solve the problem of research of questionable quality in the following sense. As the scientific community exerts pressure on scientists to publish more and more, they are often forced to publish results of dubious quality. If they can forget about time-consuming, low-level details of an experiment and focus on the scientific question to answer, hopefully they could spend more time testing and strengthening their results.

Replicability (automation)

Replicability which is also known as replayability deals with the act of repeating a given experiment under the very same conditions. In our context it means: same software, same external factors (e.g., workload, faults, etc.), same configuration, etc. If done correctly, it will lead to the same results as obtained before, allowing others to build on previous results and to carry out fair comparisons. There are several factors that hamper this goal: size of the experiment, heterogeneity and faulty behavior of testbeds, complexity of the software stack, numerous details of the configuration, generation of repeatable conditions, etc. Among other goals, experimentation tools try to control the experiment and produce the same results under the same conditions, despite the aforementioned factors.

Reproducibility

It refers to the process of independent replication of a given experiment by another experimenter. Achieving reproducibility is much harder than replicability because we have to deal with the measurement bias that can appear even with the slightest change in the environment. Therefore, in order to enhance the reproducibility of an experiment, the following features are required:

- automatic capture of the context (i.e., environment variables, command line parameters, versions of software used, software dependencies, etc.) in which the experiment is executed;
- detailed description of all the steps that led to a particular result.

The description of an experiment has to be independent of the infrastructure used. To do so abstractions for the platform have to be offered.

Controlling and exploring the parameter space

Each experiment is run under a particular set of conditions (parameters) that precisely define its environment. The better these conditions are described, the fuller is understanding of the experiment and obtained results. Moreover, a scientist may want to explore the *parameter space* in an efficient and adaptive manner instead of doing it exhaustively.

Typical parameters contained in a parameter space for a distributed system experiment are:

- number of nodes,
- network topology,
- hardware configuration (CPU frequency, network bandwidth, disk, etc.),
- workload during the experiment.

One can enlarge the set of parameters tested (e.g., considering CPU speed in a CPU-unaware experiment) as well as vary parameters in their allowed range (e.g., testing a network protocol under different topologies).

Whereas the capability to control the various experimental parameters can be, and quite often is, provided by an external tool or a testbed (e.g., Emulab), the high-level features helping with a design of experiments (DoE), as the efficient parameter space exploration, belong to experimentation tools.

Scalability

Another motivation for an experiment control is scalability of experiments, that is, being able to increase their size without harming some practical properties and scalability metrics. For example, one can consider if an experimentation tool is able to control many nodes (say, thousands) without significantly increasing the time to run the experiment, or without hampering the statistical significance of results.

The most important properties concerning scalability are:

- time – additional time needed to control the experiment (over the time to run it itself),
- resources – amount of resources required to control the experiment,
- cost of the experiment – funds required to run the experiment and control it (cf. commercial cloud computing),
- quality of results – the scientific accuracy of the results, their reproducibility in particular (contrary to the above properties, this one is hard to define and measure).

These metrics are functions of experiment parameters (see Section2.7.5) and implementation details. Among important factors that limit scalability understood as the metrics above are:

- number of nodes used in the experiment,
- size of monitoring infrastructure,
- efficiency of data management.

2.2.3 Testbeds

Testbeds play an important role in the design and validation of distributed systems. They offer controlled environments that are normally shielded from the randomness of production environments. Here, we present a non-exhaustive list of testbeds that motivated the development of experiment management tools. There exists a work on defining useful features of network testbeds, similar to the goals of our study [118]. Unsurprisingly, some features overlap in both analyses.

- Grid'5000 [25] is an experimental testbed dedicated to the study of large-scale parallel and distributed systems. It is a highly configurable experimental platform with some unique features. For example, a customized operating system (e.g., with a modified kernel) can be installed and full "root" rights are available. The platform offers a REST API to control reservations, but does not provide dedicated tools to control experiments. However, the nodes can be monitored during the experiment using a simple API.
- Emulab [130] is a network testbed that allows one to specify an arbitrary network topology (thanks to the emulation of the network). This feature ensures a predictable and repeatable environment for experiments. User has access to a "root" account on the nodes, but cannot tweak the internals of the operating system. Emulab comes with a dedicated tool to control experiments (see 2.4.3).
- PlanetLab [103] is a globally distributed platform for developing, deploying and accessing planetary-scale network services. It consists of geographically distributed nodes running a light, virtualized environment. The nodes are connected over the Internet. PlanetLab offers Plush (see 2.4.4) for the experiment control.
- ORBIT [108, 98] is a radio grid testbed for scalable and reproducible evaluation of next-generation wireless network protocols. It offers a novel approach involving a large grid of radio nodes which can be dynamically interconnected into arbitrary topologies with reproducible wireless channel models. A dedicated tool to run experiments with ORBIT platform is OMF (see 2.4.6).
- DAS[1] (Distributed ASCI Supercomputer) is a Dutch wide-area distributed system designed by the Advanced School for Computing and Imaging (ASCI). Distinguishably, it employs various HPC accelerators (e.g., GPUs) and novel network interconnect. Its most recent iteration is DAS-4. DAS does not offer a dedicated tool to control experiments, however it provides a number of tools to help with deployment, discovering problems and scheduling.

[1] `http://www.cs.vu.nl/das4/`

With the emergence of efficient and cheap virtualization, the scientists turn to *cloud computing infrastructures* as a viable experimentation platform. A popular commercial service is Amazon EC2[2], but many alternatives and variations exist (e.g., Windows Azure[3]). There are non-commercial, open-source solutions available as well (e.g., OpenStack[4]). Even though the development of cloud computing solutions was not inspired by a need of a research platform, the scalability and elasticity offered by those make it an attractive solution for science. In [84] a framework oriented toward reproducible research on such infrastructures is proposed.

2.3 List of features offered by experiment management tools

In this section, we present properties available in experiment management tools for distributed systems after doing a literature review using the following sources:

- tools used and published by the most important and large-scale testbeds (see Section 2.2.3),
- papers referenced by these tools and papers that cite them,
- IEEE and ACM digital libraries search with the following keywords in the abstract or title: *experiments*, *experiment*, *distributed systems*, *experimentation*, *reproducible*.

We ended up with 8 relevant tools for managing experiments that met our criteria of an *experimentation tool*, however we also include *Naive approach* (see Section 2.4.1) in our analysis. An extensive analysis of the papers dedicated to those tools was performed; subsequently, a set of properties and features - highlighted by each of the tools as to be important for the experimentation process - was selected and classified.

The list consists of nine groups of properties and features that have an important role in the experimentation process. The complete hierarchy is presented in Figure 2.1.

2.3.1 Description Language

The design of the experiment is the very first step in the experimentation process. The description language helps users with this step, allowing them to describe how the experiment has to be performed, as well as their needs for running the experiment. Characteristics that help with describing the experiment are presented in the following sections.

Representation (Imperative / Declarative / Workflow / Scripts) of experiments featured by a given tool is the approach used to describe the experiment and relevant details. Possible representations differ in their underlying paradigm (e.g., imperative, declarative) and in a level of abstraction that the description operates on. Some tools use low-level scripts to build experiments whereas others turn to higher abstractions, some of them graphical (e.g., workflows). The choice of a certain representation has implications on other aspects of the description language.

Modularity (Yes / No) is a property of experiment description language that enables easy adding, removing, replacing and reusing parts of experiments. An experiment expressed in a modular way can be logically split into modules with well-defined interfaces that can be worked on independently, possibly by different researchers specializing in a particular aspect of the experiment.

Expressiveness (Yes / No) that makes it effective in conveying thoughts and ideas, in short and succinct form. Expressiveness provides a more maintainable, clearer description. Various elements can improve expressiveness: well-chosen abstractions and constructions, high-level structure, among others.

[2] `http://aws.amazon.com/ec2/`
[3] `http://www.windowsazure.com/`
[4] `http://www.openstack.org/`

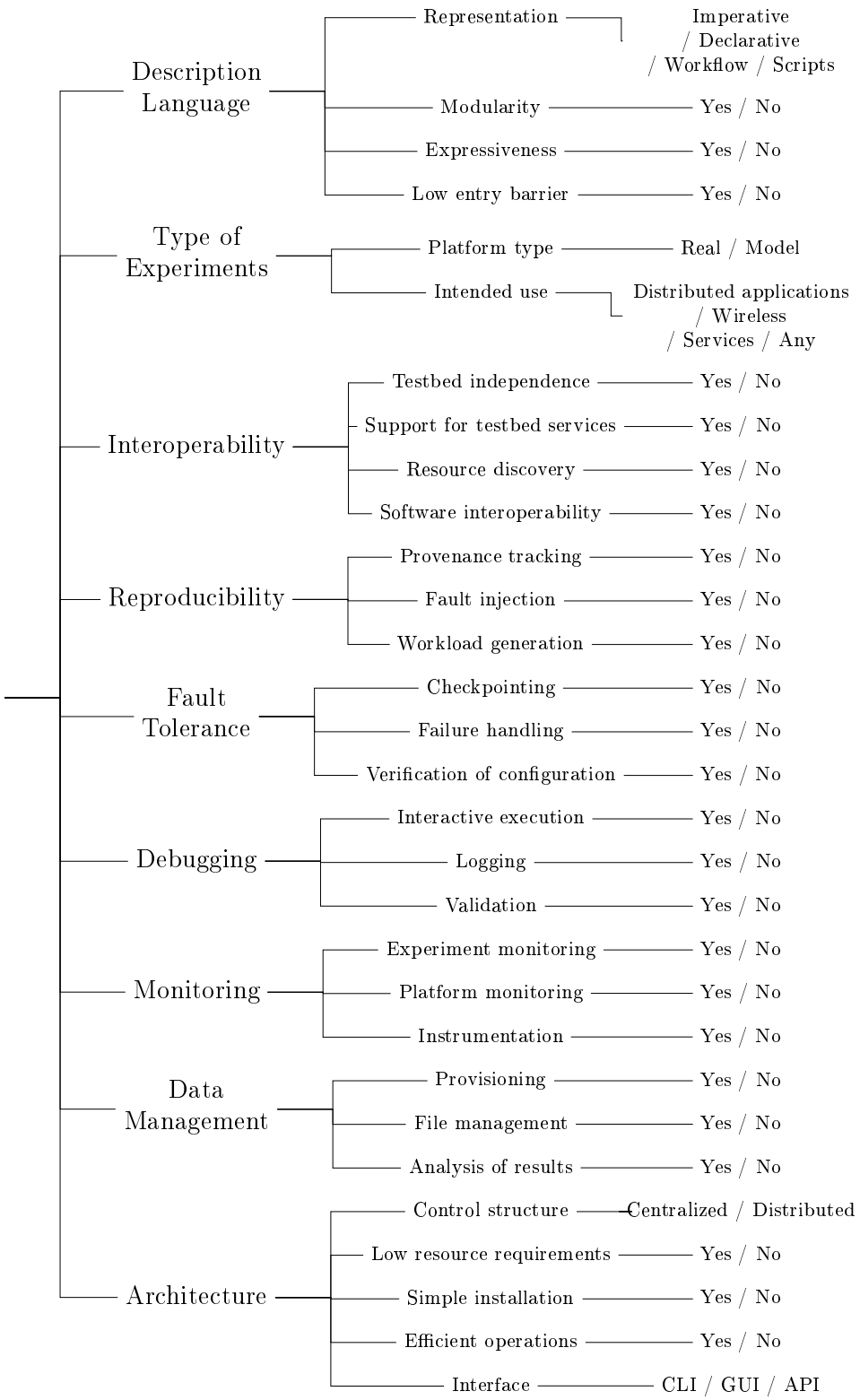

Figure 2.1: The tree of features. All evaluated properties and features are presented with their respective domains of values. The properties are grouped into 9 groups that cover different aspects of experiment management.

Low entry barrier (Yes / No) is the volume of work needed to switch from naive approach to the given approach while assuming prior knowledge about the infrastructure and the experiment itself. In other words, it is the time required to learn how to efficiently design experiments in the language of the given experimentation tool.

2.3.2 Type of Experiments

This encompasses two important aspects of an experiment: the platform where the experiments are going to run on and the research fields where those experiments are performed.

Platform type (Real / Model) is the range of platforms supported by the experimentation tool. The platform type can be *real* (i.e., consist of physical nodes) or be a *model* (i.e., built from simplified components that model details of the platform like network topology, links bandwidth, CPU speed, etc.). For example, platforms using advanced virtualization or emulation techniques (like Emulab testbed) are considered to be modeled. Some testbeds (e.g., PlanetLab) are considered real because they do not hide the complexity of the platform, despite the fact that they use virtualization.

Intended use (Distributed applications / Wireless / Services / Any) refers to the research context the experimentation tool targets. Examples of research domains that some tools specialize in include: wireless networks, network services, high performance computing, peer-to-peer networks, among many others.

2.3.3 Interoperability

It is important for an experimentation tool to interact with different platforms, as well as to exploit their full potential. The interaction with external software is an indisputable help during the process of experimenting.

Testbed independence (Yes / No) of the experimentation tool is its ability to be used with different platforms. The existing tools are often developed along with a single testbed and tend to focus on its functionality and, therefore, cannot be easily used somewhere else. Other tools explicitly target a general use and can be used with a wide range of experimental infrastructures.

Support for testbed services (Yes / No) is a capability of the tool to interface different services provided by the testbed where it is used (e.g., resource requesting, monitoring, deployment, emulation, virtualization, etc.). Such a support may be vital to perform scalable operations efficiently, exploit advanced features of the platform or to collect data unavailable otherwise.

Resource discovery (Yes / No) is a feature that allows to reserve a set of testbed resources meeting defined criteria (e.g., nodes with 8 cores interconnected with 1 Gbit network). Among methods to achieve this feature are: interoperating with testbed resource discovery services or emulation of resources by the tool.

Software interoperability (Yes / No) is the ability of using various types of external software in the process of experimenting. The experimentation tool that interoperates with software should offer interfaces or means to access or integrate monitoring tools, commands executers, software installers, package managers, etc.

2.3.4 Reproducibility

This group concerns all methods used to help with reproducibility and repeatability as was described in Section 2.2.2.

Provenance tracking (Yes / No) is defined as a way of tracing and storing information of how scientific results have been obtained. An experimentation tool supports data provenance if it can describe the history of a given result for a particular experiment. An experimentation tool can provide data provenance through the tracking of details at different layers of the experiment. At a low-level layer, the tool must be able to track details such as: command-line parameters, process arguments, environment variables, version of binaries, libraries and kernel modules in use, hardware devices used, filesystem operations executed, etc. At a high-level layer, it must track details such as: number of nodes used, details of used machines, timestamps of operations, state of the platform, etc.

Fault injection (Yes / No) is a feature that enables the experimenter to introduce factors that can modify and disrupt the functioning of the systems being studied. These factors include: node failures, link failures, memory corruption, background CPU load, etc. This feature allows to run experiments under more realistic and challenging conditions and test behavior of the studied system under exceptional situations.

Workload generation (Yes / No) is a range of features that allow to inject a predefined workload into the experimental environment (e.g., number of requests to a service). The generated workload is provided by real traces or by synthetic specification. Similarly to fault injection, this feature allows to run experiments in more realistic scenarios.

2.3.5 Fault Tolerance

This group of features encompasses all of them that help with common problems that can happen during experiments and may lead to either invalid results (especially dangerous if gone unnoticed) or to increased time required to manually cope with them.

Checkpointing (Yes / No) allows to save a state of the experiment and to restore it later as if nothing happened. It is a feature that can, above all, save the time of the user. There are at least two meanings of checkpointing in our context:

- only some parts of the experiment are saved or cached,
- the full state of the experiment is saved (including the platform).

Of course, the second type of checkpointing is much more difficult to provide. Checkpointing helps with fault tolerance as well, since a failed experiment run will not necessarily invalidate the whole experiment.

Failure handling (Yes / No) of the experimentation tool can mitigate runtime problems with the infrastructure an experiment is running on. This means in particular that failures are detected and appropriate steps are taken - restarting the experiment, for example. Typical failures are crashing nodes, network problems, etc.

Verification of configuration (Yes / No) consists in having an automatic way to verify the state of an experimentation platform. Usually such a step is performed before the main experiment to ensure that properties of the platform agree with a specification. We distinguish verification of:

- software – ensuring that the software is coherent on all computing nodes,
- hardware – ensuring that the hardware configuration is as it is supposed to be.

2.3.6 Debugging

The features grouped in this section help to find problems and their causes during the experimentation process.

Interactive execution (Yes / No) refers to an ability to run the experiment "on-the-fly" including: manually scheduling parts of the experiment, introspecting its state and observing intermediate results. This feature is inspired by debuggers offered by integrated development environments (IDEs) for programming languages.

Logging (Yes / No) consists of features that allow bookkeeping of low-level messages emitted during experiments including those that were placed at arbitrary places by the experimenter. The messages are normally stored sequentially along with their timestamps making the log is essentially a one-dimensional dataset. The log can be used to debug an experiment and document its execution.

Validation (Yes / No) is a feature that offers the user a way to perform a fast (that is, faster than full execution of the experiment) and automatic way to verify the description of an experiment. Depending on the modeling language used and other details, the validation may be accordingly thorough and complete. For our purposes, we require that at least some semantic analysis must be performed, in contrast to simple syntactic analysis.

2.3.7 Monitoring

Monitoring is necessary to understand the behavior of the platform and the experiment itself. It consists in gathering data from various sources: the experiment execution information, the platform parameters and metrics, and other strategic places like instrumented software.

Experiment monitoring (Yes / No) consists in observing the progress of the experiment understood as set of timing and causal information between actions in the experiment. The monitoring includes keeping track of currently running parts of the experiment as well as their interrelations. Depending on the model used, this feature may take different forms.

Platform monitoring (Yes / No) is the capability of an experimentation tool to know the state of resources that comprise the experiment (nodes, network links, etc.). Data collected that way may be used as a result of the experiment, to detect problems with the execution or as a way to get additional insights about the experiment.

Instrumentation (Yes / No) enables the user to take measurements at different moments and places while executing the experiment. This includes instrumentation of software in order to collect measures about its behavior (CPU usage, performance, resource consumption, etc.).

2.3.8 Data Management

The management of data is an important part of the experiment. This section contains features that help with distribution and collection of data.

Provisioning (Yes / No) is the set of actions to prepare a specific physical resource with the correct software and data, and make it ready for the experimentation. Provisioning involves tasks such as: loading of appropiate software (e.g., operating system, middleware, applications), configuration of the system and starting necessary services. It is necessary for any experimentation tool to provide at least a rudimentary form of this functionality.

File management (Yes / No) is a feature that abstracts a tedious job of working with files. Therefore the user does not have to manage them manually at a low level which often is error-prone. This includes actions like automatic collection of results stored at participating nodes.

Analysis of results (Yes / No) is a service of an experimentation tool that is used to collect, store and visualize experimental results, as well as making dynamic decisions based on their runtime values. The latter ability paves a way into intelligent design of experiments by exploring only relevant regions of parameter space and therefore saving resources like energy or time.

2.3.9 Architecture

This section contains features and properties related to how the tool is designed and what architecture decisions the authors made. This includes ways to interact with the tool, as well as technical details such as software dependencies, methods to achieve scalability and efficient execution of experiments.

Control structure (Centralized / Distributed) refers to the structure of nodes used to control the experiment. The architecture of a tool is *centralized* if the control of an experiment is centralized and there exists one node that performs all principal work. Otherwise, if there are multiple nodes involved in the experiment control, then the architecture is *distributed*.

Low resource requirements (Yes / No) of an experimentation tool refer to its resource consumption (memory, CPU, network bandwidth, etc.) associated with the activity of controlling the experiment. As the number of elements the experiment consists of increases (e.g., nodes), so does the amount of the resources necessary to control them.

Simple installation (Yes / No) is understood as a low difficulty of setting up a completely functional infrastructure that the tool needs in order to be used. This usually implies software dependencies (interpreters, libraries, special services, etc.) or a required hardware infrastructure (number of network interfaces, minimum memory size, number of dedicated nodes to control the experiment, etc.)

Efficient operations (Yes / No) is the range of features that provide methods, tools and algorithms to perform large-scale operations with the experimental infrastructure. This in particular includes: efficient and scalable methods for command execution, file distribution, monitoring of nodes, gathering of results, among others. Providing efficient versions of these actions is notably difficult as operations involving nodes in a distributed systems are non-trivially scalable as a number of nodes increases.

Interface (CLI / GUI / API) consists of different ways that the user can interact with the experimentation tool. Most of the tools provide command line interface, whereas some tools provide graphical interfaces, usually via webpage used to interact with the experiment.

2.4 Existing experimentation tools

The aim of this section is to present the state of the art of the existing tools for experimentation with distributed systems. We focus our attention on the tools that fulfill the criteria for being considered as an experimentation tool (for a list of tools that are not included in the analysis, see Section 2.6). The evaluation of all tools and the main result of our study is presented in Table 2.1 that shows a comparison of the tools based on the proposed list of features. Figure 2.2 shows a timeline of publications about these experiment management tools and the impact of these tools measured as the number of citations is shown in Table 2.2.

2.4.1 Naive method

Frequently, experiments are done using this method which includes manual procedures and use of hand-written and low-level scripts. Lack of modularity and expressiveness is commonly seen because of the *ad hoc* nature of these scripts, and it is even worse when the experiment involves many machines. The experiment is controlled at a very low level, including some human intervention. Therefore, interaction with many types of applications and platforms is possible at the cost of time required to do so. Parameters for running the experiment can be forgotten as well as the reason for which they were used. This leads to an experiment that is difficult to understand and repeat. Since the experiment is run in partially manual fashion, the user can react against some unexpected behaviors seen during the experiment.

		Naive approach	Weevil	Workbench	Plush/Gush	Expo	OMF	NEPI	XPFlow	Execo
Description Language (18/27 ≈ 67%)	Representation	Scripts	Declarative[12]	Imperative[13]	Declarative[14]	Imperative[15]	Imperative[16]	Imperative[17]	Declarative[18]	Imperative[19]
	Modularity (4/9)	No	Yes	No	No	No	No	Yes	Yes	Yes
	Expressiveness (7/9)	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
	Low entry barrier (7/9)	Yes	No	Yes	Yes[20]	Yes	Yes	Yes	No	Yes
Type of Experiments	Platform type	Real	Real	Model	Real	Real	Real	Real, Model	Real	Real
	Intended use	Any	Services	Any	Any	Any	Wireless[21]	Any	Any	Any
Interoperability (22/36 ≈ 61%)	Testbed independence (8/9)	Yes	Yes	No	Yes[22]	Yes	Yes	Yes	Yes	Yes
	Support for testbed services (7/9)	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Resource discovery (5/9)	No	No	Yes*	Yes	Yes*	Yes	Yes	No	No
	Software interoperability (2/9)	No	No	No	Yes	No	Yes	No	No	No
Reproducibility (4/27 ≈ 15%)	Provenance tracking (1/9)	No	No	Yes	No	No	No	No	No	No
	Fault injection (2/9)	No	Yes	No	No	No	Yes*	No	No	No
	Workload generation (1/9)	No	Yes	No	No	No	No	No	No	No
Fault Tolerance (12/27 ≈ 44%)	Checkpointing (4/9)	No	Yes	No	No	No	No	Yes	Yes	Yes
	Failure handling (6/9)	No	Yes	No	Yes	No	Yes	Yes	Yes	Yes
	Verification of configuration (2/9)	No	No	Yes*	No	No	Yes	No	No	No
Debugging (17/27 ≈ 63%)	Interactive execution (7/9)	Yes	No	Yes	Yes	Yes	Yes	Yes	No	Yes
	Logging (6/9)	No	No	Yes*	No	Yes	Yes	Yes	Yes	Yes
	Validation (4/9)	No	Yes	Yes	No	No	No	Yes	Yes	No
Monitoring (10/27 ≈ 37%)	Experiment monitoring (4/9)	No	No	Yes	No	No	Yes	Yes	Yes	No
	Platform monitoring (4/9)	No	No	Yes*	Yes	No	Yes	Yes	No	No
	Instrumentation (2/9)	No	No	No	Yes	No	Yes	No	No	No
Data Management (13/27 ≈ 48%)	Provisioning (5/9)	No	Yes	Yes*	Yes	No	Yes	Yes	No	No
	File management (5/9)	No	Yes	Yes	Yes	No	Yes	No	No	Yes
	Analysis of results (3/9)	No	No	Yes	No	No	Yes	No	Yes	No
Architecture (19/27 ≈ 70%)	Control structure	Centralized	Centralized	Centralized	Centralized	Centralized	Distributed	Distributed	Centralized	Centralized
	Low resource requirements (6/9)	Yes	Yes	No	No	Yes	No	Yes	Yes	Yes
	Simple installation (7/9)	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes
	Efficient operations (6/9)	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes
	Interface	CLI	CLI	GUI, CLI, API	CLI, GUI, API	CLI	CLI, GUI	CLI, GUI	CLI	CLI

[1] GNU m4
[2] Event-based (Tcl & ns)
[3] XML
[4] Ruby
[5] Event-based (Ruby)
[6] Modular API based on Python
[7] Workflows (Ruby)
[8] Modular API based on Python
[9] Using GUI
[10] Supports wired resources as well
[11] PlanetLab oriented
[12] GNU m4
[13] Event-based (Tcl & ns)
[14] XML
[15] Ruby
[16] Event-based (Ruby)
[17] Modular API based on Python
[18] Workflows (Ruby)
[19] Modular API based on Python *Provided by testbed
[20] Using GUI
[21] Supports wired resources as well
[22] PlanetLab oriented

Table 2.1: Summary of analyzed experiment management tools for distributed systems research. Each feature is presented along with a number of tools that provide it. Similarly, for each group a percentage of implemented features from this group is shown. Features that are due to the integration with a testbed are marked with ⋆.

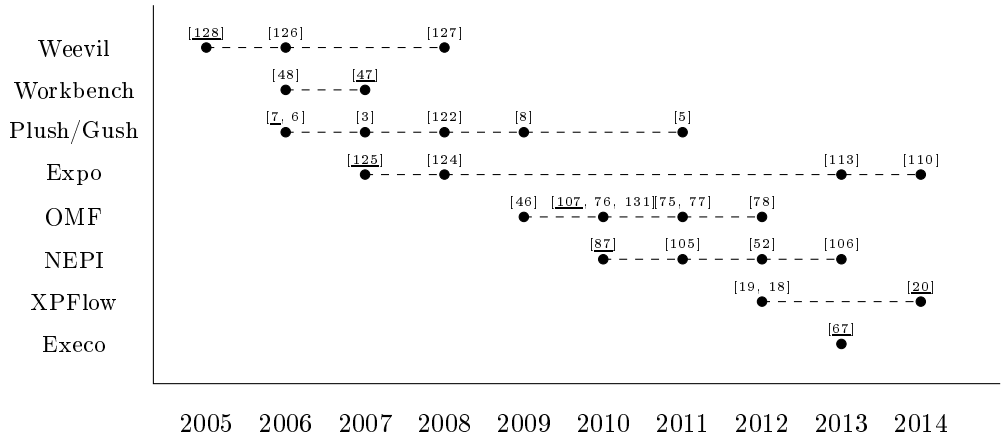

Figure 2.2: Timeline of publications dedicated to experiment management tools. The publication that attracted most of the citations (main publication) is underlined.

2.4.2 Weevil

It is a tool to evaluate distributed systems under real conditions, providing techniques to automate the experimentation activity. This experimentation activity is considered as the last stage of development. Experiments are described declaratively with a language that is used to instantiate various models and provides clarity and expressiveness. Workload generation is one of its main features, which helps with the replicability of results.

2.4.3 Workbench for Emulab

Workbench is an integrated experiment management system, which is motivated by the lack of replayable research on the current testbed-based experiments. Experiments are described using an extended version of the *ns* language which is provided by Emulab. The description encompasses static definitions (e.g., network topology, configuration of devices, operating system and software, etc.) and dynamic definitions of activities that are based on *program agents*, entities that run programs as part of the experiment. Moreover, activities can be scheduled or can be triggered by defined events. Workbench provides a generic and parametric way of instantiating an experiment using features already provided by Emulab to manage experiments. This allows experimenters to run different instances of the same experiment with different parameters. All pieces of information necessary to run the experiment (e.g., software, experiment description, inputs, outputs, etc.) are bundled together in *templates*.

Templates are both persistent and versioned, allowing experimenters to move through the history of the experiment and make comparisons. Therefore, the mentioned features facilitate the replay of experiments, reducing the burden on the user. Data management is provided by the underlying infrastructure of Emulab, enabling Workbench to automatically collect logs that were generated during the experiment.

2.4.4 Plush/Gush

Plush, and its another incarnation called Gush, cope with the deployment, maintenance and failure management of different kinds of applications or services running on PlanetLab. The description of the application or services to be controlled is done using *XML*. This description comprehends the acquisition of resources, software to be installed on the nodes and the workflow of the execution. It has a lightweight client-server architecture with a few dependencies that can be easily deployed

on a mix of normal clusters and GENI control frameworks: PlanetLab, ORCA[5] and ProtoGENI[6]. One of the most important features of *Plush* is its capacity to manage failures. The server receives a constant stream of information from all the client machines involved in the experiment and performs corrective actions when a failure occurs.

2.4.5 Expo

Expo offers abstractions for describing experiments, enabling users to express complex scenarios. These abstractions can be mapped to the hierarchy of the platform or can interface underlying tools, providing efficient execution of experiments. Expo brings the following improvements to the experimentation activity: it makes the description of the experiment easier and more readable, automates the experimentation process, and manages experiments on a large set of nodes.

2.4.6 OMF

It is a framework used in different wireless testbeds around the world and also in PlanetLab. Its architecture versatility aims at federation of testbeds. It was mainly conceived for testing network protocols and algorithms in wireless infrastructures. The OMF architecture consist of 3 logical planes: Control, Measurement, and Management. Those planes provide users with tools to develop, orchestrate, instrument and collect results as well as tools to interact with the testbed services. For describing the experiment, it uses a comprehensive domain specific language based on Ruby to provide experiment-specific commands and statements.

2.4.7 NEPI

NEPI is a Python library that enables one to run experiments for testing distributed applications on different testbeds (e.g., PlanetLab, OMF wireless testbeds, network simulator, etc). It provides a simple way for managing the whole experiment life cycle (i.e., deployment, control and results collection). One important feature of NEPI is that it enables to use resources from different platforms at the same time in a single experiment. NEPI abstracts applications and computational equipment as resources that can be connected, interrogated and conditions can be registered in order to specify workflow dependencies between them.

2.4.8 XPFlow

XPFlow is an experimentation tool that employs *business workflows* in order to model and run experiments as *control flows*. XPFlow serves as a workflow engine that uses a domain-specific language to build complex *processes* (experiments) from smaller, independent tasks called *activities*. This representation is claimed to bring useful features of Business Process Modeling (BPM), that is: easier understanding of the process, expressiveness, modularity, built-in monitoring of the experiment, and reliability.

Both XPFlow and scientific workflow systems rely on workflows. However, scientific workflows are data-oriented and the distributed system underneath (e.g., a computational grid) is merely a tool to efficiently process data, not an object of a study. Moreover, the formalism of XPFlow is inspired by workflow patterns identified in the domain of BPM, which are used to model *control flows*, as opposed to *data flows* (see Section 2.6.2).

2.4.9 Execo

Execo is a generic toolkit for scripting, conducting and controlling large-scale experiments in any computing platform. Execo provides different abstractions for managing local and remote processes as well as files. The engine provides functionality to track the experiment execution and

[5] `http://groups.geni.net/geni/wiki/ORCABEN`
[6] `http://www.protogeni.net`

Tool	First publication	Citations
Weevil	2005	69
Workbench	2006	80
Plush/Gush	2006	177
Expo	2007	16
OMF	2009	152
NEPI	2010	38
XPFlow	2012	3
Execo	2013	1

Table 2.2: Number of publications citing papers dedicated to each experimentation tool (as verified on 4 July 2014).

offers features such as *parameter sweep* over a defined set of values. The partial results of the parameter sweep can be saved to persistent storage, therefore avoiding unnecessary reruns in case of a failure.

2.5 Discussion

Existing tools for experiment control were analyzed and evaluated using our set of features defined in Section 2.3 and the final results are presented in Table 2.1. For each position in the table (i.e., each property/tool pair) we sought for an evidence to support possible values of a given property in a given tool from a perspective of a prospective user. To this end, the publications, documentation, tutorials and other on-line resources related to the given approach were consulted. If presence of the property (or lack thereof) could be clearly shown from these observations, the final value in the table reflects this fact. However, if we could not find any mention of the feature, then the final value claims that the feature does not exist in the tool, as for all practical purposes the prospective user would not be aware of this feature, even if it existed. In ambiguous cases additional comments were provided. Much more detailed analysis that led to this concise summary is available on-line[7]. Using information collected in the table, one can easily draw few conclusions.

There is no agreement whether a declarative description is more beneficial than an imperative one. Declarative descriptions seem to be associated with higher modularity and expressiveness, but at a price of a higher entry barrier. Moreover, the tools tend to be independent of a particular testbed, but those with tight integration offer a more complete set of features or features not present in other solutions (e.g., Emulab Workbench).

The majority of addressed features come from *Architecture* (70%), *Description Language* (67%), *Debugging* (63%) and *Interoperability* (61%) groups.

On the other hand, support for *Fault Tolerance* and *Monitoring* is quite low (44% and 37%, respectively), whereas support for *Reproducibility* is almost nonexistent (only 15%).

The features available in majority of the analyzed tools are: *Testbed independence* (8/9), *Expressiveness* (7/9), *Low entry barrier* (7/9), *Support for testbed services* (7/9), *Interactive execution* (7/9), *Failure handling* (6/9), *Logging* (6/9), *Resource discovery* (5/9), *File management* (5/9) and *Provisioning* (5/9). Moreover, the tools have nearly universally *Simple installation* (7/9), *Low resource requirements* (6/9) and offer methods to perform *Efficient operations* (6/9).

The two most unimplemented features are *Provenance tracking* (1/9) and *Workload generation* (1/9), both crucial for reproducibility of experiments.

[7] `http://www.loria.fr/~buchert/exp-survey.yaml`

Additionally, some tools offer unique features: *Software interoperability* (Plush and OMF), *Provenance tracking* (Workbench), *Fault injection* (Weevil and OMF), *Workload generation* (Weevil), *Verification of configuration* (Workbench and OMF) and *Instrumentation* (Plush and OMF). However, it is worth pointing out that features such as *Workload generation* are often provided by standalone tools.

Finally, we did a simple "impact analysis" of described tools by summing all unique scientific citations to papers about each tool using Google Scholar (see Table 2.2). Clearly, without adjusting the score to the age of each tool, the most cited tool is Plush. As interesting as these data may be, we abstain from drawing any more conclusions from them. The summary of this analysis is available on-line[8].

2.6 Tools not covered in the study

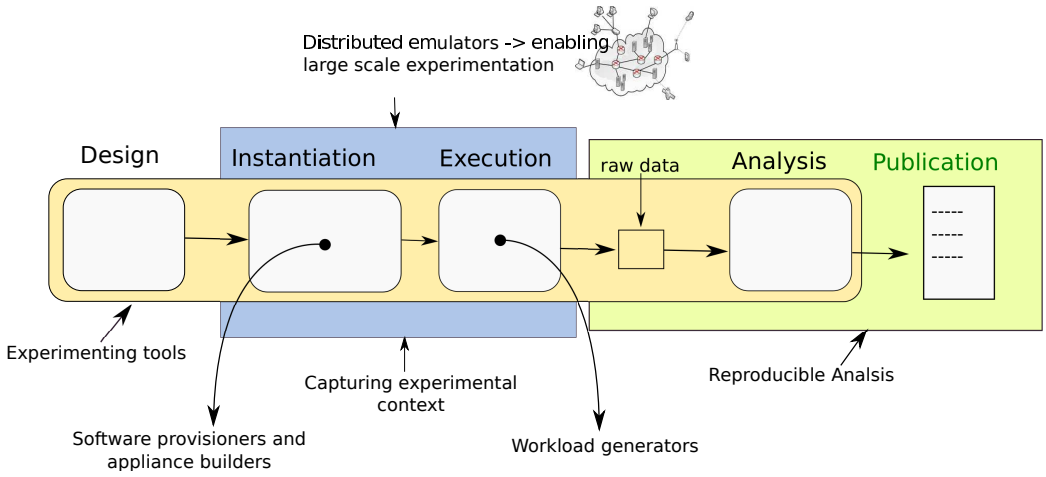

Figure 2.3: Whole panorama of tools that help with experimentation. Complementary tools are shown and their place in the experimental cycle. Those tools cover: distributed emulators, software provisioners, appliance builders, workload generators, tools for performing reproducible analysis and tools for capturing the experimental context.

In the following section, we discuss other tools that could be mistaken as an experiment management tool according to our definition. Those tools contradict our the definition (cf. Section 2.6.1) even though they support most of the experimental cycle with distributed systems.

2.6.1 Non general-purpose experiment management tools

Tools like ZENTURIO [104] and Nimrod [1] helps experimenters to manage the execution of parametric studies on cluster and Grid infrastructures. Both tools cover activities like the set up of the infrastructure to use, collection and analysis of results. ZENTURIO offers a more generic parametrization, making it suitable for studying parallel applications under different scenarios where different parameters can be changed (e.g., application input, number of nodes, type of network interconnection, etc.). Even though Nimrod parametrization is restricted to application input files, a relevant feature is the automation of the design of fractional factorial experiments. NXE [56] scripts the execution of several steps of the experimental workflow from the reservation of resources in a specific platform to the analysis of collected logs. The whole experiment scenario is described using XML which is composed of three parts: topology, configuration and scenario. All the interaction with resources and applications is wrapped using bash scripts. NXE is mainly dedicated to the evaluation of network protocols.

[8] `http://www.loria.fr/~buchert/exp-impact.yaml`

The aforementioned tools were not included in our analysis, because they are not *general-purpose* experiment management tools. They address only very specific scenarios of experimentation with a distributed system like parametric studies and network protocols evaluation.

2.6.2 Scientific workflow systems

The aim of scientific workflow systems is automation of the scientific process that a scientist may go through to get from raw data to publishable results. The main objective is to communicate analytical procedures repeatedly with minimal effort, enabling the collaboration on conducting large, data-processing, scientific experiments. Scientific workflows are designed specifically to compose and execute a series of computational or data manipulation steps. Normally, those systems are provided with GUIs that enable non-expert users to easily construct their applications as a visual graph. Goals such as data provenance and experiment repeatability are both shared by scientific workflows and experimentation tools. Some examples of scientific workflows are: Kepler [91], Taverna [65] and Vistrails [22]. An interesting analysis of these systems, and a motivation for this work, is presented in [132].

There are two main reasons why scientific workflows are not covered in our study. First, scientific workflows are *data flows* in nature – they are used to run complex computations on data, while the computational platform is abstracted and user has no direct control over it (e.g., the nodes used during computation). Hence the platform is not the object of study, but merely a tool to carry out computation. Second, the declarative representation of scientific workflows as acyclic graphs is generally limited in its expressiveness, therefore they do not meet the criteria of *general-purpose* experimentation tools according to our definition (see [39, 35] for analyses of scientific workflows expressiveness).

2.6.3 Simulators and abstract frameworks

An approach widely used for evaluating and experimenting with distributed systems is simulation. In [95] the most used simulators for overlay networks and peer-to-peer applications are presented. Another framework called SimGrid [27] is used for the evaluation of algorithms, heuristics and even real MPI applications in distributed systems such as Grid, Cloud or P2P systems.

Even though simulators provide many features required by the definition of the experimentation tool, they are not included in our study. First, they do not help with experiments on real platforms as they provide an abstract and modeled platform instead. Second, the goals of simulators are often very specific to a particular research subdomain and hence are not general-purpose tools [27].

Other tools such as Splay [89] and ProtoPeer [53] go one step further by making easy the transition between simulation and real deployment. Both tools provide a framework to write distributed applications based on a model of the target platform. They are equipped with measurement infrastructures and event injection for reproducing the dynamics of a live system.

The tools providing abstract framework to write applications under experimentation are not considered in our study, because real applications cannot be evaluated with them. Although real machines may be used to run experiments (as it is the case with Splay), the applications must be ported to APIs provided by these tools.

2.7 Complementary tools

In this section complementary tools are shown. Those tools address specific parts of the process of experimentation with distributed systems as can be seen in Figure 2.3. Experiment management tools can take advantage of these tools to implement features presented in Section 2.3.

2.7.1 Software provisioners and appliance builders

Puppet[9] and Chef[10] are commonly used in automating administrative tasks such as software provision and configuration of operating systems. They simplify complex deployments by providing unambiguous, declarative description of a desired system state and then carrying out necessary steps to reach it. Operating at even higher level are orchestration management tools, like Juju[11], which are designed to coordinate complex systems in flexible and reactive ways, usually in the cloud computing context.

Researchers start now to take advantage of cloud computing for experimentation. Tools such as Docker[12], Vagrant[13] and packer[14] have gained acceptance for creating reproducible environments for development that can be easily deployed in a variety of cloud computing providers and virtualization technologies. Kameleon [112, 49] is an appliance builder that strives to offer a reproducible environment for experimentation that can be regenerated and changed any time. It does so by taking advantage of a persistent cache mechanism that guarantees that the same software versions are used all the time, avoiding incompatibility issues. This tool constitutes one of the contributions of this thesis and as such will be described thoroughly in Part III.

2.7.2 Tools for capturing experimental context

As mentioned in Section 2.2.2 one important feature required given the complexity of software nowadays, is the capture of the experimental context, undoubtedly useful to the reproduction of an experiment. There are different levels for capturing the context which depends mostly on the kind of experiment one wants to run. Experimenters can take advantage of version control systems (e.g., Git, Subversion) or more sophisticated frameworks like Sumatra [37] which aims at recording and tracking the scientific context (i.e., changes in code or parameters and the motivations for those changes) in which a given experiment was performed. This enables researchers to have provenance in their experiments. Sumatra context capturing is limited to the middleware used. At the moment in only works with applications written in Python. To enable a complex re-executability of a given experiment, all the software dependencies have to be tracked and packed. This is the approach followed by CDE [57] which makes possible to move the experimental environment into different Linux distributions and versions. Reprozip [29] is a more sophisticated tool that follows the same principle and adds provenance information that is captured in a Vistrails workflow.

2.7.3 Tools for making the analysis reproducible

The generation of the valuable raw data from an experiment is a very costly process. Therefore, it should be expected that anyone would have access to the datasets and the analysis procedure carried out for generating certain figure or table and in turn a given conclusion. This could be done with the goal of verifying that a proper statistical study was performed or simply and most importantly enabling the conduction of alternated analysis that could lead to new conclusions.

With the aforementioned goal in mind, a R package shown in [100] is able to cache intermediate results that are stored in a database, enabling researchers to re-execute parts of the analysis. A more advance approach [54] introduces the discipline of *Verifiable Computational Research*. Its implementation creates identifiers that are associated to a given result in a data analysis process. This association uniquely links results of a computation with its context (e.g., software package dependencies, screen messages echoed, platform name and version, etc). The created identifiers can be embedded into documents for publication. *Literate programming* encourages the mix of sections of computer code and natural language with the objective of providing two types of view: documents intended for human consumption and pure source code for examination and execution.

[9] `https://puppetlabs.com/`
[10] `http://www.opscode.com/chef/`
[11] `https://juju.ubuntu.com/`
[12] `https://www.docker.io/`
[13] `http://www.vagrantup.com/`
[14] `http://www.packer.io/`

This approach is followed by *knitr* [15] which is able to generate dynamic documents by embedding R code into LaTeX. Org-mode is an emacs extension for practicing *Literate programming* providing to the user the possibility of embedding a variety of computer programming languages that can be mixed and different types of output are possible (e.g., HTML, LaTeX, DocBook, etc).

2.7.4 Workload generators

It cover all the tools and data that enable to evaluate distributed systems under semi-realistic, controlled and reproducible conditions. Benchmarks such as NAS [16], Linpack [17] have been used over years for evaluating performance of parallel systems. In the field of scheduling of parallel systems there has been an important work by Dror Feitelson which gather together in the *Parallel Workloads Archive* site [18] a considerable number of logs of large scale parallel systems in production. The failure trace archive (FTA) [19] is a public repository of availability traces of parallel and distributed systems. Those traces can be the input of workload models or tools that enable to replay them in real systems [82, 128]. Xerxes [82] is a distributed load generation framework for cloud computing that enables large scale experimentation. It is able to generate load patterns at both individual node level, and collectively across a large number of machines.

2.7.5 Distributed emulators

Emulation along with simulation is one of the techniques highly used in experimentation with distributed systems which enable to augment and control the parameter space. It is mainly targeted at enable reproducible experiments at large scale. Different strategies have appeared for emulating large and high performance machines. In [68] is described an approach for taking advantage of the heterogeneous architectures composed of CPU and GPUs widely common nowadays for emulating different kinds of parallel machines [20] using OpenCL. A parallel version of the well known emulator Qemu is proposed in [41] for emulating efficiently multicore machines. For emulating the heterogeneous nature of computational grids EHGRID [34] was proposed that provides mechanism for degrading the performance of computer processors turning an homogeneous architecture into an heterogeneous one. Additionally, it takes into account network effects for inter-cluster communication. Distem [115] follows the same philosophy of EHGRID but it is targeted to a wider community, including cloud, P2P, High Performance Computing and Grid systems. It relies on LXC (Linux Containers) which makes it efficient and scalable, enabling the building of 15000-nodes virtual topology in no time.

2.8 Conclusions

In this chapter, we presented an extensive list of properties expected from general-purpose experiment management tools for distributed systems on real platforms. The diversity of the research domain of distributed systems motivated development of different techniques and tools to control experiments, and explains the multitude of approaches to manage experiments. With the construction of the feature list, we tried to establish a common vocabulary in order to understand and compare the existing experiment management tools.

The size and complexity of distributed systems nowadays has uncovered new concerns and needs in the experimentation process. We need to control an always increasing number of variables to assure two important characteristics of an experiment, its reproducibility and replicability.

[15] `http://yihui.name/knitr/`
[16] `http://www.nas.nasa.gov/publications/npb.html`
[17] `http://www.netlib.org/linpack/`
[18] `http://www.cs.huji.ac.il/labs/parallel/workload/`
[19] `http://fta.scem.uws.edu.au/`
[20] according to the Flynn's taxonomy: Single Instruction, Single Data stream (SISD); Single Instruction, Multiple Data stream (SIMD); Multiple Instruction, Single Data stream (MISD); Multiple Instruction, Multiple Data stream (MIMD).

With the motivation of providing a controlled environment to execute experiments in the domain of distributed systems, several testbeds were created which stimulated the development of different experiment management tools. Among the benefits of experiment management tools are: encouraging researchers to experiment more and improve their results, educational value of being able to play with known algorithms and protocols under real settings, reduction of the time required to perform an evaluation and publish results, capacity to experiment with many nodes and complex scenarios, different software layers, topologies, workloads, etc.

Despite the emergence of experiment management tools, some of them are in an immature state of development which prevents them from fully exploiting the capacity of certain testbeds. There is indeed, a lot of challenges in the domain of experimentation and the need of further development of those tools is apparent. To achieve this, technologies developed with different purposes could arguably be used in the experimentation process. For instance, we mentioned that workflow systems and configuration management tools share some concerns and goals with the problem of experimenting with distributed systems.

Finally, a deeper understanding of the experimentation process with distributed systems is needed to identify novel ways to perfect the quality of experiments and give researchers the possibility to build on each others' results.

Part II

Expo

Chapter 3

Expo: a tool to manage large scale experiments

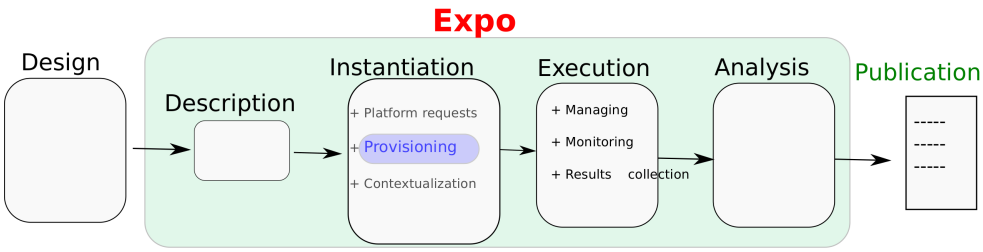

Figure 3.1: Role of *Expo* in the experiment cycle

Performing experiments that involve a large amount of resources or a complex configuration, proves to be a hard task. In this chapter we present *Expo*, which is a tool for conducting experiments on distributed platforms. *Expo* is the result of an effort to bring the scripting of experiments to the next level. It encourages the scripting of experiments by offering a set of abstractions to deal with big and complex computational infrastructures. Additionally, it provides mechanisms that make experimenters more productive when setting up their experiments. Its goal is to improve the state of the art of experimentation by encouraging their complete automation. First, the architecture of the tool is described along with its abstractions for resources and tasks that reduces the complexity in the experiment conduction. Next, the tool is compared with other similar solutions based on some qualitative criteria, scalability and expressiveness tests. The chapter finishes with the evaluation of *Expo* scalability and some use cases on Grid'5000 and PlanetLab testbeds. Our experience showed that *Expo* is a promising tool to help users with two primary concerns: (1) performing a large scale experiment efficiently and easily, (2) describing an experiment with enough detail that enables posterior reproduction. The content of this chapter was published in a paper presented at PDCN2013 [113].

3.1 Introduction

Although the software to perform simulations has improved in recent years, there is still the need to test and evaluate the software in real distributed infrastructures. Moreover, the option of experimental evaluation of an algorithm has been encouraged as an approach complementary to the theoretical evaluation [73]. In order to address limitations such as, software reconfiguration, lack of control and monitoring systems, testbeds were created [88]. A testbed is a platform for experimentation with large distributed applications. It is sometimes shielded from the instabilities of production environments and allows users to test particular modules of their applications

in an isolated fashion. Some examples of testbeds are: PlanetLab [103], Emulab [130], GENI [1], Grid5000 [25] and ORBIT [108, 98] (see Section 2.2.3). Although these platforms offer more stability and control over resources, it is still a hard task to control, deploy and run applications on them. In more detail a number of tasks must be completed before an experiment can be actually started. These tasks include resource discovery and acquisition as well as deployment of the necessary software. Once the application is launched, its execution must be controlled, and as soon as it finishes all the output must be collected. Most of the experiments performed on the testbeds are run in an ad-hoc, application-specific manner. This method may match the current requirements of experiments, but fails with the scale, heterogeneity, and dynamism of distributed systems. That is the reason why we have seen the apparition of experiment management tools that strive to cope with the problems encountered when researchers try to perform experiments involving a large amount of resources or a complex configuration. The reader is referred to Section 2.2.2 for a full list of motivations behind those tools. The main aspects those tools help the user with, are: (1) description of the experiment, (2) control and access to the resources, (3) task orchestration, (4) software deployment, (5) monitoring and collection of results. The main advantage of those tools is the possibility of embedding all the important details - that took part on the process of experimentation - using the same language. This will hopefully make easier the reproduction of a given experiment. The objective of this chapter is to present our experiment management tool called *Expo* that has already been introduced shortly in the previous chapter and qualitatively compared against existing works. *Expo* is the result of an effort to bring the scripting of experiments to the next level. It encourages the scripting of experiments by offering a set of abstractions to deal with big and complex computational infrastructures. Additionally, it provides mechanisms that make experimenters more productive when setting up their experiments. Our objective is to improve the state of the art of experimentation by encouraging their complete automation. In Chapter 1, it was shown the experiment cycle normally followed in research. *Expo* covers the description, instantiation, execution and analysis of an experiment as shown in Figure 3.1. In this chapter, *Expo* architecture, features, abstractions and syntax and their advantages will be exposed. *Expo* will be compared with the most used and actively developed experiment management tools. One of the main contribution of *Expo* is that it enabled the rapid prototyping of experiments and this will be demonstrated on Chapter 4.

The structure of this chapter is as follows: In the next section *Expo* is presented in depth with its features and advantages, some use cases are shown in Section 3.3 in two different testbeds. Results and comparisons with other experiment tools are presented in Section 6.5. Related works in software engineering are presented in Section 3.5 and finally Section 6.6 presents the conclusions and future works.

3.2 Expo

Expo is an experiment management tool designed to simplify and automate the conduction of experiments in distributed platforms. All the experimental plan is captured (i.e., access to the platform, experiment setup, experiment execution, results analysis, etc.) in a workflow where sequences of commands are grouped together in tasks and dependencies. This facilitates the recreation of the experiment setup and in turn, it will make easier the replay of experiments. Replayability of a computational experiment is the first step towards experiment reproducibility. The workflow tells how all the different tasks have to be called in order to get the results of the experiment. It comprehends tasks that can be executed sequentially, in parallel, asynchronously, etc. *Expo* strives to simplify the description of an experiment by providing a concise and readable way to describe it, specially when dealing with a big amount of nodes. It relies on parallel command executors such as TakTuk [33] which makes it scale with a big amount of nodes. TakTuk uses an adaptive and reactive work-stealing algorithm that mixes local parallelization and work distribution. A topology of deployment can be specified and this is exploited by the *Expo ResourceSet* abstraction presented in subsection 3.2.1.

[1] `http://www.geni.net`

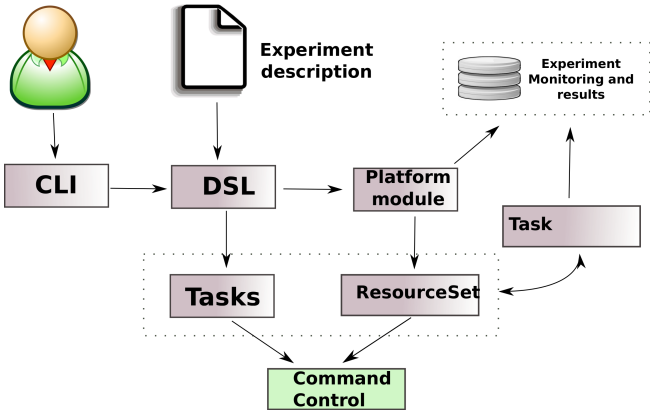

Figure 3.2: Expo architecture

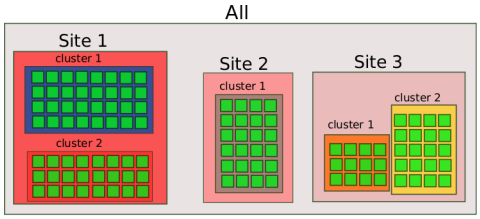

Figure 3.3: Example of resourceSet

Expo architecture is described in Figure 3.2, which mainly consists in six components: an internal Domain-Specific Language (*DSL*) [2] module features a flexible description language built on top of Ruby [3]. It enables to exploit all its richness in available libraries and mainly its descriptiveness. The *DSL* flexibility and scalability relies on two abstractions: *ResourceSet* and *Tasks*. Those abstractions are mapped into components that interact together in order to provide the necessary information to the *Command Control* and help it in translating the experimental plan into commands. The platform dependent module enables the interaction with different platforms such as: Grid'5000, PlanetLab, cloud computing infrastructures, computing clusters, etc. This module works as an interface for the *DSL* module, making an experiment description independent from the platform. *Expo* makes few assumptions about the resources to manage, relying on common system utilities suchs as: scp, ssh, unix commands, TakTuk which can deploy itself. It only requires to run a Ruby interpreter and few ruby libraries as described in its website [4]. Thus, *Expo* architecture is very simple and ligthweight. The schedule of the experimental workflow is done by the *Task manager* which is in charge of the results collection and experiment monitoring. Two execution modes are possible: interactive and standalone which execute the experiment description file without any user intervention.

3.2.1 Expo ResourceSet

A *ResourceSet* is an abstract view of the resources and their organization in distributed computational infrastructures such as Grids. It adds resources into a logical unit and associates properties to them. For instance, we can gather together the nodes from the same cluster associating to them the same frontend, as well as the same physical properties if the cluster is homogeneous. This abstraction was conceived in order to provide to the user a concise way to express actions that have to be carried out for a set of resources. Resources can be any computing unit: cores,

[2] an internal DSL means that is hosted in another language and can take advantage of its constructs.
[3] `https://www.ruby-lang.org`
[4] `http://expo.gforge.inria.fr/`

runs the command in parallel for all the nodes of the cluster 1	`run("make lu NPROCS=8 CLASS=A MPIF77=tau_f90.sh",:target => resources[:cluster_1])`
runs the command hostname for each node sequentially	`resources.each{ |node| run("hostname",:target => node) }`

runs the command for different set of resources, the length of the sets generated are powers of two.	`resources.each_slice_power2 do |nodes|`
	`  run("mpirun -np 2 --machinefile #{nodes.nodefile} ./app",:target => nodes.first)`
	`end`

selects the resources of a specific cluster, it keeps the topology of the *ResourceSet* in order to generate the right parallel command.	`fast_cluster = resources.select(:cluster){ |cluster|`
	`  cluster.properties["clock_speed"]>1700000000`
	`}`
	`run("~/benchmarks/NPB3.2-OMP/bin/BT.A_out.4",:target => fast_cluster)`

Table 3.1: ResourceSet operations

processors, nodes, clusters, sites, etc. Table 3.1 shows some operators which gives to *Expo* a high flexibility against another approaches in the description language as will be shown in Section 3.4. An example is shown in Figure 3.3 where a Grid computing like hierarchy is represented, this abstract view enables the generation of efficient parallel topology aware commands. We can divide the resources belonging to the same site as well as separate them per cluster. This can also be applied for the PlanetLab testbed, the *ResourceSet* can have information about the location of the resources for the same country or site. In other cases, it can be used to define complex configurations as in the case we would need to deploy an infrastructure where different nodes have different roles.

3.2.2 Expo Tasks

Expo adopts the notion of task, already exploited in workflow management tools as [120] and Rake [5] as well as web application deployment frameworks such as Capistrano [6]. A *Task* describes what to do and the *ResourceSet* tells the experiment management where to execute the task. Tasks can be triggered by events (e.g, availability of jobs in the infrastructure, errors, etc.). Therefore, a complete unattended experiment campaign can be carried out. In Listing 1, an example of a definition of a task is shown. The compilation of a source code instrumentation package is performed. This task is executed on a *ResourceSet* which is represented by the variable *resources*. For this case a parallel command will be generated that will carry out the task for every machine represented in the *ResourceSet*. This task could be useful when compiling a program for different architectures.

```
1 task :compile, :target => resources do
2   run("cd ~/Test_profiling/; tar -xf pdt.tgz")
3   run("cd ~/Test_profiling/pdtoolkit-3.17/; ./configure")
4   run("cd ~/Test_profiling/pdtoolkit-3.17/; make install")
5 end
```

Listing 1: Task abstraction

3.2.3 Expo interactive console

An interactive mode is proposed driven by the following reasons: (1) an important amount of the experiments are interactive [7] (2) the writing of an experiment description file is a trial-and-error process which involves using different parameters, configurations and flows of control, (3) *An*

[5] http://rake.rubyforge.org/

[6] https://github.com/capistrano/capistrano/wiki

[7] 53% of the experiments are interactive, against 47% that are run in Batch mode. Results obtained consulting the Grid5000 API

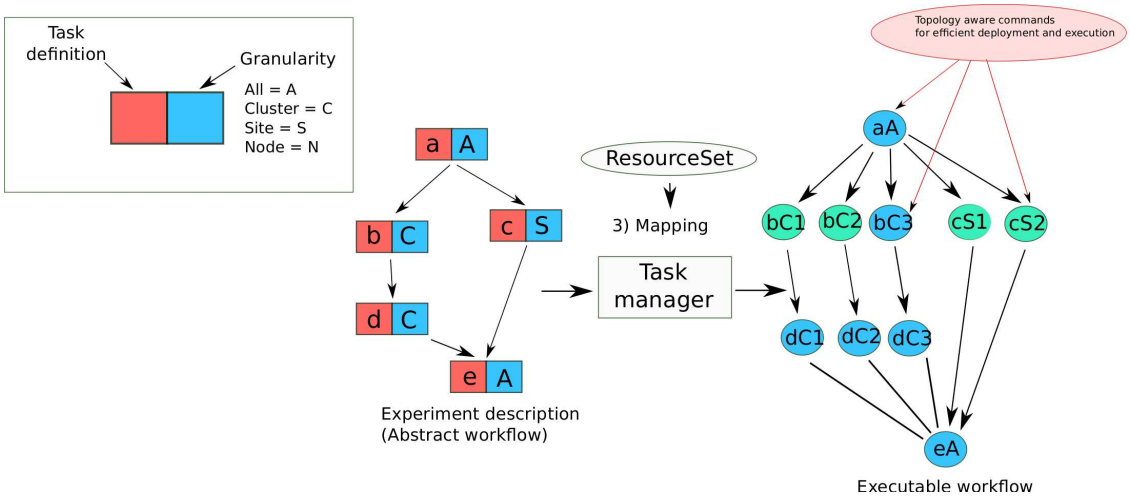

Figure 3.4: Expo workflow mapping. Tasks are split according to the granularity of execution, generating sub-tasks for the executable workflow. In the Figure, tasks are generated for 3 different clusters and 2 sites. The *Task manager* uses the information provided by the *ResourceSet* to generate the topology aware commands

interactive environment lets scientists look at data, test new ideas, combine algorithmic approaches, and evaluate their outcome directly [102]. This approach is already used by different scientific environments based on Python such as: IPython and Scipy [74]. This interactive mode can also be triggered by an error during a standalone execution, providing either a shell console or a Ruby console where the user can modify and verify the execution of the *Expo* DSL.

3.2.4 Expo experiment validation

Given that the whole workflow of an experiment could take hours to execute, it is important to avoid errors like the utilization of undeclared variables. One important feature that *Expo* offer is the validation of the experiment description. It does so through the use of two mechanisms, it first perform a static analysis of the experiment description and then it runs the logic of the experiment without executing any real action. This is equivalent to the mode *dry run* offered by configuration management tools. This helps the experimenter to verify that the experiment workflow will be executed in the desired manner.

3.2.5 Expo experiment mapping

Workflow engines map scientific workflows to distributed platforms in an automatic form. Their mapping decisions are driven by minimizing the time to run the workflow. Given that the objective of a workflow is to perform a big computation, it is more flexible when mapping the workflow into the computing platform. In contrast, an experimenting workflow aims at performing tests. Some tests are targeted to a certain machine architecture and it is important to take this into account when performing the mapping of the workflow. Consequently, a way to control the underlying infrastructure has to be provided. There is a trade-off between descriptiveness and scalability (efficient mapping). Figure 3.4 explains the procedure to map an experiment description into a distributed platform, in this particular case a Grid computing infrastructure. There are some tasks that should happened at the site level like the transfer of large files that can be shared between all the machines of the cluster using a network file system. Compilation tasks must be executed at cluster level because sites could be composed of several clusters with different architectures.

As already said, an experiment is described as a workflow composed of tasks and dependencies between them. This initial workflow is known as abstract workflow and has as a goal to capture

the experiment activity. Two important information are: the body of the task which is simply all the sequence of commands to execute and the granularity of execution. For the example shown, this granularity can be: all resources, site, cluster, node, etc. The task manager will be in charge of taking this abstract workflow and map it into the infrastructure. It uses the information provided by the granularity of execution in order to generate the executable workflow. This is an expanded version of the abstract workflow, where tasks have been split according to the granularity of execution. This enables to choose the best type of execution (parallel, asynchronous, parallel-asynchronous, etc.) and the less expensive in terms of number of connections with the remote machines and threads created to control the experiment. The tasks created at this level guarantee the generation of topology aware commands with TakTuk for an efficient deployment and execution. The scalability of commands execution will be shown in the following sections.

3.2.6 Expo evolution

During this thesis we have extended and improved in several ways the already existing implementation of *Expo* [125, 124]. We have added the task abstraction which helps to structure the experiment description and form a workflow. This makes the experiment description more readable and the detection of bugs easier. This task abstraction can interact with the *ResourceSet* for controlling the mapping of tasks into different levels of the defined infrastructure hierarchy. The new opportunities brought by this mapping will be shown in the Chapter 4. One important improvement is the support of experiment validation by default. This was one of the drawbacks of previous versions of *Expo* which made the setup of experiments costly and error-prone. Additionally, an interactive mode was implemented to boost experimenter's productivity by allowing her/him to debug the whole experiment description.

3.3 Use cases

```
1  require 'expo_planetlab'
2
3  set :resources, "MyExperiment.resources"
4  get_resources
5
6  task :monitoring, :target => resources do
7
8       File.open("Planetlab_avail.txt",'w+'){|f|
9            res=nil
10           f.puts "Date  Time  Num_Res"
11           240.times{
12               date_measure=Time::now.to_i
13               res = run("hostname")
14               time=res[:run_time]
15               f.puts "#{data_mesure}  #{time}  #{res.length}"
16               f.flush
17               sleep(60)
18           }
19      }
20 end
```

Listing 2: Monitoring nodes availability in Planetlab using Expo

The aim of this section is to show the syntax for writing an experiment using *Expo*. Listing 2 shows a simple experiment for monitoring the nodes availability on Planetlab. This is done by executing the linux command *hostname* on all the nodes of the slice and counting how many of them reply. This information is written into a file that can be used to plot the availability of the nodes over time in the slice.

```
require 'g5k_api'

set :user, "root"
set :gw_user, "cruizsanabria" ## replace with your user
set :resources, "MyExperiment.resources"

reserv = connection(:type => "Grid5000")
reserv.resources = {:nancy => ["nodes=1"], :rennes => ["nodes=1"], :lille => ["nodes=1"], :grenole=> ["nodes=1"]}

reserv.environment = "http://public.nancy.grid5000.fr/~cruizsanabria/tlm_simulation.env"
reserv.name = "TLM multisite"
reserv.walltime = 2000

##### Tasks Definition ####################################
task :run_reservation do
    reserv.run!
end

task :config_ssh do
    msg("Generating SSH config")
    File.open("#{expo_cwd}/config",'w+') do |f|
        f.puts "Host *
                StrictHostKeyChecking no
                UserKnownHostsFile=/dev/null "
    end
end

task :generating_ssh_keys do
     run("mkdir -p #{expo_cwd}/temp_keys/")
     run("ssh-keygen -P '' -f #{expo_cwd}/temp_keys/key") unless check("ls #{expo_cwd}/temp_keys/key")
end

task :trans_keys, :target => resources do
     put("#{expo_cwd}/config","/root/.ssh/")
     put("#{expo_cwd}/temp_keys/key","/root/.ssh/id_rsa")
     put("#{expo_cwd}/temp_keys/key.pub","/root/.ssh/id_rsa.pub")
end

task :copy_identity do
     resources.each{ |node|
       run("ssh-copy-id -i #{expo_cwd}/temp_keys/key.pub root@#{node.name}")
     }
end

task :deactivation_ib do
     resources.each{ |node|
       run("/sbin/ifconfig ib0 down")
     }
end

task :run_simulation, :target => resources.first do
     put(resources.nodefile,"/root/TLMME_multimode/nodes.deployed")
     run("/root/TLMME_multimode/exec_tlm 1 369 192 510 250 1 sc")
     get("/root/TLMME_multimode/profile.*","~/profiles")
end

task :free_reservation, :target => resources do
     free_resources(reserv)
end
```

Listing 3: Profiling of a parallel application running on multiple sites in Grid'5000 using Expo

Listing 17 shows the automation of the execution of a parallel application using several sites in Grid'5000. The objective of the experiment is to perform a profiling of the parallel execution of an electromagnetic simulation using TAU [8]. We deployed an operating system image with all the software already installed using Grid'5000 API that interacts with *Kadeploy* [71]. This image was generated using *Kameleon* that will be presented in Chapter 5. The specification of the corresponding image to deploy is indicated as a parameter in the function that request the resources, which is shown in the first lines of the file. Moreover, in the file we can see some *Expo* operators to ease the procedure of execution of commands on several nodes through the use of iterators. This makes easier the description of tasks such as deactivating infiniband interfaces

[8] `http://www.cs.uoregon.edu/research/tau/home.php`

on all reserved nodes. Another operator is shown for generating the correct hostfile necessary for running a MPI application. Finally, we execute the application and we get the profile generated by TAU during the execution. All the results are sent to the experimenter's machine. The modularity of the tool enables users to run their experiment in another testbed by just loading the appropriate module. Other use cases will be shown throughout all this thesis and mainly in the next chapter where *Expo* was used for performing a custom calibration of Grid'5000 clusters that enabled the efficient deployment of multisite parallel applications. *Expo* use cases include:

- Evaluation of processes placing in the deployment of a parallel application.
- Calibration of Grid'5000 processors for an electromagnetic application.
- Comparison of the two techniques of deployment: naive and hardware aware.
- Generation and collection of traces of NAS [9] benchmarks using TAU.

These examples are included in the Appendix A of this thesis.

3.4 Evaluation of experiment control systems

```
<?xml version="1.0" encoding="utf-8"?>
<gush>
  <project name="Testing overhead">
    <component name="Cluster1">
       <rspec>
          <num_hosts>20</num_hosts>
       </rspec>
       <resources>
         <resource type="ssh" group="local"/>
       </resources>
    </component>

        <experiment name="simple">
           <execution>
              <component_block name="cb1">
                 <component name="Cluster1"/>
                 <process_block name="p2">
                    <process name="test">
                        <path>hostname</path>
                         <cmdline>
                           <arg></arg>
                         </cmdline>
                    </process>
                 </process_block>
              </component_block>
           </execution>
        </experiment>
  </project>
</gush>
```

Listing 4: Gush description

```
require 'g5k_api'

set :user = "cruizsanabria"
set :resources = "MyExperiment.resources"

reserv= connection(:type => "Grid5000")
reserv.resources = { :nancy => ["nodes=200"]
                     :sophia => ["nodes100"]}

reserv.name = "Expo Scalability"
reserv.walltime=2000


task_definition_start

task :run_reservation do
   reserv.run!
end

task :scalability do

   sizes=[10,50,100,200,300]

   resources.each_slice_array(sizes) do | nodes|

      run("hostname", :target => nodes)
      # have to put tags here
   }

end
```

Listing 5: Expo Experiment description

Listing 6: Comparison between experiment description files: These files were used in the evaluation of the scalability of the two tools. It should be noticed here that the experiment description for Gush has to be changed every time we need to change the number of nodes to try with. Also Gush needs a file for the resource description that is not shown.

The aim of this section is to position *Expo* in the panorama of experiment management tools. In this thesis, we have already performed a qualitative comparison of the experiment management tools in Chapter 2. In this section the goal is to carry out a deeper comparison of similar approaches for conducting experiments on distributed infrastructures. We have chosen: *Gush*, *Execo* and

[9] http://www.nas.nasa.gov/publications/npb.html

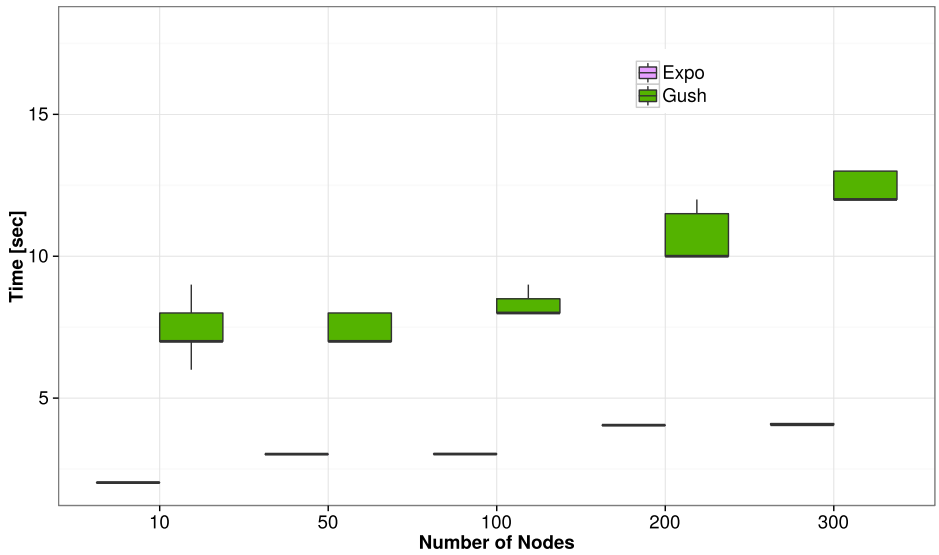

Figure 3.5: Evaluation of the scalability of Gush and Expo when executing a command in a large set of resources. The upper and lower "hinges" correspond to the first and third quartiles. Points that are out of this range, represented outliers. Each test was repeated 10 times.

XpFlow. These three tools share many features with *Expo* such as the ease of installation, the capacity to adapt to different testbeds and they are targeted at performing general experiments in distributed infrastructures involving a big amount of nodes. First, we evaluate *Expo* against *Gush* given that they used very different approaches to describe the experiment as well as different philosophies. Then, we evaluate *Expo* against *Execo* and *XpFlow* which have been developed with the purpose of managing large scale experiments.

3.4.1 *Gush* comparison

The evaluation consisted in the expressiveness of the language, as well as the performance and scalability of the command execution. The comparison between both tools was done by carrying out an experiment, which involved a large amount of nodes. We defined an experiment that consisted in executing a command in a set of resources and measuring the time elapsed, while varying the number of nodes. Therefore, we compare the time to execute the commands and the flexibility in the description of the experiment. Listing 6 shows the descriptions of the experiment used for *Gush* and *Expo*. We can note, looking at the experiment description, that for *Gush* we have either to change the file for each experiment so as to try different number of resources, or we can create a long description file with all the possibilities we want to try. This is not the case for *Expo*, which uses Ruby and provides a programmatical approach for describing the experiment, making it flexible enough to adapt to the normal activities or changes when we perform an experiment.

Figure 3.5 shows the scalability of the mechanism for the execution of commands. In this figure we can see that *Expo* outperforms *Gush* due to the use of TakTuk parallel executer, also that *Expo* presents less variability in the time to execute the experiment, which is important to the reproducibility. It was noticed as well that when we tried to execute an experiment with more than 400 nodes, problems arise trying to perform it with *Gush*.

3.4.2 *XpFlow* and *Execo* comparison

There has been a recent interest for developing experiment management tools targeted at complex experiments with distributed systems. From the tools that have been studied in Chapter 2 two tools deserve special attention *XpFlow* and *Execo* given that they are actively developed and used

by the Grid'5000 community. Additionally, they have been used in recent publications [67, 20]. At the moment of writing the versions of *XpFlow* and *Execo* used were respectively 0.1c and 2.3 [10]. This evaluation goes a step further compared to the previous evaluations. We implemented first a scalability experiment using Taktuk, the three tools support it for running experiments at large scale. The three different experiment descriptions are shown in Figure 10. Resources were reserved on 9 different sites (*nancy*, *sophia*, *toulouse*, *lille*, *lyon*, *luxembourg*, *nantes*, *grenoble* and *rennes*) in Grid'5000. Therefore all three tools received as a parameter the same set of resources. The experiment consist simply in executing the command hostname over a set of resources and measuring the time it took to carry out this task. Different sizes of nodes were tested as can be observed on the experiment description files. The results of the test are shown in Figure 3.6, we can observe that *Expo* scales better with an increasing number of nodes. This is due to the fact that it takes into account the topology of the infrastructure which is captured in the *ResourceSet* abstraction and helps to generate the right parameters for TakTuk. With the implementation of these experiments and the ones shown in the Appendix A, we gained some insights and discuss some features provided by those tools.

Description language

From the description point of view when evaluating these tools we had an interesting case study because each tool offers a different degree of abstraction. Going from the simple plain script provided by *Execo* to the most sophisticated workflow representation offered by *XpFlow*. *Expo* sits on the middle providing the *Task* abstraction to structure the experiment description. *Execo* provides an *API* for controlling remote processes, contrary to *Expo* and *XpFlow* that provide an internal[26] DSL oriented to the domain of experimentation. Each representation has its advantages and disadvantages, having a low level *API* as the one provided by *Execo* enables a fine grain control of running applications. They can be started, monitored and stopped and the workflow of the experiment can be easily modified using all the syntax and language constructs provided by Python. In the other hand, *Expo* and *XpFlow* impose their proper constructs to specify the experiment workflow. This brings modularization and makes experiment description more comprehensible. As a conclusion, we believe that the good level of abstraction will depend on the type of experiment and its complexity.

Experiment validation

One important fact that characterizes the evaluated tools is that they used interpreted languages as a means for describing the experiments. This brings high flexibility for interacting with computing systems as is demonstrated by the fact that more than 50% [11] of configuration management tools are implemented using this kind of programming languages. However, the naive use of these programming languages can have a big cost for the conduction of experiments, as simple errors like the use of undeclared variables, undefined methods, invalid arguments, etc., could break the experiment workflow and lose its progress. This is a drawback of *Execo* that by default do not integrate any validation mechanism for caching the aforementioned errors before running the experiment. *XpFlow* detects undeclared variables and undefined methods before running the experiment, stopping its execution and presenting an error to the user. Unfortunately this only happens at the level of the process abstraction, activities that are used as building block and wrap low level tasks, do not count with this type of validation. *Expo* as already presented, provides two mechanisms: static code analysis and *dry run*.

Experiment checkpoint

Execo provides checkpointing support for parametric studies. It provides a class to perform parameter sweeps which uses a local directory in disk for saving the progress of the parameter combination

[10] Those versions were accessed on 24/09/2014.

[11] Checking language used by the most popular projects: Ansible, Bcfg2, cdist, Chef, CFEngine, juju, Puppet, Salt, Rexds

```
require 'g5k_api'

set :resources, "MyExperiment.resources"
set :user, "cruizsanabria"

reserv = connection(:type => "Grid5000")
reserv.create_resource_set_file("nodes_expe")

RUNS = 5

task :scalability do
  sizes = [2,4,8,16,32,64,128,256]

  resources.each_slice_array(sizes) do |nodes|
    msg("Testing with #{nodes.length}")
    RUNS.times{
      run("hostname", :target => nodes)
    }
  end
end
```

Listing 7: Expo experiment description

```
process :main do
  log "Starting Experiment"
  RUNS = 5
  ip_adresses = YAML::load(File.read("nodes_expe"))
  nodes = []
  ip_adresses.each{ |ip|
    nodes.push(simple_node("cruizsanabria@#{ip}"))
  }

  [2,4,8,16,32,64,128,256].each do |size|
    test_nodes = nodes[1..size]
    log("Testing with #{size} nodes")
    RUNS.times{
      r = execute_many(test_nodes, "hostname")
      log(r)
    }
  end
end
```

Listing 8: XpFlow experiment description

```
from execo import *
from execo_engine import *
import yaml

class taktuk_scalability(Engine):

    def run(self):
        RUNS = 5
        with open('nodes_expe', 'r') as f:
            ip_address = yaml.load(f)

        hosts = []
        for address in ip_address:
            hosts.append(Host(address, user = 'cruizsanabria'))

        time = Timer()
        logger.info("Starting Experiment")

        for i in [2,4,8,16,32,64,128,256]:
            test_hosts = hosts[0:i]
            for i in range(RUNS):
                servers =TaktukRemote("hostname",test_hosts)
                servers.start()
                servers.wait()
                print Report([servers]).to_string()


        logger.info("Total execution time = %f" % time.elapsed())
if __name__ == "__main__":
    engine = taktuk_scalability()
    engine.start()
```

Listing 9: Execo experiment description

Listing 10: Comparison between experiment description files: These files were used in the evaluation of the scalability using taktuk. We can observe the different abstraction used by the tools and their syntax sugar.

that have already been tested. However, it does not support the checkpoint of any experimental workflow. *XpFlow* is able to save the progress of any experimental workflow by saving the state of all variables used in the experiment description. Thus, if the execution faces any eventual error, users can react, fix the error and continue to execute the experiment from the point it stopped. *Expo* does not support experiment checkpointing, instead it provides an interactive mode that is triggered when an error occurs. In this way it serves the same function of *XpFlow* checkpointing mechanism. As a consequence, the checkpoint mechanisms provided are either specific for a kind of experiment or does not take into account the state of the platform. We have to remark here

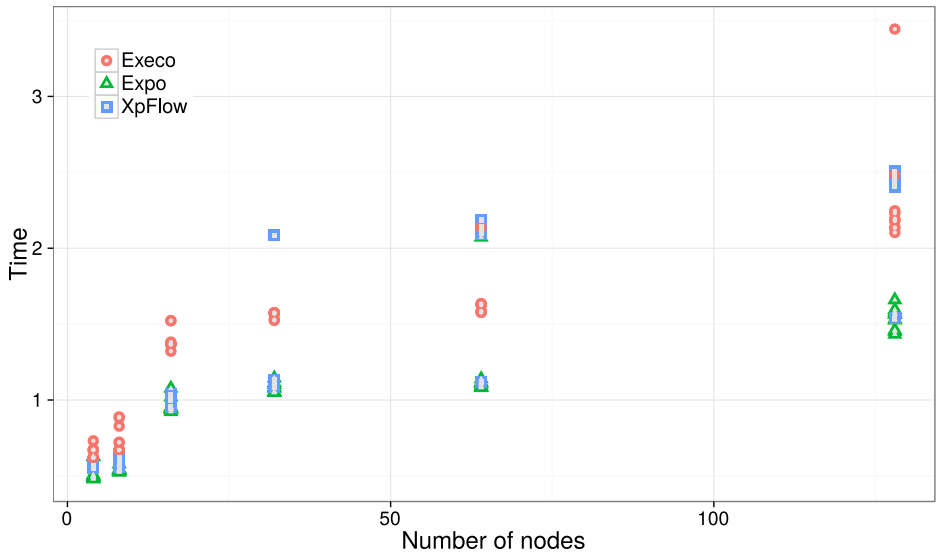

Figure 3.6: Evaluation of the scalability of Expo, Execo and XpFlow when executing a command in a large set of resources. Each test was executed ten times.

that the real sense of a checkpoint mechanism (to save the progress of an experiment) is difficult to implement. There are some difficulties such as the need of large amount of storage capacity and the capturing of the network state, those problems are addressed by works in the checkpoint of parallel applications and the snapshotting of whole virtual infrastructures [80].

3.5 Related works

Chapter 2 presented a complete state of the art in experiment management tools. Here we present two fields of constant research in software engineering that shares similar concerns with *Expo*:

- Deployment of complex distributed applications
- Regression tests for distributed applications

Those fields of research have produced a plethora of tools that seeks to remove the error-prone nature of human intervention by encouraging automation. They aim at reducing the burden of configuring and testing distributed applications.

3.5.1 Deployment of complex distributed applications

Due to the limited scalability and error-prone nature of manual approaches several tools have been developed to make easier the deployment of applications and their pre-requisites in distributed infrastructures. ADEM [62] is an automation tool for the deployment and management of grid application software. It manage efficiently the deployment and building of applications (compiling and installation of dependencies) over different grid sites. It takes into account platform heterogeneity through the use of signatures. Tune [17] is a tool to manage software in distributed infrastructures. The goal is to make easier the administration and deployment of multi-tiered applications [12]. It is based on the concept of autonomous computing for making the administration of an infrastructure as a component architecture. The main idea is to automatically create a representation based on fractal components of the real system, with two main parts: application

[12] applications that depend on different services (e.g., databases, web servers, load balancers, etc).

components and platform components. All expressed with a subset of UML diagrams. It has already been used in the installation of a cluster software and the deployment of an electromagnetic simulation code in a grid infrastructure [83]. Another work [50] address the deployment of applications in IaaS clouds. It proposes a decentralized protocol to automatically deploy applications consisting of interconnected software elements hosted on several VMs. It uses an XML-based formalism to describe the cloud applications to be deployed. *Expo* differs from the aforementioned tools in that it offers a more flexible, programmatic approach for the description of the experiment and it is designed to interact with a large number of nodes.

3.5.2 Regression tests for distributed applications

Regression tests encompass different principles aiming at the rapid test and deploy of changes in software. Those kind of tests when applied to distributed systems are hard, because applications should start efficiently and in a correct order. Additionally, they have to meet complex dependencies as the ones required by multi-tiered applications (e.g., database URL, load balancers, etc.). DART [31] was developed to facilitate the writing of distributed tests for large-scale network applications. It provides a language based on XML to specify high level details of test execution. Each test encompasses: setting up the required infrastructure, distributing code and data to all nodes, executing and controlling the distributed tests and finally collecting the results of the test from all the nodes and evaluate them. It integrates efficient tools for the execution of applications and the transfer of files. NMI [99] is a framework to build and test software in a heterogeneous, multi-user, distributed computing environment. The principal aim is to offer to the user the continual testing of software changes. The user describe the process of building and testing along with its external software dependencies by using a lightweight declarative syntax. It works along with a versioning system to log the results and changes and perform the tracking of all inputs, which ensure repeatable and reproducible tests. Another framework oriented to IaaS Clouds is Expertus [69] which through code generation techniques, automates performance testing of distributed applications. It handles automatically complex configuration dependencies of software applications and it strives to remove human error by fully automating the testing process (i.e., deployment, configuration, execution and data collection). The automation is based on script generation from templates that are specified using XML.

Nixos [123] aims at making distributed application testing as easy to write as unit tests. It provides a specification for automatically instantiate virtual machines for providing the necessary artifacts for tests, namely root privileges, system services, multiple machines, specific network topologies, etc. The system is built on top of Nix [42] the functional linux distribution which enables to provide a concise way to specify VM configurations and an efficient way to build them. The main difference between the tools mentioned in this subsection and *Expo* is the target community. The target community of those tools is most of the time software developers or system administrators which count with high technical skills and this fact is reflected in the type of languages offered to describe the environment of tests. Researchers do not always possess the required expertise to deal with distributed systems complexity and that is why high level abstractions for performing experiments were a design requirement for *Expo*.

3.6 Conclusions and future works

Experimentation in computer science and specially in distributed infrastructures has seen the emergence of different experiment control systems. From this fact we can draw a conclusion that most of the tools distinguish almost the same phases in the experimenting process. There are three main parts of the experiment process that a tool must control and help the user with: (i) the control, (ii) the supervision and (iii) the management of the experiment. The first part comprises the description of the experiment, the capture of data, the definition of the source of data, and how to get it after the experiment has finished, as well as the flow of control of the experiment. This is an important step for the reproducibility of the experiment. Second, the experiment supervision,

which means the monitoring of the experiment. The last phase is the experiment management, which is the interaction with the platform, and mainly consist in taking advantage of the services provided by the infrastructure in order to carry out the experiment.

Expo offers a way to describe the experiment by using a programming language providing a lot of flexibility and, more importantly, the abstractions that allow the user to express complex configurations. We put special attention at automating the typical tasks done when an experiment is performed. Because we think that automating the experimentation process is the way to go, being one of steps that will lead to the experiment reproducibility. Furthermore it is important to encourage the culture of experiment reproducibility, which is acknowledged to be a shortcoming in computer experimentation.

The use of experiment tools will save user time, which can be spent in improving the software itself, it will save costs and allow others to reproduce the results more easily. It is important to integrate some features to *Expo* for the sake of reproducibility, we need to improve the part of the system that logs the experiment execution with the aim of having detailed and easy to treat information. This would enable a possible replay of the experiment. Additionally, it is important to incorporate mechanisms to monitor and to generate a workload, and more importantly, to deal with fails.

Chapter 4

How HPC applications can take advantage of experiment management tools

The heterogeneous nature of distributed platforms such as computational Grids is one of the main barriers to effectively deploy tightly-coupled applications. For those applications, one common problem that appears due to the hardware heterogeneity is the load imbalance which slows down the application to the pace of the slower processor. One solution is to distribute the load adequately taking into account hardware capacities. To do so, an estimation of the hardware capacities for running the application has to be obtained. In this chapter, we present a static load balancing for iterative tightly-coupled applications based on a profile prediction model. This technique is presented as a successful example of the interaction between experiment management tools and parallel applications. The experiment management tool *Expo* is used that enabled to: (1) provide a general, lightweight and descriptive way to capture the tuning and deployment of a parallel application in a computing infrastructure, (2) perform the tuning of the application efficiently in terms of human effort and resources needed. This chapter reports the costs for carrying out the tuning of a large electromagnetic simulation based on TLM for the platform Grid'5000 and the improvements obtained on the total execution time of the application. The contents of this chapter were published in a paper [110] presented at *CCGrid2014*.

4.1 Introduction

High Performance Computing (HPC) strives to achieve the maximum performance of a given machine. The increasing complexity of computing hardware architectures nowadays, makes rise the number of variables to take into account to achieve this maximum performance and it is even worse when considering heterogeneous infrastructures as computational Grids. A common problem is the computation imbalance present in tightly-coupled applications that run in Grid infrastructures which is due to the unawareness of the underlying infrastructure characteristics. One of the best options to get the maximum performance is to tune the application code for a given architecture. This approach is used by ATLAS [129] which gets its speed by specializing itself for a given platform. Architecture aware tools such as hwloc [16] are now available in high performance runtime environments of parallel applications. Therefore, a deep knowledge of the underlying infrastructure by the application is the evident trend to achieve the best performance. For some regular scientific codes, it is possible to derive a performance model and the tuning of the application can be guided based on this performance model [61]. This performance model can be constructed either from a detailed understanding of the application execution or by analyzing multiple runs. A multiple-runs approach is simpler because it takes into account the complex interaction between the application and for instance the memory hierarchy. To do so, several tools

such as profilers, tracers, statistical engines, runtime environments have to be linked together in order to carry out the task of automating the generation, collection and treatment of performance information and provide the appropriate data to create the model.

In this chapter, it is shown how parallel applications can take advantage from experiment management tools. A technique of load balancing for large simulations codes based on a prediction model is analyzed. This technique relies on the interaction between experimental management tools and parallel applications. The technique is applied to a large electromagnetic simulation code based on Transmission-Line Matrix (TLM) numerical method [60], deployed in a heterogeneous Grid infrastructure. This technique is classified as a *Static* load balancing which is well adapted to highly regular applications. It requires few changes to the application code compared to adopting a new programming model and given the high memory requirements of the application, a dynamic approach would generate a considerable overhead. The used of our experiment management tool *Expo* presented in Chapter 3 is shown. This enabled us to manage the modeling workflow where the execution of big campaigns of application runs are needed and the orchestration of different tools that could participate in the process of creation of the performance model. Doing this task efficiently is important in order to not delay the execution of the real application, reduce the perturbation of the results and provide in a short period of time valuable information to the application.

The contribution of this chapter is twofold:

- Show the importance of experiment management tools in helping users to manage the complexity of distributed infrastructures, to automate several tasks and to make efficient use of computational resources.
- A load balancing technique for regular scientific codes based on the calibration of the platform and a prediction model. The approach is not expensive in terms of code source modification, user intervention and presents almost no overhead. An average improvement of 36% in the execution time is achieved.

4.2 Related work

The related work is organized into two parts: the load balancing techniques in parallel applications and the different techniques to carry out such a task. The second part presents the state of the art of experiment management tools and works related to the benchmarking of Grid platforms.

4.2.1 Load balancing of distributed applications

An important phase of the execution of parallel codes is the assignment of work to compute units. The problem of load balancing then is defined as the assignment of work to the compute units according to its performance or load. This assignment of work can occur at the startup of the application (static partitioning) or it can happen several times during the execution of the application (dynamic partitioning). Both of them will be described in the following subsections.

Dynamic techniques

Dynamic techniques are very popular now given the apparition of infrastructures such as cloud computing. It is the case of Charm++ runtime system [58] which through continuous estimation of processor load, it adapts to the imbalance created by known fluctuations in shared infrastructures. Another approach based on Charm++ [85] takes into account the latency existing in cross-site communications for Grid infrastructures. As it can be very cumbersome to convert applications to newer paradigms such as Charm++, AMPI was proposed in [13] which enables a bigger number of application benefits from the framework features as load balancing. These dynamic techniques were mainly created due to the large presence of high irregular load in parallel computational science and engineering. Our approach applies to highly regular codes executed on Grid infrastructures

where the CPU is not shared between users. Therefore, the gain obtained with a dynamic approach would be negligible and there exist a potential overhead of context switching and migration.

Static techniques

In [109], a static load balancing technique for mapping iterative algorithms onto heterogeneous clusters is presented focusing on the complexity of application partitioning and the efficient heuristics for the distribution schemes. Load balancing for Grid applications is proposed as well by PaGrid[64] which proposes a partitioner to balance mesh based applications. A graph is generated for the platform where processors are weighed according to its relative performance at executing standard benchmarks. This graph is matched with the graph generated for the application. In [40] is described a resource-aware partitioning where information about a computing environment is combined with traditional partitioning algorithms. The approach collects information about the computing environment and processes it for partitioning use.

4.2.2 Experiment management tools

GrapBench [94] provides a framework to carry out a semi-automatic benchmarking process for studying application behavior in grid infrastructures. The framework controls the number of benchmarking measurements required by a given application which are managed then by its experiment engine. The work outlined here differs from this in that it provides a more general experiment engine conceived to carry out any kind of study for an application in distributed platforms. Plush [4] is a widely used tool in PlanetLab, for deploying and monitoring application execution in distributed platforms. It provides abstractions to specify the steps to deploy an application, however, a real experiment entity is not taken into account. The inflexibility of its description language makes it difficult to write parametric studies. ZENTURIO [104] enables the management of parametric studies for an application in a framework for experimenting, but their high number of modules makes it difficult to port it to different platforms.

Workflows engines are well known for their capacity for carrying out parametric studies. Vistrails [23] provides parameter exploration and comparison of different results. It improves the experimentation activity providing data provenance tracking mechanisms. One limitation of Vistrails is its inability to adapt to distributed environments. Pegasus[38] offers a mapping between tasks in a workflow and distributed infrastructures (cloud, grid, clusters). Despite the capacity of some workflow engines to use distributed infrastructures, it is difficult to use them when considering the setup of an application. This setup could incur several complex steps that need a constant supervision. For more information about the aforementioned tools the reader is referred to Chapter 2. The approach proposed in this chapter addresses those issues and it is based on the experiment management tool presented in Chapter 3. In that chapter it was shown that *Expo* is based on two abstractions *resources* and *tasks* which can be combined to represent a workflow. The workflow specification describes all the experiment activity: platform access, application deployment and setup, application execution, analysis and generation of results.

4.2.3 Transmission-Line Matrix

The main idea of this application is to simulate the propagation of an electromagnetic field inside large structures such as tunnels and airplane cabins. TLM numerical method models the electromagnetic field propagation by filling the space with a network of transmission-lines fed by electrical signals whose voltage and current correspond to the electric and magnetic fields. The intersection of these lines, that have the free-space impedance, is modeled with the Symmetrical Condensed Node (SCN) [72] scheme, whose scattering matrix is derived directly from the behavior of the fields. The TLM method requires significant computing resources, but its algorithm has the advantage of being parallelizable, which makes it possible to simulate oversized structures on multiple computing machines. Using a parallel approach, large electromagnetic structures can be

modeled by means of large scale computing systems such as Grid or supercomputers in a HPC scenario.

In order to avoid a heavy TLM calculation, the discretized domain is sliced into several sub-domains that are assigned to the processors where will be computed in parallel. The CPUs communicate between them to achieve the job. The parallel approach, based on Message-Passing Interface (MPI), is designed for Single Program Multiple Data (SPMD) programming model as it is presented in [9]. In the proposed parallel TLM application, a one-dimension Cartesian topology is implemented for the partitioning process.

4.3 Load Balancing approach

Here, the technique of load balancing applied to the TLM application is described. Considering a fully heterogeneous infrastructure, such as Grid'5000, a Grid computing with many clusters geographically distributed composed of different hardware configurations. The application needs to assign an adequate workload for each node in order to fully exploit the infrastructure capacities. Given that the application is highly regular as shown in [9], a static load balancing technique is chosen, where all the work is divided and distributed at the beginning. The amount of work assigned to each processor depends on the relative performance of the application on such processor. As this relative performance can be difficult to get from processor characteristics, a prediction model is used in order to have a more accurate indicator. It was already shown that the expected runtime of the computation part of the application scales linearly with the number of TLM cells N_x, N_y, N_z on the three Cartesian directions, y being the partitioning direction. Thus, a simple linear function given in [9] is used to model the performance:

$$T_{calc} = c_1 + c_2 N_x N_y N_z t, \tag{4.1}$$

where $c_{1,2}$ are the time coefficients corresponding to different blocks of the TLM application and t represents the number of computing iterations. The prediction model, given in (4.3), takes into consideration the algorithm to be executed and the processor architecture performing the computation. They represent the processor architecture information inside the prediction model. This model takes into account the effects of cache misses, according to the problem size. The first term may be neglected as it is very small compared to the second one. Lets consider that the partitioning procedure gives the length of the computing sub-domain assigned to the process i, as:

$$l_i = \alpha_i N_y, \tag{4.2}$$

with

$$\sum_{i=1}^{p} \alpha_i = 1$$

for all p processes the structure is computed by. Consequently, the amount of work is distributed according to the fact that the computation time has to be the same for each process i:

$$T_{calc_i} = c_i N_x l_i N_z t, \forall i \in [1, p] \tag{4.3}$$

where c_i is the second coefficient from (4.3) corresponding to the process i. This leads to describe (4.2) by:

$$l_j = \frac{N_y}{c_j \sum_{i=1}^{p} \frac{1}{c_i}}, \tag{4.4}$$

where l_j is the work assigned to the process j. Therefore, a construction of a prediction model of the application for each different computing hardware available on the Grid infrastructure has to be performed. In order to have a good prediction model, a given set of chosen simulations have to be run and analyzed for each different machine. This process is depicted in Figure 4.1. *Expo*

is used to automate the task of conducting this big number of executions. This process will be called *calibration*. The module used to this end is described in Section 4.3.1. The load-balancing approach implemented in this work considers the communication between different clusters being homogeneous. The communication capabilities of the computing environment are not taken into account. Not all resources have to be involved especially when the structure to be computed is not so large, because the communications due to an excess of processors may slow down the entire simulation, despite the increased accumulated speed.

The execution of the application will be wrapped in two *Expo* modules, which will automate all the process in the platform chosen for testing (Grid'5000).

- Calibration of the platform. This module runs once, it can contact the platform in order to know if there has been a change in the hardware configuration and deploys the necessary calibration.
- Deployment of the application. Generation of a file that contains platform fitness information for the application and carry out the load balancing at application level.

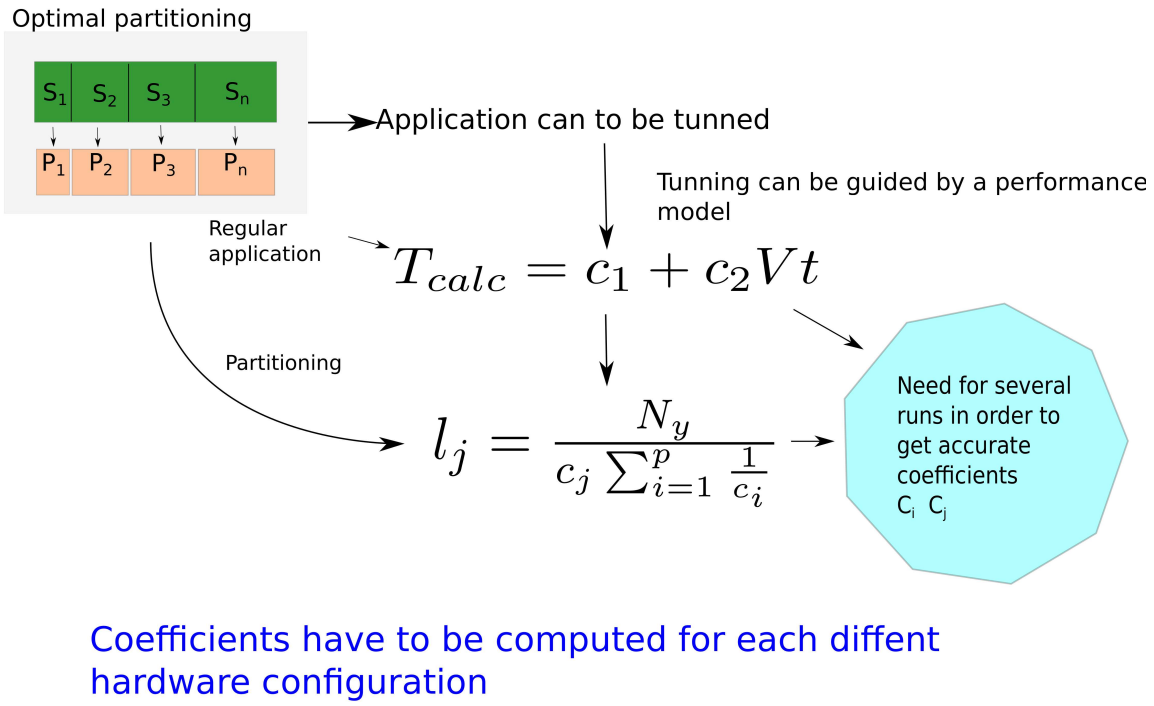

Figure 4.1: Load balancing approach

4.3.1 Expo calibration module

All the procedure of platform calibration was captured using *Expo* tasks abstractions. The following tasks were defined:

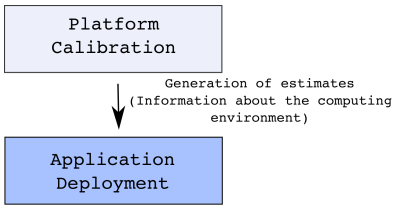

Figure 4.2: Expo Modules: the calibration modules is executed once

Task name	Execution time [sec] per cluster									
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
Transfert site	15.09	13.31	16.32	14.06	26.76	42.55	10.26	10.46	11.92	35.03
Compiling code	21.84	24.35	30.14	22.38	23.49	27.10	20.56	21.36	29.94	20.28
Calibration	1770.14	4860.31	3630.55	1770.47	4660.67	7590.81	1640.23	1600.83	3430.70	1620.87
Free resources	1.76	1.62	2.20	1.25	1.33	1.54	1.42	1.77	1.06	1.55

Table 4.1: Execution time of the different tasks that compose the calibration module.

- **Run reservation:** make a request to the computing platform in order to reserve the resources needed.
- **Transferring code to each site on the grid:** The code is sent from one chosen site to every site in Grid'5000.
- **Extracting and compiling the code:** The code is extracted and compiled with the right configuration.
- **Calibration:** It comprehends the execution of several simulations with different parameters. Two types of calibration are performed in order to take into account the cache effects.
- **Compute coefficients:** The statistical engine R[1] is used in order to process the files generated by the calibration and perform a linear regression in order to calculate the coefficients of the model.
- **Free resources:** It makes a request to the platform in order to free the resources used by the calibration.

These tasks were described using *Expo* DSL using 180 lines. An extract of the description is shown in Listing 11 and the different execution times of each task for different clusters are shown in Table 4.1. It is important to note that the time to execute the whole module for a particular cluster mainly depends on the execution time of the simulations. There is an almost negligible overhead in the execution time with *Expo*, which was already shown in Chapter 3.

In Figure 4.3 is shown the executable workflow generated from the abstract calibration experiment definition. Here, the level of execution is the job. The system submits a job into the infrastructure for every different (different architecture) cluster in Grid'5000. Thereby, every task defined in the abstract representation is mapped into a cluster and managed asynchronously. Several machines were used per cluster in order to lower the time to get the results. The simulation were deployed in parallel for this case using TakTuk which enable us to maintain a low number of ssh connections to control the experiment. In Figure 4.4, it is shown the heterogeneity of Grid'5000 in terms of coefficients of the prediction model. This figure was generated using the results obtained by the calibration module.

Advantages of using *Expo*:

- It helps to deploy efficiently the simulations used for the calibration part, making independent from the platform. More than 1359 simulations were necessary to get data for the prediction model.

[1] http://www.r-project.org/

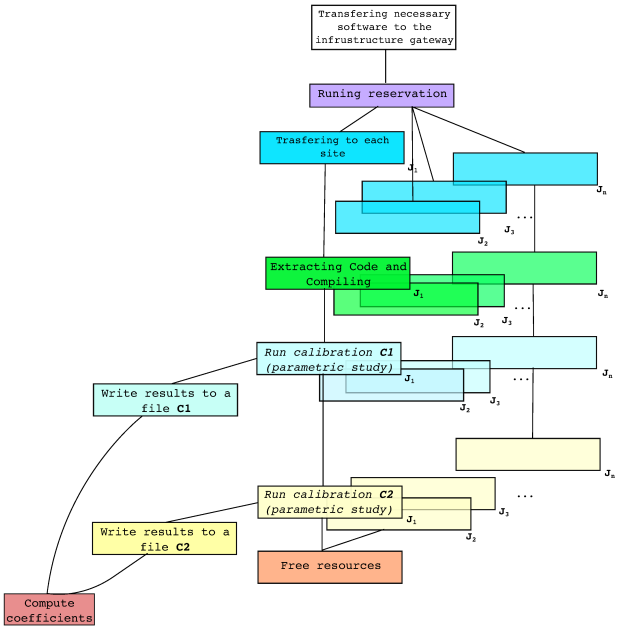

Figure 4.3: Experiment calibration executable workflow

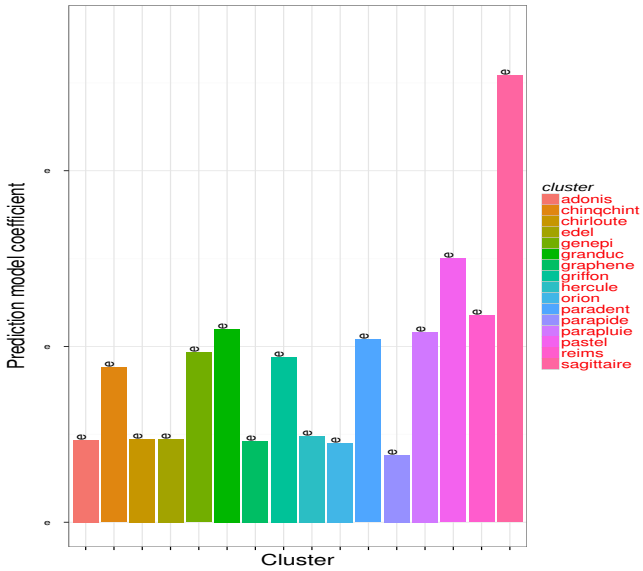

Figure 4.4: Heterogeneity of Grid'5000

- Makes all the procedure more reproducible and repeatable.

- Frees the application from implementing this functionality. Relying on more flexible languages for this task.

```
task :transfering_tlm, :target => resources.gw do
    put("~/TLM/tlm_v1.tar","/tmp/tlm_test.tar",:method => "scp")
end

task :run_reservation, :depends => [:transfering_tlm] do
  reserv.run!
end

task :transfert_site, :target => resources, :depends => [:run_reservation] do
  options_put = {:method => "scp", :nfs => :site}
  run("mkdir -p ~/Exp_tlm")
  put("/tmp/tlm_test.tar","~/Exp_tlm/tlm_test.tar",options_put)
end

task :compiling, :target => resources, :depends => [:transfert_site] do

  check("ls ~/Exp_tlm/TLMME/") then
    run("cd ~/Exp_tlm/; tar -xf tlm_test.tar")
    run("make -C ~/Exp_tlm/TLMME/tlm/")
  end

end

task :calibration_c2, :target => resources, :depends => [:compiling] do

 params_c2.each_with_index{ |par,index|
    number_sim = 2
    RUNS.times do
      tag = {:parameters => par,:size => size_c2[index] }
      commands =["cd ~/Exp_tlm/TLMME/tlm/;./run 1 #{par} matched"]
      run(commands, :ins_per_machine => number_sim,:log => tag)
    end
    puts "Finishing parameter #{par}"
  }
end
```

Listing 11: Extract of the calibration module

4.4 Results

4.4.1 Experimental platform

The simulations were performed on Grid'5000 platform [55]. For performance reasons, only two processes are executed on grid nodes, each one on a different processor. The architectures of the computing nodes from Grid'5000 are different from cluster to cluster. The same clusters where used in order to keep the homogeneity between the experiment results concerning the simulation time. These clusters are geographically distributed in two sites. These sites are connected by RENATER, the French network for research and teaching. All *Expo* description files used two run the experiments are available in[2].

4.4.2 Using different configurations

Here, it was evaluated the performance gain obtained using load balance under different hardware configurations. In order to show the improvement in performance for large simulations, we opted for using different simulation sizes proportional to the number of nodes. This enabled to maintain a favorable rate between computation and communication. The results are shown in the Figure. 4.5. A maximum gain of 42.84% was obtained using clusters located in the same site. The gain obtained using several geographically distributed sites varies a great deal, we observed here performance gains ranging from 3.25% to 19.92%.

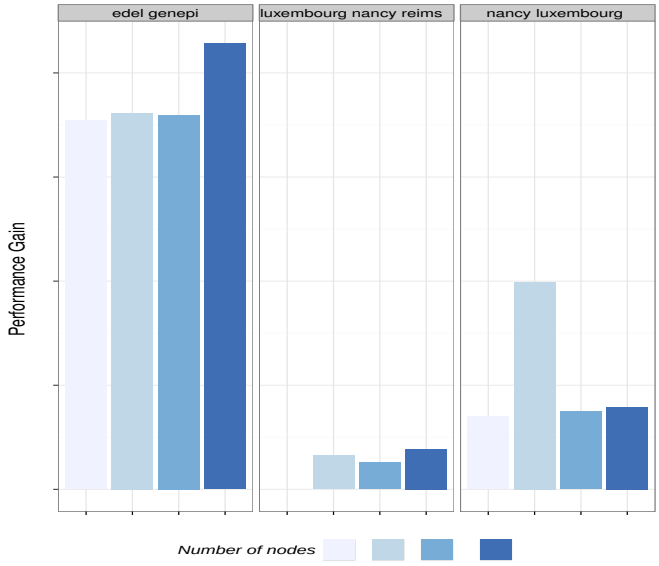

Figure 4.5: Using different heterogeneous configurations. First tests used cluster located in the same site (*edel-genepi*). The other two series of test used different geographically distributed sites (*luxembourg, nancy, reims*).

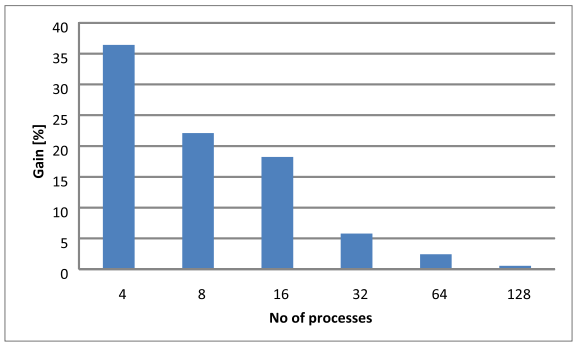

Figure 4.6: Gain obtained with the same simulation parameters changing the number of nodes.

4.4.3 Changing the number of nodes

The experiment simulates the electromagnetic field propagation, using the TLM method, for 10000 time steps inside a waveguide structure, having the dimensions: 172 mm width, 86 mm height, 2432 mm length, a mesh step of 1 mm. In this experiment the computing nodes belong to Griffon, Chinqchint and Chirloute clusters. The simulation time values are presented in Figure. 4.6. The maximum gain obtained when using load-balancing approach is about 36%. The values of the simulation time when the load is balanced according to the calibration model given by *Expo* are smaller than the time values when the structure is divided identically on all MPI processes. The gain obtained by load balance approach decreases while the number of processes increases, because the computation time decreases according to communication time.

[2] http://expo.gforge.inria.fr/

4.4.4 Large structure

In order to prove the real benefits of the grid environment for TLM large simulations, a supersized rectangular matched waveguide, discretized upon 95 million TLM cells is simulated. Its dimensions are: 345 mm width, 173 mm height, 1600 mm length and a mesh step of 1 mm.

Distributed experiment

In the first experiment, the simulations are performed using four nodes from Griffon and Chirloute clusters. The gain obtained by load balancing approach is about 25.5%.

Local experiment

A second experiment was carried out using nodes from clusters Paradent and Parapide which are localized on the same site. The gain obtained by load balancing approach is about 48.5%, much better than the distributed experiment because the communication time is much smaller between nodes on the same site.

4.5 Conclusions and Future Works

This work showed the interaction between applications and experiment management tools, which is not limited to reproducibility purposes and replayability of experiments. This calibration is an example of how experiment management tools can free applications of doing certain tasks and how can they help them to perform a tuning for a given platform. The use of tools as *Expo* serves the following purposes: it makes easy the access to complex platforms, helping non-expert users to make an efficient use of the resources. It helps to combine tools in order to capture the experimenting process.

It is difficult to perform an efficient deployment of the application using just information provided by the hardware. Performance models based on runs provide a more accurate information for using the platform resources more efficiently. At the same time, a load balancing based on a performance model gives to the application high flexibility for estimating the best work placing for a certain size given the hardware configuration.

In perspective, smarter reservation mechanisms taking into account the calibration and the availability of the platform, the different number of possible configurations for deploying and their cost represent a viable solution toward fast and automatic multidisciplinary application simulations.

Part III

Kameleon

Chapter 5

Setting up complex software stacks

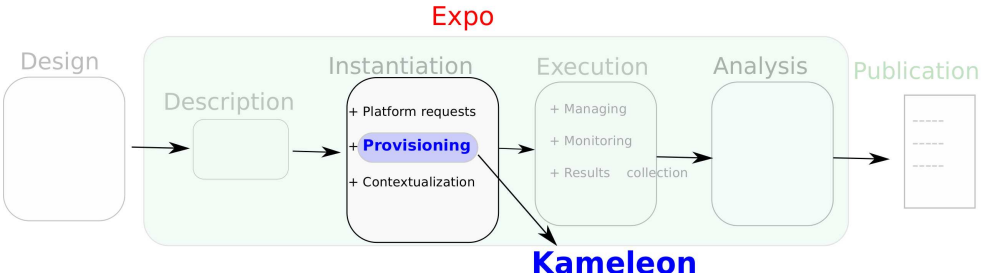

Figure 5.1: Role of *Kameleon* in the experiment cycle

A *software appliance builder* bundles together an application with its needed middleware and an operating system to allow easy deployment on Infrastructure as a Service (IaaS) providers. These builders have the potential to address a key need in our community: the ability to reproduce an experiment. This chapter reports the experiences on developing a software appliance builder called *Kameleon* that leverages popular and well tested tools. *Kameleon* simplifies the creation of complex software appliances that are targeted at research on operating systems, HPC and distributed computing. It does so by proposing a highly modular description format that encourages collaboration and reuse of procedures. Moreover, it provides debugging mechanisms for improving experimenter's productivity. To justify that our appliance builder stands above others, we compare it with the most known tools used by developers and researchers to automate the construction of software environments for virtual machines and IaaS infrastructures. The results shown in this chapter were published in [111].

5.1 Introduction

Thanks to the advances in virtualization, the lowering of the cost of computing hardware and the increasing popularity of cloud computing. Now software infrastructures can be deployed easily and applications can be bundled together with their middleware requirements and operating system in what is called a *software appliance*. Two use cases for software appliances in industry and research are:

- *Industry*: the pervasiveness of cloud computing makes feasible the replacement of a whole software stack from scratch instead of trying to fix it. This has led to a new model of provision software based on *software appliances* [28], which is also known as *Immutable servers*. This brings several advantages to IT administration as: faster deployment time, all the dependencies are already satisfied, it is easy to have a production like environment

on the development machines, etc. Hence, approaches like: *vagrant* [1], *veewee* [2], *packer* [3], *docker* [4] have gained wide acceptance in industry. Those approaches strive for having a common reproducible and disposable software environment that can be rebuilt from scratch or from a base image using a definition file that can be versioned.

- *Research*: Large testbed infrastructures for experimentation in networks and large scale systems such as Grid'5000 [25], FutureGrid [51], etc. are available, which enable the deployment of complex software stacks either on bare metal or using an IaaS provider. These infrastructures' high degree of software stack customizability appeal to researchers who want to test their ideas in real settings. However, the management of these software stacks is not always trivial, their setup is a tedious and time consuming task that should be automated whenever possible. The lack of automation can be attributed to the low expertise, lack of the proper tools and the long learning path for researchers. The lack of automation leads to the inability to reproduce an experiment, since it is not even possible to build or set the experimental setup under the exact same conditions where an experiment took place. A recent study [30], where the buildability of artifacts was evaluated, found that only 24% of publications in ACM conferences and journals can be built. To preserve the experimental setup some works are relying on *software appliances* technology.

Therefore, it is evident the importance and benefits of *software appliances* for both industry and research. This chapter focus more on the latter use of *software appliances* that deals with the problematic of experimentation under real settings in computer science.

5.1.1 Motivations

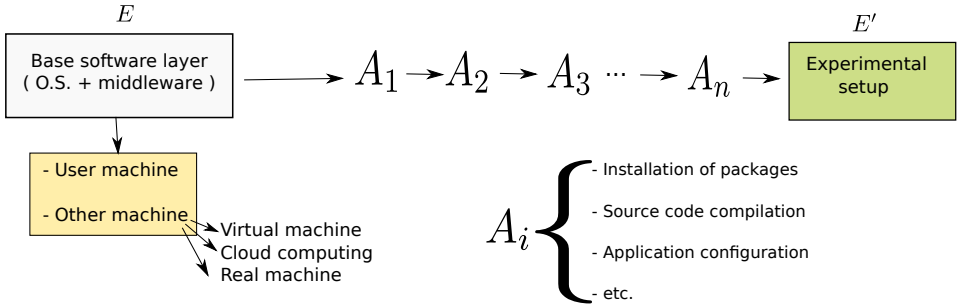

Figure 5.2: Creation process of an experimental setup.

Figure 5.2 illustrates the process to derive an experimental setup. Experimenters start from a base setup which includes an operating system plus a middleware. This base setup could be located in the same machine of the experimenter, in a virtual machine, in an IaaS provider as Amazon EC2[5], OpenStack[6], etc; or in a real machine that belongs to a computing cluster. The experimenter will apply a sequence of actions $\langle A_i \rangle$ which consists in, for instance: installation of software packages, source code compilation, software configuration, etc. Applying these actions $\langle A_i \rangle$ produce an experimental setup E', which is then used for the evaluation of a given implementation, algorithm, etc. Due to space limitations in research papers the composition of E' is not properly described, nor are the sequence of actions $\langle A_i \rangle$ that were taken to derive E'. In domains such as High Performance Computing, Distributed Systems and Operating Systems

[1] http://www.vagrantup.com/
[2] https://github.com/jedi4ever/veewee
[3] http://www.packer.io/
[4] https://www.docker.io/
[5] http://aws.amazon.com/ec2/
[6] http://www.openstack.org/

research, experimental setup configuration, which includes the operating system, version of libraries and compilers, compilation flags, etc, are crucial requirements to be able to repeat an experiment [26].

5.1.2 Reconstruct-ability

To improve experimentation, we claim that an experimenter needs to know the exact process that led to the creation of a particular experimental setup, E', as well as to be able to replay and modify this process to arrive at the same and alternative experimental setups. We introduce the concept of reconstructability of an experimental setup to formally capture this process. An experimental setup E' is reconstructable if the following three facts hold:

- Experimenters have access to the original base experimental setup E.
- Experimenters know exactly the sequence of actions $\langle A_1, A_2, A_3, ..., A_n \rangle$ that produced E'.
- Experimenters are able to change some action A_i and successfully re-construct an experimental setup E''.

Reconstructability can be expressed functionally as $E' = f(E, \langle A_i \rangle)$, where f applies $\langle A_i \rangle$ to E to derive the experimental setup E'. Thus, if reconstructability holds, we are guaranteed to be able to derive E' no matter when $\langle A_i \rangle$ is applied to E. Reconstructability does not hold when:

- An action A_i is composed of sub-tasks that are executed concurrently making the process not deterministic. For example: compilation of software using `Makefiles` with the option `-j` that runs parallel compilation process. This provokes compilation rules to run in any order if they are not connected by dependencies.
- Packages with the latest release of Debian (*Debian 8*) have a time of expiration. Therefore, old packages can not be installed.

Reconstructability also does not hold when either the base setup, E, or the specific software used in an action, A_i, is no longer available. The availability of software becomes an issue when reconstructability depends on package managers and configuration management tools [42]. For example, there is no guarantee that a git repository which is used by an action will be available at a later point in time.

5.1.3 Contributions of this chapter

This chapter identifies the necessary ingredients for a software appliance builder to be a viable solution for the preservation and packaging of experimental setups. The contributions of this chapter are:

1. In Section 5.1.2, we introduced the concept of reconstructability, which identifies the process to build an experimental setup so that the setup can be rebuilt and can be built with variations.
2. In Section 5.3, we evaluate existing software appliance builders against the criteria needed to improve user productivity.
3. In Section 5.4, we refine the *Kameleon* syntax and concepts, and we extend the persistent cache mechanism so that it supports new concepts.
4. In Section 5.5, we demonstrate that *Kameleon* is modular, enables the reuse of code, and builds on proven technology.
5. Section 5.5.2, we identify the container requirements for different types of software appliances.

The rest of this chapter is structured as follows: Section 6.2 presents related work. Section 5.3 presents a qualitative comparison of the most widely used software appliance builders. Section 5.4 presents a complete description of *Kameleon* architecture, concepts and features. Section 5.5 presents use cases that validate our approach. Section 5.5.4 presents future work. Section 5.5.5 concludes.

5.2 Related work

We use the term software appliance, which is defined as a pre-built software that is combined with just enough operating system *(jeOS)* and can run on bare metal (real hardware) or inside a hypervisor. A virtual appliance is a type of software appliance, which is packed in a format that targets a specific platform (normally virtualization platform). A software appliance encompasses three layers:

- **Operating System:** In the broadest sense includes the most popular operating systems (e.g GNU/Linux, Windows, FreeBSD). This element of the appliance can also contain modifications and special configurations, for instance a modified kernel.
- **Platform Software:** This encompasses compiled languages such as C, C++ and interpreted languages such as Python and Ruby. Additionally, applications or middle-ware (e.g., MPI, MySQL, Hadoop, Apache, etc.). All Those software components are already configured.
- **Application Software:** New software or modifications to be tested and studied.

Virtual appliances bring up numerous benefits to administration of big infrastructures [114] and education on operating systems [86]. A system for deploying lightweight virtual appliances was proposed in [28] which is based on COW-based virtual block disks for splitting a virtual disk image into smaller disk images for rapid deployment of requested services. A similar system was proposed in [117] based on virtual machine snapshots with the goal of improving response time of cloud computing infrastructures. The feasibility, of using *virtual appliances* for service deployment, was shown in [119]. The approach resulted easy and simple compared to traditional deployment mechanisms. A system called Strata proposed in [96] enables more efficient creation, provisioning and management of *virtual appliances*. Another system called *Typical Virtual Appliances* is proposed in [133] which brings more flexibility to service deployment, consuming a few storage and bandwidth.

Re-running an experiment with the original software artifacts could be achieved by using virtual appliances and virtual machine snapshots [63, 45]. Brammer et. al [14] present a system to create executable papers, which relies on the use of virtual machines and aims at improving the interactions between authors, reviewers and readers with reproducilibity purposes. *Kameleon* differs in that it allows the re-execution of an experiment with the original software artifacts and the ability to modify the experimental setup cleanly and easily.

Widely used tools such as Vagrant, provide reproducible environments for development. Vagrant uses pre-built images which hinders understanding of the operating system layer and makes modifications to this layer difficult. *Kameleon* differs in that the construction of the operating system layer is part of the software appliance generation. This fact makes its recipes less complex than the recipes used by popular configuration management tools such as Puppet[7] and Chef[8].

From the traceability point of view, *Kameleon* can be compared to interactive notebooks such as IPython[9] where the goal is to track every step that leads to a given result. *Kameleon* keeps a trace of all the steps that led to the creation of a given software stack, it does so by providing a structured, modular and understandable language. *Kameleon* makes reconstructability of software appliances possible, experimenters are able to explore all the actions, modify and repeat the environment generation.

[7] `http://puppetlabs.com/`
[8] `https://www.getchef.com/chef/`
[9] `http://ipython.org/notebook.html`

In Section 5.3.3, we discuss software appliance builders.

5.3 Software appliance builders comparison

We describe and evaluate the most widely used software appliance builders in cloud infrastructures and development environments. The evaluation uses as criteria: 1) how well they support the software appliance build cycle and 2) whether they meet the criteria for improving experimenters' productivity to build an experimental setup.

5.3.1 Software Appliance Build Cycle

All the analyzed tools follow the same pattern in the process of building a software appliance. The tool takes as input a *Description File* that details all the requirements that the software appliance should meet. Then, it initializes a *Container*. A container is the environment that it is used for building the software appliance. This term container encompasses: system level virtualization techniques (e.g., chroot, openVZ, Linux Containers), full virtualization technologies (e.g., VirtualBox, KVM, Xen, VMware) and real physical machines. Once the container is initialized, the tool parses the description and starts to carry out the *bootstrap*, *setup* and *export* procedures. The output of this process is a software appliance formatted for the infrastructure that will finally host it. Table 5.1 shows how this build cycle is supported by each tool. The main steps in the software appliance build cycle are explained below:

- **Bootstrap**: This refers to the process of getting a bootable operating system. This bootable image can be either built from scratch or it can be retrieved from some external source. The normal procedure is to get an ISO image from the target operating system and follow the installation procedure. Another option is to download and load a software appliance already created.
- **Setup**: In this step, users apply several procedures to customize the base system and make it meet their needs. These procedures include mainly the installation and configuration of software. There are many possible ways to customize, by using shell scripts or configuration management tools such as Salt, Chef, Puppet, Ansible, etc.
- **Export**: This step creates the final format for the software appliance. The final format ranges form the available virtual disk formats (e.g., VDI[10], VMDK[11] ,QCOW2[12]) to more simple formats based on *tarballs*[13].

5.3.2 Criteria for Improving User Productivity

The evaluation is driven by the question: *What makes an experimenter more productive when building a complex software appliance?* The following criteria will be used for the evaluation:

- **Easiness**: The tool has a low learning curve. Specially, a low learning curve is supported by providing a simple language to describe the appliance across the different levels of the software appliance's software stack (e.g., O.S. level, middleware or application).
- **Support during the build process**: Long compilation times are commonplace when building these kinds of software stacks, for instance the compilation of operating system kernels, modules, scientific libraries. Because this process is frequently error prone, a mechanism for debugging or checkpointing the process makes the experimenter more productive. Validation of the correct functioning of the software appliance is required as well.

[10] `https://www.virtualbox.org/manual/ch05.html`
[11] `http://www.vmware.com/app/vmdk/?src=vmdk`
[12] `http://www.linux-kvm.org/page/Qcow2`
[13] It refers to a computer file format that can combine multiple files into a single file.

Feature \ Tool		**Docker**	**Packer**	**OZ**	**Veewee**	**Kameleon**	**BoxGrinder**
Building	Bootstrap	Read only tarballs that can be obtaind form *Docker Hub*	Installation ISO, existing software appliance	Installation ISO	Installation ISO	Any bootstrap option	Installation ISO
	Setup	DockerFile instructions	Shell scripts, File upload, Ansible, Chef, Puppet, Salt	Shell scripts	Shell scripts	Shell scripts with Kameleon syntax	Shell scripts
	Export	Linux Containers	Amazon EC2, DigitalOCean, Docker, Google Compute Engine, OpenStack, Parallels, QEMU, VirtualBox, VMware	QEMU	VirtualBox, QEMU, VMware	VirtualBox, QEMU, VMware, Docker, Grid'5000	Amazon EC2, QEMU, Virtualbox, VMware
Description	Language	Plain text docker language	JSON	XML	Ruby	YAML	YAML
Execution	Container support	Linux containers	Same as Export	QEMU	Same as Export	Same as Export	guestfs
	User facilities	Able to commit changes in the File system layer	Validation of description, ISO caching	ISO caching , generation of metadata manifest	Image configuration validation	Persistent cache mechanism, checkpoints, interactive shell	None

Table 5.1: This table shows how the software appliance build cycle is supported by each tool

Table 5.2: Comparison of widely used appliance builders based on criteria that would make an experimenter more productive.

Tool	**Kameleon**	**Docker**	**Packer**	**BoxGrinder**	**Veewee**	**Oz**
Easiness	Yes	Yes	No	Yes	No	No
Support in the building process	Yes	Yes	Yes	No	No	No
Container diversity	Yes	No	Yes	No	Yes	No
Shareability	Yes	Yes	No	Yes	No	No
Reconstructability	Yes	Yes	No	No	No	Yes

- **Containers diversity**: The tool should support a variety of container types. This enables hassle-free transportation of an experimental setup from one infrastructure to another, because experimenters are more comfortable with working in specific environments. Additionally, it should be easy to integrate new types of containers that meet the requirements of the experimenter. For example, libraries such as ATLAS[14] which gets its speed by specializing itself for the underlying architecture, needs to be compiled on the target machine where it will finally run. Certain Linux modules need direct access to real hardware. Therefore, they could not run on virtualize systems. That is the case for Dune [12] and CControl [101].
- **Shareability**: Instructions for building a software appliance must be organized and stored in a modular way to enable the reuse of procedures and collaborate within a community.
- **Reconstructability**: One important shortcoming is the reproduciblity of experiments in computer science. It has been demonstrated that one of the causes is the impossibility to build the same software artifacts[15] used in a publication [30]. Thus a requirement is to be able to reconstruct a software appliance from its definitions, which will at the same time enable later customization as defined in Section 5.1.2.

5.3.3 Software Appliance Builders

In this section, we describe and evaluate the most widely used software appliance builders according to our criteria for improving user productivity. Table 5.2 shows the evaluation.

Docker

Docker[16] offers a powerful and lightweight way to build software appliances that are packed in Linux Containers (LXC). Docker manages and tracks changes and dependencies, making it easier for users to understand how the final appliance was built. It relies on repositories for enabling users to share their artifacts with other collaborators. The most appealing feature of Docker is that it makes applications portable across many infrastructures. As a downside, however, applications are run inside Linux Containers which could be not suitable for certain uses (e.g., run an application that uses *cgroups*[17]). The description of the building process is done using a simple syntax based on few constructs that help customize the containers.

Packer

Packer[18] helps users to create identical software appliances targeted at multiple platforms. The process is composed of: builders, responsible for creating machines and generating images from them for various platforms; provisioners, used to install and configure software (many options are

[14] `http://math-atlas.sourceforge.net/`
[15] It refers to source code compiled for testing.
[16] `https://www.docker.io/`
[17] `https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt`
[18] `http://www.packer.io/`

available from simple shell scripts to high-end configuration management tools) and postprocessors, that help manage the final produced image. Packer supports a variety of container types and it strives to make descriptions portable across different containers. Thus the burden of changing from one development environment to another is reduced. However, a different language is used to describe the operating system layer, which makes difficult to add modifications to this layer. Additionally, the tool do not provide any mechanism for organizing the instructions which hampers *shareability*.

BoxGrinder

BoxGrinder[19] creates appliances from simple plain text descriptions for various platforms. It utilizes the host system to perform the image creation using the *guestfs*[20] library which results in a faster process. Then, the newly created software appliance can be exported locally to be used for a virtualization technology or it can be moved outside to be used in IaaS providers. Software appliance descriptions are simple and easy to understand and can be composed for reuse. BoxGrinder does not offer any mechanism for supporting the build process and it is tied to build the software appliance using the host system which could be problematic when some isolation is needed.

Veewee

Veewee[21] is a tool for automating the creation of custom virtual machine images. It is able to interact with several virtual machine hypervisors. It offers to the user the possibility of validating the generated software appliance through the execution of behavioral tests. The capacities of the tool for customizing a software appliance are very limited. Description files are written in Ruby restricting the interaction with shell scripts.

OZ

Oz[22] was created to ease the automatic installation of operating systems. It uses QEMU as a container and uses the native operating system tools to install software. The cycle of building a software appliance includes the generation of metadata about the packages installed. Software appliances are created using an XML-based language. Even though the language allows almost any operation of customization, the descriptions rapidly become complex and difficult to maintain.

Kameleon

Kameleon achieves easiness by proposing a structured language based on few constructs and which relies on shell commands. The hierarchical structure of recipes and the extend mechanism allow shareability. *Kameleon* supports the build process by providing debugging mechanisms such as interactive shell sessions, break-points and checkpointing. Containers diversity is achieved by allowing the easy integration of new containers using the same language for the recipes. Furthermore, persistent cache makes possible reconstructability. In Section 5.4, we present *Kameleon* in detailed.

5.3.4 Discussion

We found that many software appliance builders rely on archive files (e.g. ISO images) to bootstrap a software appliance. However, if the archive files is no longer available in a repository, then reconstructability is impossible. We found that 30% of Veewee definition files[23] point to

[19] `http://boxgrinder.org/`
[20] `http://libguestfs.org/`
[21] `https://github.com/jedi4ever/veewee`
[22] `http://www.aeolusproject.org/oz.html`
[23] This was tested with the version of veewee 0.3.7 by trying to build all templates during the period of 02/12/2013 and 20/12/2013.

repositories that either no longer exist or have some packages missing. Furthermore, management of containers is implemented either in the core of the tool or as plugins. This makes integration of new containers for non-advanced users difficult. Most of the tools support a wide variety of containers, however, because they are tied to virtualization, real hardware is not taken into account. Shareability which implies modularity and collaboration is not available. Docker is the only tool, at the moment, which implements a collaborative model for building software appliances. These tools do not support debugging or check pointing in the build process.

Finally, the way tools support the build cycle has an important impact on the recontructability given that some actions would be out of the user's control. When the language used in the tool's *Description file* is based on less human-readable languages, such as XML, or on complex recipes, such as the ones used by Chef and Puppet, that tool ranks lower in the easiness criteria.

5.4 Kameleon: the mindful appliance builder

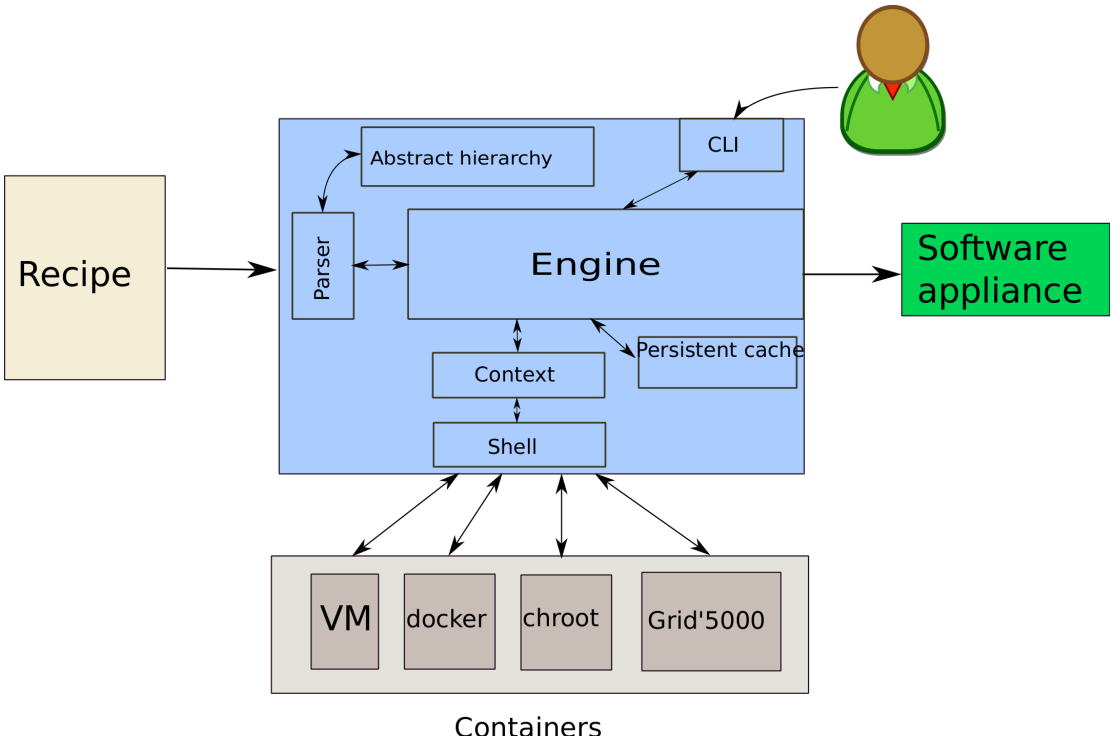

Figure 5.3: Kameleon architecture.

Kameleon is a small and flexible software appliance builder, which eases the construction and reconstruction of custom software stacks for research in HPC, Grid or Cloud computing and Distributed Systems. *Kameleon* version 2.2.4 is written in 2278 lines of Ruby[24] and has few dependencies. *Kameleon* achieves ease of use by structuring the specification (recipes) for the construction of software appliances into a hierarchy. The hierarchy's structure is composed of sections that allow a separation of customization and low level tasks. This structure separates out the customization tasks that can be easily performed by non-expert users from the low level tasks, such as setting up a complete operating system or exporting the whole file system, which are more difficult. These sections are divided into steps that represent actions $\langle A_i \rangle$ such as: installation and configuration of a certain scientific library, kernel patching, configuration of a base system. Steps are composed of microsteps that enable the customization and re-utilization of the same step

[24] Measured with SLOCCount `http://www.dwheeler.com/sloccount/`

in different recipes. Finally, the last level of the hierarchy wraps shell commands and *Kameleon* defined commands. All the aforementioned hierarchy is written using YAML, which encourages more human readable shell scripts[25].

An advantage of *Kameleon*, and what distinguished it from the existing appliance builders, is that it serves simply as a recipe parser and orchestrator of shell commands, which means that all the logic for the creation of a software appliance resides entirely in the recipes. *Kameleon* recipes enable four advantages for experimenters: 1) it helps to understand how the software appliance was created (all the details are embedded in the same language); 2) it gives a total control over the whole process, which reduces the burden of integrating new containers, new operating systems, or new export formats; 3) it enables the easy customization of software appliances at any level (e.g. O.S., middleware, applications, etc.); 4) it encourages a collaboration model where researchers can reuse code and given that all details are in the hierarchy of recipes and steps (text files) they can be easily versioned.

Figure 5.3 shows the architecture of the system and the interaction between the different modules. First, the parser, with the help of the abstract hierarchy, parses the recipe and creates as output the internal data structures that are input to the engine module. The engine orchestrates the workflow of execution. The workflow is executed sequentially. The context module helps to abstract the access to a given container. All the low level operations (e.g., execution of shell commands, I/O and file management) are performed by the shell module. The engine integrates three important mechanism for debugging: checkpoints, breakpoints and interactive shell sessions. The persistent cache captures all the data used during the process of building a software appliance, which is archived to allow the software appliance to be reconstructed at a later time. Finally, the CLI module implements the user interface.

5.4.1 Syntax

Figure 5.4 shows an example of a *Kameleon* recipe. We can highlight three different elements: sections, steps and variables. Four sections are proposed by *Kameleon* but more can be created. One section, called `global`, is dedicated to the declaration of global variables that can be used through out the recipe. The other sections correspond to the main steps in the software appliance build cycle (bootstrap, setup and export). Different sections in a *Kameleon* recipe allow a high degree of customizability, reuse of code, and total control of software appliance creation process by the experimenter. In Figure 5.4, the based system is built from scratch using the package manager of the Debian distribution as specified in the `bootstrap` section.

Alternatively, it is possible to use existing images (e.g., Grid'5000 base environments, cloud images for different Linux distributions, or software appliances market places[26]). The `setup` section installs packages, configures the O.S., etc. Within a section, users can execute shell commands, read and write files, or perform other commands that are necessary to carry out the desired customization. The options in the `export` section depend on the disk formats that the container supports. At the moment we have implemented recipes for exporting to the most popular virtual disk formats, tarballs and specific Grid'5000 format.

Listing 12 shows the definition of a step file. Each step file is loaded automatically by *Kameleon* after parsing the recipe. A step is divided into microsteps (e.g., `create_group`) which are in turn divided into commands. The goal of dividing steps into microsteps is the possibility of activating certain actions within a step. For example, from Listing 12 we have the possibility of executing only the microstep `create_group` without executing the rest of the microsteps. There are two types of variables: user defined variables that are provided in the recipe such as: Linux distribution (`distrib`), architecture (`kernel_arch`), etc., and *Kameleon* variables such as `$$kameleon_cwd` (*Kameleon* work directory) that interact with the engine. Contexts are mapped to special variables (`out_context` and `in_context`) in the global section. They indicate the necessary actions to set a shell in the respective context (the concept of context is explained in the next section). In the

[25] `http://yaml.org/spec/1.2/spec.pdf`
[26] `http://www.turnkeylinux.org`

```
global:
  ## User varibales : used by the recipe
  user_name: kameleon
  user_password: $$user_name
  # Distribution
  distrib: debian
  release: wheezy
  kernel_arch: $$arch
  hostname: kameleon-$$distrib
  ## Disk options
  nbd_device: /dev/nbd1
  image_disk: $$kameleon_cwd/base_$$kameleon_recipe_name.qcow2
  image_size: 10G
  filesystem_type: ext4
  # rootfs options
  rootfs: $$kameleon_cwd/rootfs

out_context:
  cmd: bash
  workdir: $$kameleon_cwd
  proxy_cache: 127.0.0.1

in_context:
  cmd: USER=root chroot $$kameleon_cwd/rootfs bash
  workdir: /root/kameleon_workdir
  proxy_cache: 127.0.0.1

bootstrap:
  - initialize_disk_chroot
  - debootstrap:
    - repository: http://ftp.debian.org/debian/
  - start_chroot

setup:
  - install_software:
    - packages: >
      debian-keyring sudo less vim acpid linux-image-$$kernel_arch
  - configure_kernel
  - install_bootloader
  - configure_network
  - create_group:
    - name: admin
  - create_user:
    - name: $$user_name
    - groups: sudo admin
    - password: $$user_password
export:
  - qemu_save_appliance:
    - input: $$image_disk
    - output: $$kameleon_cwd/$$kameleon_recipe_name
    - save_as_qcow2
# - save_as_vdi
```


Figure 5.4: In the example, the section headers illustrate contexts (`out_context` and `in_context`), declarations (`global`) and sections (`bootstrap`, `setup` and `export`). This example uses a chroot jail as a container for building a software appliance based on Debian Wheezy.

example, the recipe creates a Debian Wheezy appliance with some base configuration, which is specified as the `distrib` and `release` variables in the global section, and exports the appliance in QCOW2 format, which is specified in the export section as the step "`- save_as-qcow2`". The *Kameleon* recipe illustrates that sections are composed of steps that can be customized using variables. Table 5.3 illustrates `exec_*` commands, which are the minimal building blocks of microsteps. An `exec_*` command wraps a shell command to add error handling and interactiveness in case of a problem.

```
# Create User
- create_group:
  - exec_in: groupadd $$group

- add_user:
  - exec_in: useradd --create-home -s /bin/bash $$name
  - exec_in: adduser $$name $$group
  - exec_in: echo -n '$$name:$$password' | chpasswd
  - on_export_init:
    - exec_in: chown '$$user_name:' -R /home/$$user_name

- add_group_to_sudoers:
  - append_in:
    - /etc/sudoers
    - |
      %admin ALL=(ALL:ALL) ALL
```

Listing 12: Example of a step file. The prefix '$$' is used for variables.

Exec: executes a command in a given context	`- exec_in: echo "Hello!" > hello.txt` `- exec_in: apt-get -y update`

Pipe: it works as Unix pipelines but between contexts	`- pipe:`
	`  - exec_out: cat tlm_code.tar`
	`  - exec_in: cat > ./tlm_code.tar`

Write: allows to write a file in a context	`- write_in:`
	`  - /root/.ssh/config`
	`  - |`
	`  Host *`
	`  StrictHostKeyChecking no`
	`  UserKnownHostsFile=/dev/null`

Hooks: defers some initialization or clean actions.	`- on_setup_clean:` `  - exec_in: rm -rf /tmp/mytemp`

Table 5.3: *Kameleon* commands.

5.4.2 Kameleon Contexts

By dividing the building process into independent parts, contexts provide a way for a user to structure the software appliance creation process so that it is independent from the final target platform. When an appliance is built with *Kameleon* it is necessary to deal with 3 different contexts (more can be defined if required). The objective of all these contexts is to have a contextualized shell session. Contexts are as follows:

- *Local context*: It refers to the location where *Kameleon* is executed. Normally, it is the user's machine.
- *OUT context*: It is where the process of bootstraping will take place. Some procedures have to be carried out in order to create the place where the software appliance is built (IN context). This could be: the same user's machine using *chroot*. Thus, this context is where the setup of the chroot takes place. Other examples of *OUT context* are: setting up a virtual machine, access to an infrastructure in order to get an instance and be able to deploy, setting

Section	Context used	Description
Bootstrap	Local context and OUT context	Two possibilities: (1) build a file system layout form scratch. (2) start form an already created software appliance.
Setup	Mostly IN context	The commands run on the chosen container: chroot, Docker, Linux container, virtual machine and real machine
Export	Local context and OUT context	Use of the container supported tools for creating the final format for the software appliance.

Table 5.4: *Kameleon* concepts, interrelation between contexts and sections.

up a Docker container. This context also allows the appliance's base file system layout to be setup.

- *IN context*: It makes reference to inside the container created by the *OUT context*. This context can be mapped to a *chroot*, virtual machine, physical machine, Linux container, etc. This context is frequently used for customizing the software appliance.

The relation between the possible contexts used and the section execution is shown in Table 5.4.

5.4.3 Checkpoint mechanism

The construction of a software appliance is a trial and error process. *Kameleon* provides a modular checkpoint mechanism that saves time when debugging the software appliance construction process. Time consuming tasks such as the installation of an operating system from scratch are not repeated during the debugging process. Thus, a checkpoint mechanism encourages the automation of software appliance building as it makes the construction of software appliances less time consuming. We have integrated different checkpointing mechanisms for each container supported by *Kameleon*. They are based on snapshots of virtual machines (QEMU, VirtualBox) and based on snapshots of *QCOW2* disk images for the chroot container. Another checkpoint mechanism use *Docker* commits to preserve the state of a Docker image. The abstraction provided by the engine makes it very flexible, users can think of any way of saving the state of the file system layout and map it to *Kameleon*.

5.4.4 Extend mechanism

Listing 13 shows a *Kameleon* recipe that builds a software appliance for the *hpl* benchmark. This recipe adds steps to the setup section and reuse steps from the recipe shown in Figure 5.4. This is done by using the `extend:` and `"@base"` keywords. Recipes are provided as templates, which enable a user to write a new recipe based on another existing recipe by overwriting certain sections and variables. The main purpose of this mechanism is to reduce the entry barrier for non-expert users by encouraging the reuse of recipes. This allows *Kameleon*'s users to take advantage from the recipes already developed by the community.

```
extend: qemu/debian7.yaml

global:

bootstrap:
  - "@base"
setup:
  - "@base"
  - install_software:
    - packages: g++ make openssh openmpi build-essential fort77
  - install_atlas:
    - repository: http://sourceforge.net/math-atlas/Stable/
    - version: "3.10.1"
  - install_hpl:
    - repository: "http://www.netlib.org/benchmark/hpl/"
    - version: "2.1"
    - hpl_makefile: "$$kameleon_recipe_dir/data/Make.Linux"

export:
  - "@base"
```

Listing 13: Extend mechanism.

5.4.5 Persistent cache mechanism

This mechanism as already mentioned constitutes one of the central contributions of *Kameleon* that enables the preservation of environments for experimentation. Thus, software appliances built are reconstruct-able any time. Chapter 6 will be dedicated enterly to this mechanism.

5.4.6 Comparison with the previous Kameleon version

During this thesis two versions of *Kameleon* were used. *Kameleon* was already presented in [49] and it has evolved form a single file script (900 lines of code) to a more modular improved version. Many isolation problems were solved given that the previous version was mainly based on *chroot*. The process of software appliance creation was structured with a new hierarchy based on sections, steps, microsteps and commands as already shown throughout this chapter. Additionally, the concept of context was added which enables to integrate more containers in a cleaner way, resolving many isolation problems. This results in a more stable tool, able to take advantage of recent technologies. The entry barrier for non-experts users was reduced as well, thanks to the new structured recipes and debugging mechanisms. Figure 5.5 shows the syntax of the old *Kameleon*. We can observe that all the process of creation is mixed in one sequence of steps, there is not distinction between *bootstrap*, setup and export.

5.5 Use cases

In this section, we demonstrate how *Kameleon* was used to build different software appliances. These software appliances illustrate a variety of software stacks (Table 5.5) with different requirements. Specially, they are taken from different domains (high performance computing, operating system and distributed system); they use different container technologies (chroot, Docker, VirtualBox, QEMU and real machine in Grid'5000); and they use different container isolation (lightweight, service, kernel module, and hardware dependent).

Table 5.5: Software appliances built with *Kameleon*

Name	**Description**	**Software stack**	**Containers used**	**Container isolation**	**Domain**
Debian basic	Debian console mode	Debian Wheezy	chroot, Docker, VirtualBox, QEMU, Grid'5000	Lightweight	Operating systems.
Debian Desktop	Debian GNOME Desktop environment	Debian Wheezy, GNOME	QEMU, VirtualBox	Service	Operating systems.
ArchLinux	Archlinux based system	ArchLinux last release	VirtualBox, QEMU	Lightweight	Operating systems.
CentOS	CentOS console mode	CentOS 6.5	VirtualBox, QEMU	Lightweight	Operating systems.
Dune	Dune library which provides safe and efficient access to privileged CPU features	Ubuntu Precise, Linux headers, Git, make, GCC	Grid'5000	kernel module	Operating systems
Formal java	A JavaScript module system	Debian Wheezy, Haskell, JavaScript modules	Chroot, Docker	Lightweight	Operating systems
CControl	Kernel Module to control the amount of cache available to an application	Debian wheezy, make, Git, build tools, CControl libraries, PAPI	QEMU, VirtualBox	kernel module	High performance computing.
hpl benchmark	LinPACK benchmark	Debian Wheezy, OpenMPI, OpenSSH, C++, make, Fortran, ATLAS library, *hpl* benchmark	chroot, Docker, VirtualBox, Grid'5000	Hardware dependent	High performance computing.
Hadoop	Framework for storage and large-scale processing	Ubuntu Lucid, Python, OpenSSH, Java 6, Hadoop.	chroot	Lightweight	Distributed computing.
TLM stack	Large scale electromagnetic simulations	Debian Wheezy, OpenMPI, OpenSSH, TLM application.	chroot	Lightweight	High performance computing.
OAR	Resource and task manager for HPC clusters and other computing infrastructures.	Debian wheezy, Git, Perl, Postgresql, OAR server packages	QEMU, VirtualBox	Service	High performance computing.

```
#### Basic Debian Kameleon recipe ###
global:
  workdir_base: /tmp/kameleon
  workdir: /tmp/kameleon
  distrib: ubuntu
  debian_version_name: lucid
  distrib_repository:  http://archive.ubuntu.com/ubuntu/
  output_environment_file_system_type: ext4
  include_dir: scripts
  arch: amd64
  kernel_arch: "amd64"
  network_hostname: "hadoop"
  extra_packages: "openssh-server wget"
  checkpoint_file:  "/tmp/ubuntu_lucid_hadoop.tgz"
  user_name: "root"
  key_dir: "/home/cristian/.ssh/"
steps:
  - bootstrap
  - system_config
  - mount_proc
  - kernel_install
  - software_install:
      - extra_packages
  - java_6/java_6_install
  - autologin
  - hadoop/config
  - hadoop/install
  - tuning/root_ssh_localkey
  - tuning/fix_locales
  - strip
  - umount_proc
#Building the appliance
  - build_appliance_kpartx:
      - clean_udev
      - create_raw_image
      - attach_kpartx_device
      - mkfs
      - mount_image
      - copy_system_tree
      - get_kernel_initrd
      - install_extlinux
      - umount_image
      - save_as_raw
      - save_as_vdi
  - clean
```

```
java_6_install:
  - adding_java_repository:
      - exec_chroot: apt-get -f install -y --force-yes python-software-properties
      - exec_chroot: add-apt-repository ppa:ferramroberto/java
      - exec_chroot: apt-get update
  - installing_java:
      - exec_chroot: bash -c "echo \"sun-java6-jdk shared/accepted-sun-dlj-v1-1 boolean true\" | debconf-set-selections"
      - exec_chroot: bash -c "DEBIAN_FRONTEND=noninteractive apt-get -f install -y --force-yes sun-java6-jdk"
```

Figure 5.5: Example of the old *Kameleon* recipe. This corresponds to the version 1.2.8 presented in [49]

5.5.1 Software Appliance Complexity

We start by describing different basic software appliances that can be used as a base experimental environment. Then we describe more complex software appliances used in research papers.

- **Basic software appliances**: These software appliances include several Linux flavors, for example: Fedora, CentOS, Debian, Archlinux. Different configurations were built from the very basic console mode to the complete desktop configuration. This shows that complete computer environments for researchers can be built.

- **Complex software appliances**: These software appliances were used in different research papers: an application for controlling cache utilization [101], a safe user-level access to privileged CPU features [12], a formal specification of a JavaScript module system [79]. Other appliances provide widely used computing frameworks such as *MapReduce*[27], benckmarks such as *hpl*[28] and batch schedulers such as *OAR*[29]

5.5.2 Container Isolation

Because software appliances require different levels of isolation at build time, a software appliance builder needs to provide isolation mechanisms. *Kameleon* provides isolation with its notion of context. Below are examples of the isolation requirements by different types of software appliances.

[27] `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`
[28] `http://www.netlib.org/benchmark/hpl/`
[29] `http://oar.imag.fr`

Lightweight.

Lightweight software appliances do not need any kind of isolation, thus they can run inside a *chroot*. This kind of software appliances can be exported to any format and run in any infrastructure. Examples of lightweight software appliances include: MPI + TLM[30] (electromagnetic simulation code), Map Reduce framework. Formal Java [79], *hpl* benchmark, Debian Wheezy basic system.

Service.

Service software appliances run a service (e.g. databases). Since the appliance's service may conflict with services running on the build machine, *Kameleon* allows the experimenter to use container isolation to isolate appliance services from build machine services.

Kernel modules.

When the installation of a kernel module is part of the software appliance creation, isolation at the level of operating system calls is needed, because the target kernel has to be running. Therefore, the IN context has to take place inside either a virtual or real machine. Sometimes a real machine is required, for example: 1) installation of CControl library for cache coloring[31], 2) installation of Dune[32], a kernel module that provides ordinary user programs with safe and efficient access to privileged CPU features, which are normally hidden when using a virtual machine.

Hardware dependent.

In contrast to the previous types of software appliances, which can be built and deployed on different machines, a hardware dependent software appliance must be built and deployed on the same machine. An example of hardware dependent software appliance is the *hpl* benchmark. This benchmark is based on the linear algebra library ATLAS, which must be optimized at built time for the deployment machine.

5.5.3 Results and Discussion

Table 5.6 shows the building time of some of the software appliances described above. The purpose of this data is to show the different steps that compose the build process and the time using various container technologies. For experimenters the process of generating an experimental environment could be perceived as a time consuming process. However, we observe that the built time of each of the software appliances is less than 30 minutes, which could encourage users to generate their custom experimental setups.

Hardware dependent software appliance evaluation

In this section, we use the *hpl* benchmark to evaluate hardware dependence container isolation. *hpl* benchmark requires the installation of multiple software packages whose parameters need to be configured, for performance, to the hardware that the appliance is running on. The parameter configuration requires significant compilation time. The evaluation was performed using two different machines.

- M1: Machine available in Grid'5000 in the cluster genepi. Intel Xeon E5420 QC CPU 2.5 Ghz with 8GB of RAM and HDD SATA disk.
- M2: Local machine. Intel Core i7-2760QM CPU 2.4 GHz with 8GB of RAM and SSD disk.

[30] `http://www.petr-lorenz.com/emgine/`
[31] `https://github.com/perarnau/ccontrol`
[32] `http://dune.scs.stanford.edu/`

Table 5.6: Building time of some software appliances. The time is presented in seconds.

Steps	AP1[1]	AP2[2]	AP3[3]	AP4[4]	AP5[5]	AP6[6]	AP7[7]	AP8[8]	AP9[9]	AP10[10]	AP11[11]	AP12[12]	AP13[13]	AP14[14]	AP15[15]
start-virtualbox							21	12	15	20	21	20	20	19	20
g5k-reserv					177										
start-docker			12												
start-qemu						10									
install-requirements						11	11	11	12	37	41	13	12	13	36
debootstrap	131	70	170	77		73		76	73			187	188	188	
yum-bootstrap										154	279				141
arch-bootstrap							150								
switch-context-virtualbox								10	10	162	105	93	26	35	32
switch-context-qemu						7									
-init-setup							5								
Boostrap	**131**	**70**	**182**	**77**	**177**	**101**	**187**	**109**	**110**	**373**	**446**	**313**	**246**	**255**	**229**
install-software	119	25	20	81	339	18	15	209	22	242	61	38	36	46	264
configure-system	7	6	6	6		6	17	6	6	11	8	11	11	10	10
configure-apt	13	13	7	9	37	9		9	9			12	15	13	
configure-kernel		5						5	5						
configure-keyboard	16	10	14	16	13	10		9	10			18	19	19	
install-atlas								497							
install-hpl								12							
install-ccontrol									18						
init-pxeboot										7					13
update-system										14	27				24
minimal-install										121					89
install-gnome												821			
oar-prereq-install						89								188	
oar-devel-prereq-install						20								50	
install-lambdajs				78											
upgrade-system					212										
install-kameleon					76										
oar-git-install						53									
oar-config-frontend						5									
tlm-installation		16													
-clean-setup	5	5		5	9	12		10	9	5	12	23	11		14
Setup	**291**	**150**	**229**	**272**	**863**	**323**	**219**	**866**	**189**	**773**	**554**	**1236**	**338**	**581**	**643**
qemu-save-appliance	63	83				88									
virtualbox-save-appliance							47	75	34	86	71	150	34		89
save-docker-appliance			5	6											
save-appliance-from-g5k					157										
Total	**354**	**233**	**234**	**278**	**1020**	**411**	**266**	**941**	**223**	**859**	**625**	**1386**	**372**	**581**	**732**

[1] chroot-debian
[2] chroot-tlm-mpi-debian
[3] docker-debian
[4] docker-formal-java-debian
[5] grid5000-kameleon-ubuntu
[6] qemu-oar-debian
[7] vbox-arch
[8] vbox-ATLAS-deb
[9] vbox-ccontrol-deb
[10] vbox-centos7
[11] vbox-centos
[12] vbox-debian-desktop
[13] vbox-debian
[14] vbox-debian-oar
[15] vbox-fedora

Table 5.7: Containers comparison machine M1.

Container	Build Time[Secs]	Image Size [Mbytes]	hpl result [MFLOPS]
VirtualBox	2722	1100	3.3
QEMU	1826	1200	109.1
Docker	2293	1600	110.1
Grid'5000	1782	638	113.3

Table 5.8: Containers comparison machine M2.

Container	Build Time[Secs]	Image Size [Mbytes]	hpl result [MFLOPS]
VirtualBox	1004	1100	8.1
QEMU	971	1200	189.7
Docker	1066	1600	222.3

The machine descriptions indicate that the machines differ only in their disk technology. Table 5.7 shows the results for machine M1. Table 5.8 shows the results for machine M2. The tables illustrate the time to build the software appliance (Build Time[Secs]), the software appliance size (Image Size[MBytes]) and the time to execute the benchmark *hpl* (hpl result[MFLOPS]). In the worst case scenario, the build time never exceeds one hour (or 3,600 seconds). All the elements necessary for reproducing these results are available in our repository[33].

Additionally, both tables show the millions of floating-point operations per second (MFLOPS) obtained by deploying the generated appliance and executing the benchmark. This is illustrative for a hypothetical experiment which goal would be to evaluate for example, the performance of virtual machine monitors. From this simple experiment, we can see that the virtualization provide by VirtualBox significantly impacts hpl benchmark performance: a factor of 34 times for M1 (from 113 Mflops to 3.3) and a factor of 27 times for M2 (222.3 to 8.1). In addition, the difference in performance is minimal for the other containers on a particular machine. Finally, across machines, the difference in disk technology make a significant difference in both build and execute time.

Table 5.9 illustrates the correlation between the image size of a software appliance and the cache size needed to store the data used to build the appliance. We are using the image size from Table 5.8: building *hpl* benchmark on machine M1. Finally, the total archive space to build all three appliances is illustrated on the last row. We can observe that storage requirements is reduced in a factor of 5.

Experiment packaging example

This section demonstrates how *Kameleon* and its persistent cache allow an experimenter to evaluate the performance of a high performance application using different virtualization techniques on different machines. This section's demonstration approximates the process used in the evaluation of Section 5.5.3. This section demonstrates the advantage of using *Kameleon* and its persistent cache system through an example. Let us suppose an experimenter wants to measure the performance of different techniques of virtualization and implementations of them for the execution of high performance applications. Assume that we have run an experiment that measures execution time for two virtualization techniques: system level virtualization (Docker) and full virtualization (VirtualBox and QEMU-KVM) on a machine M1. Now, suppose a different experimenter wants to run the same experiment in another machine M2.

Here are the issues they would face:

[33] This chapter was written using Org mode which enables to embed all the analysis presented. This is available along with persistent cache archives, *Kameleon* recipes and some additional scripts at `http://exptools.gforge.inria.fr/kameleon/`

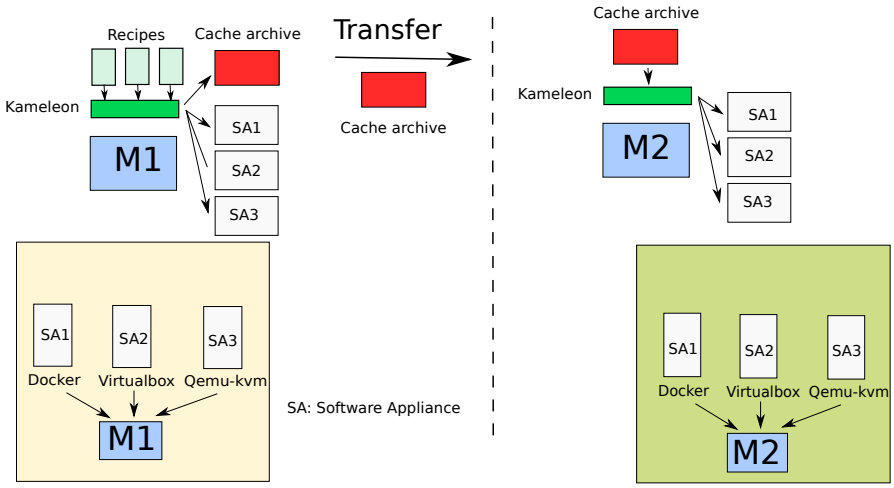

Figure 5.6: Example of experiment packaging with *Kameleon*.

Table 5.9: Some persistent cache archives

Software appliance	**Container**	**Image Size [Mbytes]**	**Cache Size[MBytes]**
hpl benchmark	VirtualBox	1100	581
hpl benchmark	QEMU	1200	582
hpl benchmark	Docker	1600	520
Archive for all appliances		3900	703

- The software appliances are rarely well described and the information of how they are configured is missing.
- Three different images have to be available which will consume space to store them and time to transfer.
- The images are static and introducing changes into them is not always easy and clean.
- Depending on the type of applications or benchmarks run in the experiment, recompilation could be needed in order to re-run the experiment in the same exact conditions. Therefore the images are not directly executable on M2.

The process using *Kameleon* is depicted in Figure 5.6. *Kameleon* brings the following advantages:

- All the details of composition and configuration resides on the recipes as shown in Section 5.4.
- In the process of generating the different software appliances, a persistent cache archive will be generated that contains all the data used during the generation of the respective software appliances. This is the only file that has to be stored and, in terms of size it is most of the time smaller than the images generated as shown in Table 5.9.
- The persistent cache archive contains all the original data used for generating the images. This means that the software appliance can be adapted to new contexts.

5.5.4 Future work

In future work, we plan to generalize the persistent cache to provide a repository of persistent cache files, and make this repository available to the community. Our vision of this community includes

researchers and software developers: anyone who needs to build a particular software stack. This repository will include the instructions (recipes and steps files) and its associated data. Therefore, multiple software appliances can be stored, reducing significantly the storage requirements (as demonstrated in the last row of Table 5.9). Using this repository and *Kameleon* eliminates the need to store large binary files. *Kameleon* can impact the manage of IT infrastructures as it can be used to manage the deployment and customization of software appliances. Furthermore, we are interested in exploring *Kameleon* as a platform for continuous integration. We believe that Kameleon's automation of software appliance building is well suited for continuous integration. Finally, because the whole environment setup is known, we believe that *Kameleon* can make bug tracking easier.

5.5.5 Conclusions

We introduced the concept of reconstructability which establishes the requirements that a software experimental setup has to meet for improving the reproducibility of experiments in computer science. We proposed *Kameleon* a software appliance builder that supports reconstructability. *Kameleon* provides a modular way to describe the construction of software appliances, which encourages collaboration and reuse of work. Support of reuse lowers the entry barrier for experimenters with low sysadmin skills. *Kameleon* persistent cache makes experimental setups reconstructable at any time.

Chapter 6

Reproducible appliances for experimentation

Experiment reproducibility is a milestone of the scientific method. Reproducibility of experiments in computer science would bring several advantages such as code re-usability and technology transfer. The reproducibility problem in computer science has been solved partially, addressing particular class of applications or single machine setups. In this chapter we present the design of a persistent cache mechanism that has been integrated to our software appliance builder *Kameleon*. The main goal of our approach is to enable the exact and independent reconstruction of a given software environment and the reuse of code. Additionally, we share our experience in finding a way to preserve over time; the software stack used for experimentation in computer science. A generalization of the persistent cache is proposed that would enable researchers to lower storage requirements for their appliances. The results shown in this chapter were published in a paper [112] presented at *TRIDENTCOM 2014*.

6.1 Introduction

In order to strengthen the results of a research it is important to carry out the experimental part under real environments. In some cases, these real environments consist in a complex software stack that normally comprises a configured operating system, kernel modules, run-time libraries, databases, special file systems, etc. The process of building those environments has two shortcomings: (a) It is a very time consuming task for the experimenter that depends on his/her expertise. (b) It is widely acknowledged that most of the time, it is hardly reproducible. A good practice at experimenting is to assure the reproducibility. For computational experiments this is a goal difficult to achieve and even a mere replication of the experiment is a challenge [37]. This is due to the numerous details that have to be taken into account. The process of repeating an experiment was carefully studied in [32] and among the many conclusions drawn, the difficulty of repeating published results was highly relevant.

With the advent of testbeds such as Grid'5000 [25] and FutureGrid [51], cloud-based testbeds like BonFIRE [1], the ubiquity of cloud computing infrastructures and the virtualization technology that is accessible to almost anyone that has a computer with modest requirements. Now it is possible to deploy virtual machines or operating system images, which makes interesting the approach of software appliances for experimentation. In [63] the author gives 13 ways that replicability is enhanced by using virtual appliances and virtual machine snapshots. Another close approach is shown in [45] where snapshots of computer systems are stored and shared in the cloud making computational analysis more reproducible. A system to create executable papers is shown in [14], which relies on the use of virtual machines and aims at improving the interactions between authors, reviewers and readers with reproducilibity purposes.

[1] `http://www.bonfire-project.eu`

Those approaches offer several advantages such as simplicity, portability, isolation and more importantly an exact replication of the environment but they incurred in high overheads in building, storing and transferring the final files obtained. Additionally, it is not clear the composition of the software stack and how it was configured. We lose the steps that let to their creation.

In the previous chapter we established that two requirements for reconstruct-ability are: to know exactly the sequence of actions that produced a determined environment for experimentation and to be able to change any action and regenerate another environment. It was already shown that our tool *Kameleon* strives to provide the former through a modular system of recipes where all actions to generate a software appliance are described. In this chapter, we present our approach to achieve the latter. The approach is based on a persistent cache mechanism that stores every piece of data (e.g., software packages, configuration files, scripts, etc.) used to construct the software appliance. Kameleon persistent cache mechanism presents three main advantages: (1) it can be used as a format to distribute and store individual and related software appliances (virtual cluster) incurring in less storage requirements; (2) *provenance of data*, anyone can look at the steps that led to the creation of a given experimental environment; (3) it helps to overcome widespread problems occasioned by small changes in binary versions, unavailability of software packages, changes in web addresses, etc. Experimental results and validation of this cache mechanism are shown in this chapter.

This chapter is structured as follows: In Section 6.2, some approaches to reproduce a given environment for experimentation are discussed. Then, the implementation of the persistent cache mechanism is shown in Section 6.3 which enables preservation of software stacks used in experimentation. In Section 6.4, we show some experimental results and validation of our approach. Finally the conclusions are presented in Section 6.6.

6.2 Related works

Experimenters have different options to make the environment for experimentation more reproducible. They can capture the environment where the experiment was run or they can use a more reproducible approach to set up the experiment from the beginning.

6.2.1 Tools for capturing the environment of experimentation

CDE [57] and ReproZip [29] are based on the capture of what it is necessary to run the experiment. They capture automatically software dependencies through the interception of Linux system calls. A package is created with all these dependencies enabling it to be run on different Linux distributions and versions. ReproZip unlike CDE allows the user to have more control over the final package created. Both tools provide the capacity of repeating a given experiment. However, they are aimed at single machine setups, they do not consider distributed environments and different environments that could interact between them.

6.2.2 Methods for setting up the environment of experimentation

Here, we describe the different methods that experimenters use for setting up and preserving their environments for experimentation. These methods apply to infrastructures where a whole software stack can be deployed (e.g., Grid'5000, FutureGrid, BonFIRE, any IaaS cloud, etc.). This is how the process shown in Section 5.1.1 is mapped to real use cases.

Manual

The experimenter deploys a *golden image* [2] that will be provisioned manually. The image modifications have to be saved some way (e.g snapshots) and several versions of the environment can be created for testing purposes. Possibly, the experimenter has to deal with the contextualization

[2] This term refers to the base operating system images available in an infrastructure.

of the images or it could be done using the underlying testbed infrastructure. In terms of reproducibility, the experimenter end up with a set of pre-configured software appliances that can be deployed later on the platform by him/her or another experimenter. This approach is relevant due to its simplicity and has been used and mentioned in [45] and [14]. Despite its simplicity, the storing of software appliances or snapshots incurs in high storage costs.

Script Automation

It is as well based on the deployment of golden images, however, the provisioning part is automated using scripts. The experimenter possibly has no need to save the image, because it can be reconstructed from the golden image at each deployment. Many experimenters opt for this approach because it gives a certain degree of reproduciblity and automation and it is simple compared to using configuration management tools. This was used in [11] for deploying and scheduling thousands of virtual machines on Grid'5000 testbed. Script automation incurs in less overhead when the environment has to be transmitted, for post execution. Nevertheless, it is still dependent on the images provided by the underlying platform.

Configuration management tools

Unlike the previous approaches, the golden images are provisioned this time with the help of configuration management tools (e.g., *Chef* [14] or *Puppet* [13]) which gives to the experimenter a high degree of automation and reproducibility. However, the process of porting the non-existing software towards those tools is complex and some administration expertise is needed. In [84] it is shown the viability of reproducible eScience on the cloud through the use of configuration management tools. A similar approach is shown in [15].

Software appliances

Experimenters can opt for software appliances that have to be contextualized at deployment time. In [81] the viability of this approach was shown. Those images can be either built or downloaded from existing testbed infrastructures (e.g Grid'5000, FutureGrid) or sites as TURNKEY [3] or cloud market [4] oriented to Amazon EC2 images. Those images are independent from the ones provided by the platform and experimenters have access to more operating system flavors. Different software stacks are available that are already configured, but we dont know anything about how they were built. We have already shown in Chapter 5 an extensive literature about the tools that enable the creation of software appliances.

6.3 Reconstructable software appliances

From the methods mentioned in the previous section, we believe that the use of software appliances gives the highest degree of flexibility and reproducibility as it provides a way for preserving the whole software stack. Our proposal is to make those software stacks easy to setup and reconstructable by taking advantage of the best of the aforementioned methods. As shown in Chapter 5, we propose to build software appliances with *Kameleon* which offers some standard methods for setting up software, similar to *Configuration management tools* but without its complexity. In order to assure the reconstruct-ability of the software appliance, we implemented a persistent cache module that generates an archive and enables the distribution of software appliances that can be reconstructed from scratch. It is targeted to make easier the reconstruction of custom software stacks in HPC, Grid, or Cloud-like environments.

```
global:
  user_name: kameleon
  user_password: $$user_name
  # Distribution
  distrib: debian
  release: wheezy
  kernel_arch: $$arch
  hostname: kameleon-$$distrib
  ## Disk options
  nbd_device: /dev/nbd1
  image_disk: $$kameleon_cwd/base_$$kameleon_recipe_name.qcow2
  image_size: 10G
  filesystem_type: ext4
  rootfs: $$kameleon_cwd/rootfs

  out_context:
    cmd: bash
    workdir: $$kameleon_cwd
    proxy_cache: 127.0.0.1

  in_context:
    cmd: USER=root HOME=/root LC_ALL=POSIX chroot $$kameleon_cwd/rootfs bash
    workdir: /root/kameleon_workdir
    proxy_cache: 127.0.0.1

bootstrap:
  - initialize_disk_chroot
  - debootstrap:
    - repository: http://ftp.debian.org/debian/
  - start_chroot

setup:
  - install_software:
    - packages: >
      debian-keyring sudo less vim curl less acpid linux-image-$$kernel_arch
  - configure_kernel
  - install_bootloader
  - configure_network
  - create_group:
    - name: admin
  - create_user:
    - name: $$user_name
    - groups: sudo admin
    - password: $$user_password
export:
  - qemu_save_appliance:
    - input: $$image_disk
    - output: $$kameleon_cwd/$$kameleon_recipe_name
    - save_as_qcow2
```

Figure 6.1: Kameleon recipe example

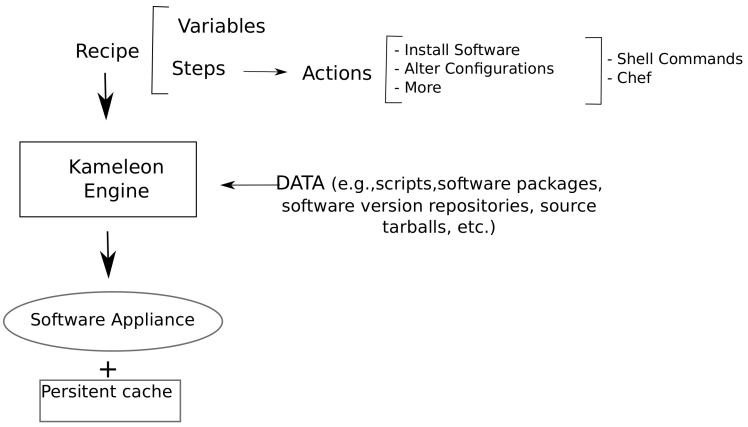

Figure 6.2: Software appliance creation with Kameleon

6.3.1 Requirements for reconstruct-ability

The approach for software appliance reconstruct-ability is based on four requirements:

1. A recipe (Figure 6.1) that describes how the software appliance is going to be built. This recipe is a higher level description easy to understand and contains some necessary meta-data in form of global variables and steps.

2. The *DATA* which is used as input of all the procedures described in the recipe. It encompasses software packages, tarballs, configuration files, control version repositores, scripts and every input data that make up a software appliance. Whenever used the term *DATA* in this chapter, it will refer to this.

3. *Kameleon* appliance builder which parses the recipe and carry out the building. This part includes as well the persistent cache mechanism that will be described later on.

4. Metadata that describes the context where the software appliance was built the first time. For instance: date of build, version of the external tools used during the build, etc.

5. A computer capable of executing *Kameleon*.

Therefore, the problem of guaranteeing the exact reconstruction of software appliances is reduced to keeping the three following parts unchanged: (1) the recipe, (2) DATA (3) *Kameleon* appliance builder. Two different experimenters having those three exact elements and fulfilling the requirements given by the Metadata (4) and computer hardware (5) will generate the same software appliance (under the hypothesis described in Section 5.1.2). Kameleon can generate in an automatic and transparent way a persistent cache archive that will contain the exact *DATA* used during the process of construction along with the recipe, steps and metadata, all bundled together enabling the easy distribution. The whole process is depicted in Figure 6.2.

Our approach to achieve reconstruct-ability is to use a persistent cache to capture all the *DATA* used during the construction. As we cannot guarantee that a particular download link will exist forever [116] or always point to the same software with the same version.

A persistent cache mechanism brings the two followings advantages: (a) Data can always be retrieved and (b) The software versions will be exactly the same.

[3] `http://www.turnkeylinux.org`
[4] `http://www.thecloudmarket.com`

6.3.2 Design

The persistent cache mechanism has to be transparent and lightweight for the user in the two following phases: the construction of the software appliance, and its respective ulterior reconstruction. As most of *DATA* comes from the network (e.g., operating system, software packages), the obvious approach was to integrate a caching proxy for web. Such a caching proxy will capture transparently every piece of data downloaded using the network. However, there are still some missing parts of the *DATA*, because some files - that make the software appliance unique - are provided by the user from its local machine or even worse some packages cannot be cached. That is the reason why we opted for an approach consisting in two parts:

- A caching web proxy, that caches packages coming from the network. This relies on Polipo [5] which is a very small, portable and lightweight caching web proxy. We chose Polipo because it can run with almost zero configuration. Polipo can be configured with different policies for validating the cache generated. Therefore, it can be forced to not request the server for up-to-date packages assuring that software packages will be always taken from the persistent cache. This is a desired behavior in order to avoid incompatibility due to changes in packages versions.

- Ad hoc procedures that cache what could not be cached using the caching web proxy. This represents data that come from control version repositories such as Git, svn, mercurial, etc or using https. These Ad hoc procedures are based on simple actions depending on the data to cache, for instance: control version repositories have special mechanisms to track the version used that are integrated into the *Kameleon* persistent cache module, user's files are cached by intercepting *kameleon* pipes, which are the only way to transfers files between contexts.

In order to make more clear the composition and limitations of the persistent cache, we define four properties of *DATA*:

- Location: it can be either Internal (I) or External (E).
- Cacheability: whether it is possible to cache it (C) or not ($\bar{\text{C}}$).
- Method of caching: it can be Proxy (P) or Ad hoc (A).
- Scope: two possible values *Private* or *Public*.

The scope makes necessary the creation of two types of cache *Private* and *Public* for distribution purposes. Combining the properties *Location*, *Cacheability* and *Method of caching* we can identify five types of data:

- E,C,P: data which comes from an external location (e.g., local network, internet) and can be cached with the proxy (e.g., Software packages, tarballs, input data).
- E,C,A: same external location, however, it cannot be cached with the proxy (e.g., version control repositories, https traffic).
- E,$\bar{\text{C}}$: this data comes from an external location but can not be cached due to some restrictions (e.g., proprietary licenses) or due to its size it can not be stored (e.g., big databases).
- I,C,A: data that comes from the local machine and it is cached by some ad hoc procedures.
- I,$\bar{\text{C}}$: it comes form local machine but can not be cached.

Figure 6.3 shows the composition of a generated persistent cache file. A hash is associated to both a step file and its generated persistent cache directory. This enables *Kameleon* to assure the

[5] `http://www.pps.univ-paris-diderot.fr/~jch/software/polipo/`

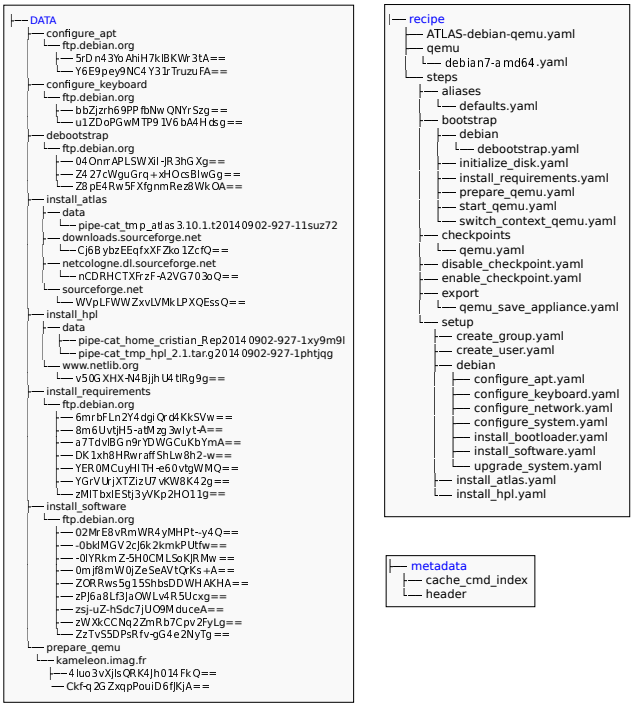

Figure 6.3: Here is depicted an example of the contents of a persistent cache archive. The requirements for reconstructabiliy are shown. The DATA is structured by step (*Kameleon* hierarchy) and it contains files, control version repositories and mainly cache files generated by Polipo. Only the steps that generate data are taken into account. The whole recipe is included with its respective step files and metadata.

coherency between instructions and data used to build a determined software appliance. This way of associating step files with persistent cache directories brings an adequate granularity (given that they represent an installation of one kind of software) for sharing bricks of software. A generalization of a cache could be implemented in which it would work as a central repository where users will share steps with their respective persistent cache files, lowering substantially the storage requirement needed for the software appliances.

Kameleon persistent cache mechanism enables the rebuilding of any software appliance from its respective persistent cache file. The only requirement is that the software appliance has to be built successfully a least once. The low size of *Kameleon* and Polipo (less than 1MB) makes feasible the distribution of the exact version used to create the environment, avoiding the incompatibility between versions.

Data type	**Persistent cache**	**Referenced cache**
O.S packages	Web proxy	Debian snapshot
Repositories	Hard copy of the repository	Checkout reference
User's files	Interception and storage of a hard copy	No option

Table 6.1: Persistent cache approaches

The persistent cache mechanism could use another alternative approach called *Reference cache*. It relies for the moment on systems like Debian snapshot [6] in order to access a certain dates and

[6] `http://snapshot.debian.org/`

General Appliances		
Name	Main software stack	Size [MB]
Hadoop	Java 1.6 Hadoop 1.03 Ubuntu 10.04 LTS	229
HPC Profiling	PAPI 5.1.0 TAU 2.22 OpenMPI 1.6.4 Debian Wheezy	226

Table 6.2: Software appliances generated

version of the packages. This is only use for the O.S layer and all the software that is available through the package manager. For revision control repositories, the referenced cache will keep the URL of the repository and the revision number. The two approaches are summirize in Table 6.1:

The approach using references is an option to lower the storage requirements but it will depend on an external service to be available. It is still under development and at the moment of writing the persistent cache approach is more reliable.

6.4 Experimental results and validation

This section will start with results of the persistent cache generated with *Kameleon* version 1.28 which were the subject of the paper [112]. The rest will be dedicated to persistent cache generated with the new version that was described in Chapter 5 and developed during the last part of this thesis. It will be shown in this section that *Kameleon* syntax can evolve without affecting the reconstruct-ability. All the persistent cache archives are available on Kameleon web site l[7].

6.4.1 Kameleon old version

As described in Section 6.3.1 we required a version of *Kameleon* which could be obtained by using the control version repository. The code is under a control revision system, the old engine can be retrieved from its git repository by doing:

```
1 $ git checkout remotes/origin/old/old-engine
```

Kameleon is a single script that can be executed in the following way:

```
1 sudo ./kameleon tests/debian_etch_oar2.2.17_i386.yaml --from_cache cache-debian_etch_oar2.2.17-2013-05-26.tar
```

Table 6.3: Software appliances generated

OAR Version	date of release	GNU/Linux version	Size [MB]
2.2.17	27 Nov 2009	Debian etch	112
2.3.5	30 Nov 2009	Debian etch	113
2.4.7	11 Jan 2011	Debian Lenny	137
2.5.2	23 May 2012	Debian Squeeze	140
2.5.0	5 Dec 2011	Debian Squeeze	140

In order to show that our approach is very portable between versions of Linux distributions, we carried out successfully construction and reconstruction of different appliances as shown in Table 6.2 that consist in different flavors of GNU/Linux (Debian, Ubuntu) and different middleware: Hadoop [8] and TAU [32]. A design goal was to achieve a self contained cache. Hence, we

[7] http://kameleon.imag.fr/archive/
[8] http://hadoop.apache.org/

tested the portability of the persistent cache mechanism. The aforementioned software appliances where reconstructed using their respective persistent cache files, the Kameleon engine and the Polipo binary which made only 984 K Bytes. This was tested in the following Linux distributions: Fedora 15, OpenSUSE 11.04, Ubuntu 10.4 and CentOS 6.0. Other tests consisted in reproducing old environments of test back to 2009 based on OAR [24] a very lightweight batch scheduler. The description is presented in Table 6.3.

6.4.2 Building old environments

The persistent cache mechanism enable the building of environments generated at any point of time. It does so by using the same versions that are compatible with the scripts used at the moment of the first generation of the software appliance. Not using the same exact versions can sometimes generate unexpected errors that are time consuming and researchers do not want to deal with. This could be one of the causes of the famous sentence "It worked yesterday". Problems with library versions dependency can appear as well, what it is known as *Dependency hell* [57].

We faced those problems when building software appliances based on *Archlinux* distribution and on the OAR batch scheduler. Their current versions posed several incompatibility problems with the scripts used for generating the software appliances a year ago. The persistent cache mechanism enabled the reconstruction of these software appliances.

6.5 Discussion

With the aim of capturing an experimental environment with reproducibility purposes, it is obvious that wrapping all the environment into a virtual machine is the simplest approach, which brings isolation and portability. Nevertheless, we exposed the following advantages of *Kameleon* over virtual machines as a means to achieve reproducibility.

- It is not possible to run everything on a virtual machine. It is most of the time possible to convert the virtual machine disk into a raw disk and deploy it into bare-metal. However, that implies additional steps for the user, it is not automatic.
- Space overhead, virtual machines are saved in large binary files.
- If the virtual machine needs to be modified, for instance, by installing a new version of a given software. It is necessary to uninstall the present version and install the required version, which is not always clean in most of the operating systems using either the package manager or tarballs.
- With *Kameleon* is a must to generate metadata. It is necessary to specify all the software versions to install, specific distribution packages to install, etc. It tells exactly what was done in order to create a given environment. This goes further than just the act of repeating. It enables the reuse of code, experimenters will understand the steps followed in order to get a certain complex stack of software. Thus, they will be able to adapt such stacks to their needs and get more insights.
- Rigid virtual machines are not a good option when dynamically deploy the virtual appliance under different environments what it is called as *Appliance contextualization*. The whole environment used to execute the experiment should be able to be reconfigured [97].

6.6 Conclusions and Future Works

Experiment reproduciblity is a big challenge nowadays in computer science, a lot of tools have been proposed to address this problem, however there are still some environments and experiments that are difficult to tackle. Commonly, experimenters lack of expertise to setup complex environments

necessary to reproduce a given experiment or to reuse the results obtained by someone else. We presented in this chapter, a very lightweight approach that leverage existing software and allows an experimenter to reconstruct independently the same software environment used by another experimenter. Its design offers a low storage requirement and a total control on the environment creation which in turn allows the experimenter to understand the software environment and introduce modifications into the process. Furthermore, several methods to carry out the setup of the environment for experimentation were described and we showed the advantages of our approach Kameleon. As a future work we plan to carry out more complex experiments with our approach and measure the gains in terms of reproducilibity and complexity as well as to study the contextualization of environments (e.g., post installation process) in different platforms.

Part IV

Conclusions

Chapter 7

Conclusions

During this thesis we have studied the conduction of experiments in computer science in general and mainly focus on our domains of research which are *Distributed Systems* and *High Performance Computing*. The difficulty involved in conducting an experiment and its later reproduction is due to the hard task of detailing all the factors that determined the state of the experimental context. The goal of experiments in our domain most of the times is to measure that our implementation is faster, it scales better, it uses less storage space, etc. As a consequence, the measures taken are highly dependent on the most minimal detail of the experimental context. There are many variables to take into account and many ways in which a determined experiment can be performed. Thus missing information about the procedure followed prevents the verification and reproduction of a given research work.

Due to the complexity of systems nowadays and the fast change of software and hardware, it is not surprising the difficulty in the simple fact of repeating an experiment. One first attempt to repeat successfully an experiment is to have access to the same software and hardware used, however, there are some unavoidable facts that could prevent short and long term reproduction of an experiment: some infrastructures are restricted to be used by few researchers, the access to the same hardware is costly, the lifespan of software and computer hardware is too short, software licenses and proprietary software, etc.

Through our studies we have found a plethora of tools that strives for conducting a more sound experimental process. Those tools seek to offer means for describing the context in which an experiment took place. To do so, they used different languages and abstractions for describing complex experimental workflows and embed as many details as possible. It is clear that no tool will cover all experimenter's necessities and that is why we put a lot of effort in comparing tools and providing their purpose. This was summarized in Chapter 2 and it is expected to be used as guide for researchers that want to improve the quality of their experiments. One conclusion of this study is that even though the current state of experimentation is not encouraging, this panorama will change given the number of tools available nowadays.

It seems obvious that due to this complexity users have to be assisted when conducting their experiments, manual controlled experiment is not viable anymore. The main idea is to provide a way to create, package, transfer and preserve their experiments. We found that experiment management tools have to serve three purposes:

- Make the act of experimenting less cumbersome. Reduce the complexity of managing large infrastructures and different software layers. The entry barrier of such tools could be reduced by encouraging collaboration where the reuse of code is made easy.

- Provide a way to package an experiment and make it easily portable across different software and hardware infrastructures. This package should generate enough metadata that rend the comprehension of the experiment straightforward. Regarding transmission, the goal to be achieved by an experiment tool is the possibility of being easily embedded in a publication or referenced. This has brought the concept of executable paper. We need to change the way

we communicate science and be in favor of using dynamic documents, online resources and invest effort in providing the maximum level of details about our experiment to the research community.

- Provide means to a least enable the short term preservation of the experimental environment.

In this thesis we addressed experimentation by performing a separation of concerns. We divided an experiment into two parts static and dynamic.

- **Static**: It refers to the part that do not change so often. The software stack and its configuration. Contrary to hardware, software is the cheapest requirement that we can preserved and should be accessible anytime. In this thesis we proposed an appliance builder called *Kameleon* that reduces the entry barrier for non-experts and help researchers to automate their experiments. We found with *Kameleon* a way to package software artifacts used for experimentation. More importantly, it has enabled to make software stacks reconstruct-able.

- **Dynamic**: It refers to the experiment execution, the definition of all the actions that have to be carried out during the experiment. This was addressed in this thesis by improving the experiment management tool called *Expo*. It was shown its flexibility and efficiency by implementing complex experiments that demanded a big amount of resources and complex workflows.

With this separation we believe that experimenter productivity is improved. When performing large scale experiments this separation is necessary for software installation procedures, otherwise the following issues could appear: a bottleneck when accessing the server for downloading packages, compilation process over several machines a part from being time consuming, it could be error-prone.

Another important contribution of this thesis is the use of experiment management tools for assisting users in the deployment and execution of their parallel applications. We showed the gains of performance by choosing better deployment schemes that have into account hardware capabilities. This was easily implemented using our experiment management tool and it opens the door to application optimization that are possible without knowing the internals of the application.

For illustrating the proposed experiment cycle and how the two tools interact together, a use case is presented in the next section.

7.1 Experiment cycle

The experimenter start by setting up all the software required for his/her environment of experimentation. For this the experimenter will use *Kameleon* to install (independent of the experimental workflow) all the software required using the best suited technology for him/her (Linux container, virtual machine, real machine, etc.). The setup of a software stack is an error-prone process where *Kameleon* features like checkpointing and interactive execution would come in handy. Several different software stacks can be created and exported to the most convenient format depending on the target infrastructure where the experiment will finally run. When the experimenter reaches a stable version of her/his environment, she/he will generate a persistent cache file which will freeze the software versions of the experimental environment and avoid any future incompatibility issue that could generate a considerable lost of time. Once the software stack to be used is set, all the workflow of the experimentation is done with *Expo*, this workflow can be tried locally in a virtual infrastructure by choosing the right infrastructure module. Many errors can be caught given that the infrastructure is running locally. Complex workflows of experimentation with many nodes can be easily expressed with *Expo*. The software appliances can be updated with more software if necessary in order to keep all the installation procedures in one place and then manage the deployment of software appliances which will make the experiment scale better. Finally, when all the experiments are finished successfully and the experimenters obtained the desired results, all

Expo scripts can be stored along with the persistent cache files generated by *Kameleon*. This will guarantee that the experimental workflow, experimental environment description and the exact software used in the experiment will be available for later reproduction.

7.2 Future works

One important step before further development of the tools presented during this thesis is to cross the adoption barrier. It is difficult to encourage researchers to automate their workflow for experimentation which is highly dependent on their technical skills. Unfortunately, no new tool come at no cost, resulting in the difficulty to convince researchers to change their experimental workflows. We believe that the level of adoption will increase with the level of maturity of the tools giving that early bugs, few documentation can discourage new users and make them return back to their previous workflow.

7.2.1 *Expo* perspectives

Currently, *Expo* enables the efficient execution of the experiment, it makes easier the managing of large amount of resources and provides an automatic collection of results. Although it is easier to conduct experiments than it was before, we still face some difficulties: failures are pervasive, experiments are not optimized, users do not have any help to run their application efficiently. The experiment tool should take decisions on behalf of the user, because, important events may occur when experimenting, for example:

- Some nodes failed when my experiment was deployed, I have to detect quickly and repair them (possibly by rebooting the machine).
- My application is getting a really bad performance, probably it is running with the wrong parameters. I have to stop it and not let it run for another 72 hours.
- The variance of my runs is low enough, it does not make any sense to do more runs.
- I need for my experiment a minimum of performance in the interconnection fabric, otherwise I could biased my results.
- My level of CPU performance is still good, I can deploy more virtual machines to simulate more clients.

Hence, an autonomic behavior is envisioned for dealing with this difficulties. Autonomic computing aims at developing self manage and self repair distributed systems for reducing deployment and administration costs. Experiments involving large amount of resources are costly, if we incorporate an intelligent behavior we could know for example: which tests can run in parallel, the number of runs needed to reach a certain confidence value, etc. We have already envisioned the evaluation and possible integration of the framework Frameself [2].

One of the biggest difficulties we had during the development of *Expo* was to choose the building blocks for the description language. We provided very high-level building blocks that can be customized for different purposes and some other operators that make easy the description of experiments with many nodes. In order to refine this operators and abstractions, a study about how researchers perform their experiments in our domain has to be conducted. The implications of such study on the improvement of the description language are threefold: the uncovering of hidden patterns, the reduction of the entry barrier for non-expert users and the enhancement on the readability. We can learn from studies about programming languages readability and its implication on software development which will provide a better criteria to perform a more complete evaluation of the current experiment management tools.

Another path for further research is the development of interfaces to increase the degree of interoperability of the tool and make it interact with workload generators and emulators systems

such as [115]. This will make possible a model of hybrid simulation as the one shown in [105] for large scale systems, where experimenters can take advantage of simulation, emulation and real execution techniques in order to enrich their environments of experimentation.

7.2.2 Kameleon perspectives

During the last part of this thesis *Kameleon* achieved a good stability and started to be used by a small community of local users. Apart from researchers, it has been used by engineers for building specialized software stacks for ARM architectures.

There is one path - among the many possible - for improving *Kameleon* that we would like to follow. The generalization of the persistent cache, where a repository of persistent cache files is available for the community. This community will not only include researchers, but also software developers and anyone who needs the creation of particular software stacks. This will reduce significantly the storage requirements for software appliances and it will make feasible that anyone with sufficiently data transmission and computing capacity can reconstruct his/her environment at will, without storing large amounts of data and without worrying about software incompatibilities. This can impact the manage of IT infrastructures as *Kameleon* can be used to manage the deployment and customization of software appliances. Impact on software development is foreseen as well, continuous integration can be easily automated and controlled and bugs reporting would be simplified as the whole environment configuration is known.

Bibliography

[1] D. Abramson, B. Bethwaite, C. Enticott, S. Garic, and T. Peachey. Parameter Exploration in Science and Engineering Using Many-Task Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):960–973, June 2011.

[2] Mahdi Ben Alaya and Thierry Monteil. Frameself: an ontology-based framework for the self-management of machine-to-machine systems. *Concurrency and Computation: Practice and Experience*, pages n/a–n/a, 2013.

[3] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Remote control: distributed application configuration, management, and visualization with Plush. In *Proceedings of the 21st conference on Large Installation System Administration Conference*, LISA'07, pages 15:1–15:19, Berkeley, CA, USA, 2007. USENIX Association.

[4] Jeannie Albrecht, Ryan Braud, Darren Dao, Nikolay Topilski, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Remote control: distributed application configuration, management, and visualization with plush. In *Proceedings of the 21st conference on Large Installation System Administration Conference*, LISA'07, pages 15:1–15:19, Berkeley, CA, USA, 2007. USENIX Association.

[5] Jeannie Albrecht, Christopher Tuttle, Ryan Braud, Darren Dao, Nikolay Topilski, Alex C. Snoeren, and Amin Vahdat. Distributed Application Configuration, Management, and Visualization with Plush. *ACM Transactions on Internet Technology*, 11:6:1–6:41, December 2011.

[6] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 28–28, Berkeley, CA, USA, 2006. USENIX Association.

[7] Jeannie Albrecht, Christopher Tuttle, Alex C. Snoeren, and Amin Vahdat. PlanetLab Application Management Using PluSH. *ACM SIGOPS Operating Systems Review*, 40:33–40, January 2006.

[8] Jeannie R. Albrecht. Bringing big systems to small schools: distributed systems for undergraduates. In *Proceedings of the 40th ACM technical symposium on Computer science education*, SIGCSE '09, pages 101–105, New York, NY, USA, 2009. ACM.

[9] M. Alexandru, T. Monteil, P. Lorenz, F. Coccetti, and H. Aubert. Large electromagnetic problem on large scale parallel computing systems. In *International Conference on High Performance Computing and Simulation*, 2012.

[10] S. Azarnoosh, M. Rynge, G. Juve, E. Deelman, M. Niec, M. Malawski, and R.F. da Silva. Introducing precip: An api for managing repeatable experiments in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 2, pages 19–26, Dec 2013.

[11] Daniel Balouek, Adrien Lèbre, and Flavien Quesnel. Flauncher and DVMS – Deploying and Scheduling Thousands of Virtual Machines on Hundreds of Nodes Distributed Geographically. In *IEEE International Scalable Computing Challenge (SCALE 2013), held in conjunction with CCGrid'2013*, Delft, Pays-Bas, 2013.

[12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association.

[13] Milind Bhandarkar, L. V. Kale, Eric de Sturler, and Jay Hoeflinger. Object-Based Adaptive Load Balancing for MPI Programs. In *Proceedings of the International Conference on Computational Science, San Francisco, CA, LNCS 2074*, pages 108–117, May 2001.

[14] Grant R. Brammer, Ralph W. Crosby, Suzanne Matthews, and Tiffani L. Williams. Paper mâché: Creating dynamic reproducible science. *Procedia CS*, 4:658–667, 2011.

[15] John Bresnahan, Tim Freeman, David LaBissoniere, and Kate Keahey. Managing appliance launches in infrastructure clouds. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*, TG '11, pages 12:1–12:7, New York, NY, USA, 2011. ACM.

[16] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 180–186, 2010.

[17] L. Broto, D. Hagimont, P. Stolf, N. De Palma, and S. Temate. Autonomic Management Policy Specification in Tune. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 1658–1663, New York, NY, USA, 2008.

[18] Tomasz Buchert. Orchestration d'expériences à l'aide de processus métier. In *ComPAS : Conférence d'informatique en Parallélisme, Architecture et Système.*, Grenoble, France, October 2012.

[19] Tomasz Buchert and Lucas Nussbaum. Leveraging business workflows in distributed systems research for the orchestration of reproducible and scalable experiments. In *9ème édition de la conférence Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication (2012)*, Lille, France, August 2012.

[20] Tomasz Buchert, Lucas Nussbaum, and Jens Gustedt. A workflow-inspired, modular and robust approach to experiments in distributed systems. In *CCGrid 2014 – The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, Illinois, USA, May 2014.

[21] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. A survey of general-purpose experiment management tools for distributed systems. *Future Generation Computer Systems*, 45(0):1 – 12, 2015.

[22] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. VisTrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 745–747, New York, NY, USA, 2006. ACM.

[23] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 745–747, New York, NY, USA, 2006. ACM.

[24] 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 *Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid (CCGrid'05) - Volume 2 - Volume 02*, CCGRID '05, pages 776–783, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Franck Cappello, Frédéric Desprez, Michel Dayde, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Nouredine Melab, Raymond Namyst, Pascale Primet, Olivier Richard, Eddy Caron, Julien Leduc, and Guillaume Mornet. Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform. In *6th IEEE/ACM International Workshop on Grid Computing (Grid)*, pages 99–106, November 2005.

[26] Alexandra Carpen-Amarie, Antoine Rougier, and FelixD. Lübbe. Stepping stones to reproducible research: A study of current practices in parallel computing. In *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 499–510. Springer International Publishing, 2014.

[27] Henri Casanova, Arnaud Legrand, and Martin Quinson. SimGrid: a Generic Framework for Large-Scale Distributed Experiments. In *Proceedings of the Tenth International Conference on Computer Modeling and Simulation*, UKSIM '08, pages 126–131, Washington, DC, USA, 2008. IEEE Computer Society.

[28] Bin Chen, Nong Xiao, Zhiping Cai, Zhiying Wang, and Ji Wang. Fast, on-demand software deployment with lightweight, independent virtual disk images. In *Grid and Cooperative Computing, 2009. GCC '09. Eighth International Conference on*, pages 16–23, Aug 2009.

[29] Fernando Chirigati, Dennis Shasha, and Juliana Freire. Reprozip: using provenance to support computational reproducibility. In *Proceedings of the 5th USENIX conference on Theory and Practice of Provenance*, TaPP'13, pages 1–1, Berkeley, CA, USA, 2013. USENIX Association.

[30] Gina Moraila Akash Shankaran Zuoming Shi Alex M Warren Christian Collberg, Todd Proebsting. Measuring reproducibility in computer systems research. Technical report, Arizona Univeristy, Technical Report, 2013.

[31] BrentN. Chun. Dart: Distributed automated regression testing for large-scale network applications. In Teruo Higashino, editor, *Principles of Distributed Systems*, volume 3544 of *Lecture Notes in Computer Science*, pages 20–36. Springer Berlin Heidelberg, 2005.

[32] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 47–47, Berkeley, CA, USA, 2004. USENIX Association.

[33] Benoit Claudel, Guillaume Huard, and Olivier Richard. Taktuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, HPDC '09, pages 91–100, New York, NY, USA, 2009. ACM.

[34] B. Clout and E. Aubanel. Ehgrid: An emulator of heterogeneous computational grids. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, May 2009.

[35] V. Curcin and M. Ghanem. Scientific workflow systems - can one size fit all? In *Biomedical Engineering Conference, 2008. CIBEC 2008. Cairo International*, pages 1–9, Dec 2008.

[36] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: Challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1345–1350, New York, NY, USA, 2008. ACM.

[37] Andrew Davison. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. *Computing in Science and Engg.*, 14(4):48–56, July 2012.

[38] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. Pegasus: Mapping scientific workflows onto the grid. In MariosD. Dikaiakos, editor, *Grid Computing*, volume 3165 of *Lecture Notes in Computer Science*, pages 11–20. Springer Berlin Heidelberg, 2004.

[39] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, 25(5):528–540, 2009.

[40] Karen D. Devine, Erik G. Boman, and George Karypis. Partitioning and load balancing for emerging parallel applications and architectures. In M. Heroux, A. Raghavan, and H. Simon, editors, *Frontiers of Scientific Computing*. SIAM, Philadelphia, 2006.

[41] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, and Yeh-Ching Chung. Pqemu: A parallel system emulator based on qemu. In *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pages 276–283, Dec 2011.

[42] Eelco Dolstra and Andres Löh. Nixos: A purely functional linux distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367–378, New York, NY, USA, 2008. ACM.

[43] D.L. Donoho, A Maleki, IU. Rahman, M. Shahram, and V. Stodden. Reproducible research in computational harmonic analysis. *Computing in Science Engineering*, 11(1):8–18, Jan 2009.

[44] C. Drummond. Replicability is not reproducibility: Nor is it good science. In *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, page 4972–4975, 2009.

[45] Joel T. Dudley and Atul J. Butte. In silico research in the era of cloud computing. *Nature Biotechnology*, 28(11):1181–1185, November 2010.

[46] Christoph Dwertmann, Ergin Mesut, Guillaume Jourjon, Max Ott, Thierry Rakotoarivelo, and Ivan Seskar. Mobile Experiments Made Easy with OMF/Orbit. In Konstantina Papagiannaki, Luigi Rizzo, Nick Feamster, and Renata Teixeira, editors, *SIGCOMM 2009, Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, New York, NY, USA, August 2009. ACM.

[47] Eric Eide, Leigh Stoller, and Jay Lepreau. An Experimentation Workbench for Replayable Networking Research. In *Proceedings of the 4th Symposium on Networked System Design and Implementation (NSDI)*, pages 215–228, 2007.

[48] Eric Eide, Leigh Stoller, Tim Stack, Juliana Freire, and Jay Lepreau. Integrated scientific workflow management for the Emulab network testbed. In *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ATEC '06, pages 33–33, Berkeley, CA, USA, 2006. USENIX Association.

[49] Joseph Emeras, Bruno Bzeznik, Olivier Richard, Yiannis Georgiou, and Cristian Ruiz. Reconstructing the software environment of an experiment with kameleon. In *Proceedings of the 5th ACM COMPUTE Conference: Intelligent and scalable system technologies*, COMPUTE '12, pages 16:1–16:8, New York, NY, USA, 2012. ACM.

[50] Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, and Noël De Palma. Reliable self-deployment of cloud applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1331–1338, New York, NY, USA, 2014. ACM.

[51] Geoffrey Fox, Gregor von Laszewski, Javier Diaz, Kate Keahey, Jose Fortes, Renato Figueiredo, Shava Smallen, Warren Smith, and Andrew Grimshaw. *FutureGrid - a reconfigurable testbed for Cloud, HPC, and Grid Computing*. CRC Computational Science. Chapman & Hall, 04/2013 2013.

[52] ClaudioDaniel Freire, Alina Quereilhac, Thierry Turletti, and Walid Dabbous. Automated Deployment and Customization of Routing Overlays on Planetlab. In Thanasis Korakis, Michael Zink, and Maximilian Ott, editors, *Testbeds and Research Infrastructure. Development of Networks and Communities*, volume 44 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 240–255. Springer Berlin Heidelberg, 2012.

[53] Wojciech Galuba, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. ProtoPeer: A P2P Toolkit Bridging the Gap Between Simulation and Live Deployement. In *Proceedings of the 2Nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 60:1–60:9, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[54] Matan Gavish and David Donoho. A universal identifier for computational results. *Procedia Computer Science*, 4(0):637 – 647, 2011. Proceedings of the International Conference on Computational Science, {ICCS} 2011.

[55] Grid5000. Grid5000:hardware, 2013.

[56] Romaric Guillier and Pascale Vicat-Blanc Primet. A User-oriented Test Suite for Transport Protocols Comparison in Datagrid Context. In *Proceedings of the 23rd International Conference on Information Networking*, ICOIN'09, pages 265–269, Piscataway, NJ, USA, 2009. IEEE Press.

[57] Philip J. Guo. Cde: run any linux application on-demand without installation. In *Proceedings of the 25th international conference on Large Installation System Administration*, LISA'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[58] A. Gupta, O. Sarood, L.V. Kale, and D. Milojicic. Improving hpc application performance in cloud through dynamic load balancing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 402–409, 2013.

[59] Jens Gustedt, Emmanuel Jeannot, and Martin Quinson. Experimental Methodologies for Large-Scale Systems: a Survey. *Parallel Processing Letters*, 19(3):399–418, 2009.

[60] W.J.R. Hoeffer. The transmission-line matrix method–theory and applications. *Microwave Theory and Techniques, IEEE Transactions on*, 33(10):882–893, oct 1985.

[61] Torsten Hoefler. Bridging performance analysis tools and analytic performance modeling for hpc. In *Proceedings of the 2010 conference on Parallel processing*, Euro-Par 2010, pages 483–491, Berlin, Heidelberg, 2011. Springer-Verlag.

[62] Zhengxiong Hou, Jing Tie, Xingshe Zhou, I. Foster, and M. Wilde. Adem: Automating deployment and management of application software on the open science grid. In *Grid Computing, 2009 10th IEEE/ACM International Conference on*, pages 130–137, Oct 2009.

[63] Bill Howe. Virtual appliances, cloud computing, and reproducible research. *Computing in Science and Engg.*, 14(4):36–41, July 2012.

[64] Sili Huang, Eric Aubanel, and VirendrakumarC. Bhavsar. Pagrid: A mesh partitioner for computational grids. *Journal of Grid Computing*, 4(1):71–88, 2006.

[65] Duncan Hull, Katherine Wolstencroft, Robert Stevens, Carole Goble, Matthew Pocock, Peter Li, and Thomas Oinn. Taverna: a tool for building and running workflows of services. *Nucleic Acids Research*, 34(Web Server issue):729–732, July 2006.

[66] Sascha Hunold and Jesper Larsson Träff. On the state and importance of reproducible experimental research in parallel computing. *CoRR*, abs/1308.3648, 2013.

[67] Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, and Takahiro Hirofuchi. Using the EXECO toolbox to perform automatic and reproducible cloud experiments. In *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013*, Bristol, Royaume-Uni, September 2013.

[68] P. Jakubco, N. Adam, and E. Dankoval. Distributed computer emulation: Using opencl framework. In *Applied Machine Intelligence and Informatics (SAMI), 2011 IEEE 9th International Symposium on*, pages 333–338, Jan 2011.

[69] D. Jayasinghe, G. Swint, S. Malkowski, J. Li, Qingyang Wang, Junhee Park, and C. Pu. Expertus: A generator approach to automate performance testing in iaas clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 115–122, June 2012.

[70] E. Jeannot. Experimental validation of grid algorithms: A comparison of methodologies. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[71] Emmanuel Jeanvoine, Luc Sarzyniec, and Lucas Nussbaum. Kadeploy3: Efficient and Scalable Operating System Provisioning. *USENIX ;login:*, 38(1):38–44, February 2013.

[72] P.B. Johns. A symmetrical condensed node for the tlm method. *IEEE Trans. on Microwave Theory and Tech.*, 35(4):370–377, apr 1987.

[73] David Johnson. A theoretician's guide to the experimental analysis of algorithms, 1996.

[74] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[75] Guillaume Jourjon, Salil Kanhere, and Jun Yao. Impact of IREEL on CSE Lectures. In *the 16th Annual Conference on Innovation and Technology in Computer Science Education (ACM ITiCSE 2011)*, pages 1–6, Germany, June 2011.

[76] Guillaume Jourjon, Thierry Rakotoarivelo, and Max Ott. From Learning to Researching - Ease the shift through testbeds. In *Internatinonal ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, pages 496–505, Berlin, May 2010. Springer-Verlag.

[77] Guillaume Jourjon, Thierry Rakotoarivelo, and Max Ott. Why simulate when you can experience? In *ACM Special Interest Group on Data Communications (ACM SIGCOMM) Education Workshop*, page N/A, Toronto, August 2011.

[78] Guillaume Jourjon, Thierry Rakotoarivelo, and Max Ott. A Portal to Support Rigorous Experimental Methodology in Networking Research. In Thanasis Korakis, Hongbin Li, Phuoc Tran-Gia, and Hong-Shik Park, editors, *Testbeds and Research Infrastructure. Development of Networks and Communities*, volume 90 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 223–238. Springer Berlin Heidelberg, 2012.

[79] Seonghoon Kang and Sukyoung Ryu. Formal specification of a javascript module system. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 621–638, New York, NY, USA, 2012. ACM.

[80] A. Kangarlou, Dongyan Xu, U.C. Kozat, P. Padala, B. Lantz, and K. Igarashi. In-network live snapshot service for recovering virtual infrastructures. *Network, IEEE*, 25(4):12–19, July 2011.

[81] Katarzyna Keahey and Tim Freeman. Contextualization: Providing one-click virtual clusters. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 301–308, Washington, DC, USA, 2008. IEEE Computer Society.

[82] M. Kesavan, A Gavrilovska, and K. Schwan. Xerxes: Distributed load generator for cloud-scale experimentation. In *Open Cirrus Summit (OCS), 2012 Seventh*, pages 20–24, June 2012.

[83] Fadi KHALIL. Multi-scale modeling: from electromagnetism to grid, 2009.

[84] Jonathan Klinginsmith, Malika Mahoui, and Yuqing Melanie Wu. Towards Reproducible eScience in the Cloud. In *3rd IEEE International Conference on Cloud Computing Technology and Science (CLOUDCOM)*, pages 582–586, 2011.

[85] G.A. Koenig and L.V. Kale. Optimizing distributed application performance using dynamic grid topology-aware load balancing. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, 2007.

[86] Oren Laadan, Jason Nieh, and Nicolas Viennot. Teaching operating systems using virtual appliances and distributed version control. In *Proceedings of the 41st ACM technical symposium on Computer science education*, SIGCSE '10, pages 480–484, New York, NY, USA, 2010. ACM.

[87] Mathieu Lacage, Martin Ferrari, Mads Hansen, Thierry Turletti, and Walid Dabbous. NEPI: Using Independent Simulators, Emulators, and Testbeds for Easy Experimentation. *SIGOPS Oper. Syst. Rev.*, 43(4):60–65, January 2010.

[88] Stephane Lanteri, Julien Leduc, Nouredine Melab, Guillaume Mornet, Raymond Namyst, Benjamin Quetier, and Olivier Richard. Grid'5000: a large scale and highly reconfigurable grid experimental testbed.

[89] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 185–198, Berkeley, CA, USA, 2009. USENIX Association.

[90] Bertram Ludäscher, Ilkay Altintas, Shawn Bowers, Julian Cummings, Terence Critchlow, Ewa Deelman, David D Roure, Juliana Freire, Carole Goble, Matthew Jones, et al. Scientific process automation and workflow management. 2009.

[91] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.

[92] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, 2006.

[93] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 265–276, New York, NY, USA, 2009. ACM.

[94] Farrukh Nadeem, Radu Prodan, Thomas Fahringer, and Alexandru Iosup. Benchmarking grid applications. In *Grid Middleware and Services*, pages 19–37. Springer US, 2008.

[95] S. Naicken, B. Livingston, A. Basu, S. Rodhetbhai, I. Wakeman, and D. Chalmers. The State of Peer-to-peer Simulators and Simulations. *SIGCOMM Comput. Commun. Rev.*, 37(2):95–98, March 2007.

[96] Shaya Potter Jason Nieh. Improving virtual appliance management through virtual layered file systems. Technical report, Columbia Univeristy, Technical Report CUCS-008-09, 2009.

[97] Daniel Oliveira, FernandaAraujo Baião, and Marta Mattoso. Towards a taxonomy for cloud computing from an e-science perspective. In Nick Antonopoulos and Lee Gillam, editors, *Cloud Computing*, Computer Communications and Networks, pages 47–62. Springer London, 2010.

[98] M. Ott, I. Seskar, R. Siraccusa, and M. Singh. ORBIT testbed software architecture: supporting experiments as a service. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2005. Tridentcom 2005. First International Conference on*, pages 136–145, 2005.

[99] Andrew Pavlo, Peter Couvares, Rebekah Gietzel, Anatoly Karp, Ian D. Alderman, Miron Livny, and Charles Bacon. The nmi build & test laboratory: continuous integration framework for distributed computing software. In *Proceedings of the 20th conference on Large Installation System Administration*, LISA '06, pages 21–21, Berkeley, CA, USA, 2006. USENIX Association.

[100] Roger D. Peng and Sandrah P. Eckel. Distributed reproducible research using cached computations. *Computing in Science and Engg.*, 11(1):28–34, January 2009.

[101] Swann Perarnau, Marc Tchiboukdjian, and Guillaume Huard. Controlling cache utilization of hpc applications. In *International Conference on Supercomputing (ICS)*, 2011.

[102] Fernando Pérez and Brian E. Granger. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng.*, 9(3):21–29, May 2007.

[103] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, January 2003.

[104] R. Prodan, T. Fahringer, and F. Franz. On using ZENTURIO for performance and parameter studies on cluster and Grid architectures. In *Proceedings of Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 185–192, Feb 2003.

[105] A. Quereilhac, M. Lacage, C. Freire, T. Turletti, and W. Dabbous. NEPI: An integration framework for Network Experimentation. In *19th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–5, Sept 2011.

[106] Alina Quereilhac, Daniel Camara, Thierry Turletti, and Walid Dabbous. Experimentation with large scale ICN multimedia services on the Internet made easy. *IEEE COMSOC MMTC E-Letter*, 8(4):10–12, July 2013.

[107] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. OMF: a control and management framework for networking testbeds. *ACM SIGOPS Operating Systems Review*, 43(4):54–59, Jan 2010.

[108] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, pages 1664–1669 Vol. 3, 2005.

[109] Hélène Renard, Yves Robert, and Frédéric Vivien. Static load-balancing techniques for iterative computations on heterogeneous clusters. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003 Parallel Processing*, volume 2790 of *Lecture Notes in Computer Science*, pages 148–159. Springer Berlin Heidelberg, 2003.

[110] Cristian Ruiz, Mihai Alenxandru, Olivier Richard, Thierry Monteil, and Herve Aubert. Platform calibration for load balancing of large simulations: TLM case. In *CCGrid 2014 – The 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Chicago, Illinois, USA, 2014.

[111] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. Reconstructable software appliances with kameleon. *SIGOPS Oper. Syst. Rev.*, 49(1):80–89, January 2015.

[112] Cristian Ruiz, Olivier Richard, and Joseph Emeras. Reproducible software appliances for experimentation. In *Proceedings of the 9th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (Tridentcom)*, Guangzhou, China, 2014.

[113] Cristian Ruiz, Olivier Richard, Brice Videau, and Iegorov Oleg. Managing Large Scale Experiments in Distributed Testbeds. In *Proceedings of the 11th IASTED International Conference*, pages 628–636. IASTED, ACTA Press, feb 2013.

[114] Constantine Sapuntzakis, David Brumley, Ramesh Chandra, Nickolai Zeldovich, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the 17th USENIX conference on System administration*, LISA '03, pages 181–194, Berkeley, CA, USA, 2003. USENIX Association.

[115] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. Design and evaluation of a virtual experimental environment for distributed systems. In *PDP*, pages 172–179, 2013.

[116] Carmine Sellitto. The impact of impermanent web-located citations: A study of 123 scholarly conference publications. *Journal of the American Society for Information Science and Technology*, 56(7):695–703, 2005.

[117] Xuanhua Shi, Chao Liu, Song Wu, Hai Jin, Xiaoxin Wu, and Li Deng. A cloud service cache system based on memory template of virtual machine. In *Chinagrid Conference (ChinaGrid), 2011 Sixth Annual*, pages 168–173, Aug 2011.

[118] Christos Siaterlis and Marcelo Masera. A survey of software tools for the creation of networked testbeds. *International Journal On Advances in Security*, 3(1 and 2):1–12, 2010.

[119] C. Sun, Le He, Qingbo Wang, and R. Willenborg. Simplifying service deployment with virtual appliances. In *Services Computing, 2008. SCC '08. IEEE International Conference on*, volume 2, pages 265–272, July 2008.

[120] Masahiro Tanaka and Osamu Tatebe. Pwrake: a parallel and distributed flexible workflow management tool for wide-area data intensive computing. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 356–359, New York, NY, USA, 2010. ACM.

[121] Walter F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.

[122] Nikolay Topilski, Jeannie Albrecht, and Amin Vahdat. Improving scalability and fault tolerance in an application management infrastructure. In *First USENIX Workshop on Large-Scale Computing*, LASCO'08, pages 2:1–2:12, Berkeley, CA, USA, 2008. USENIX Association.

[123] Sander Van Der Burg and Eelco Dolstra. Disnix: A toolset for distributed deployment. *Sci. Comput. Program.*, 79:52–69, January 2014.

[124] B. Videau and O. Richard. Expo : un moteur de conduite d'experiences pour plates-forme dedies. In *Conference Fransaise en Systemes d'Exploitation (CFSE)*, 2008.

[125] Brice Videau, Corinne Touati, and Olivier Richard. Toward an experiment engine for lightweight grids. In *MetroGrid workshop : Metrology for Grid Networks*. ACM publishing, October 2007.

[126] Yanyan Wang. *Automating experimentation with distributed systems using generative techniques*. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA, 2006. AAI3219040.

[127] Yanyan Wang, Antonio Carzaniga, and Alexander L. Wolf. Four enhancements to automated distributed system experimentation methods. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 491–500, New York, NY, USA, 2008. ACM.

[128] Yanyan Wang, Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. Automating Experimentation on Distributed Testbeds. In *Proceedings of the 20th IEEE/ACM International Conference On Automated Software Engineering (ASE)*, ASE '05, pages 164–173, New York, NY, USA, 2005. ACM.

[129] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005. `http://www.cs.utsa.edu/~whaley/papers/spercw04.ps`.

[130] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[131] Jolyon White, Guillaume Jourjon, Thierry Rakotoarivelo, and Max Ott. Measurement Architectures for Network Experiments with Disconnected Mobile Nodes. In Anastasius Gavras, Nguyen Huu Thanh, and Jeff Chase, editors, *TridentCom 2010, 6th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, pages 315–330, Heidelberg, Germany, May 2010. ICST, Springer-Verlag Berlin.

[132] Jia Yu and Rajkumar Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Record*, 34:44–49, September 2005.

[133] Tianle Zhang, Zhihui Du, Yinong Chen, Xiang Ji, and Xiaoying Wang. Typical virtual appliances: An optimized mechanism for virtual appliances provisioning and management. *Journal of Systems and Software*, 84(3):377 – 387, 2011.

Part V

Appendix

Appendix A

Other experiment descriptions implemented

```
require 'g5k_api'

set :user, "cruizsanabria"
set :gateway, "grenoble.g5k"
set :resources, "MyExperiment.resources"

reserv = connection(:type => "Grid5000")

reserv.resources = {:nancy =>["{cluster='griffon'}/nodes=10"],
  :luxembourg => ["{cluster='granduc'}/nodes=10"],
  :reims =>["nodes=10"]}

reserv.name ="Tlm Load Balancing"

WORK_DIRECTORY="~/Exp_tlm_load_balancing"
TLM_TARBALL = "tlm_load_balancing.tar"
RUNS = 5
SIMULATION_PARAMETERS = "1 10000 152 172 86 matched"
RESULTS_FILE = "tlm_vs_tlmlb"
############ Experiment workflow ############################################################
task :run_reservation do
  reserv.run!
end

task :extracting_and_compiling, :target => resources, :once => true, :each => :site do
  msg("Compiling in site ")
  unless check("ls #{WORK_DIRECTORY}/TLMME_lb")
    run("mkdir -p #{WORK_DIRECTORY}")
    put("/tmp/#{TLM_TARBALL}","#{WORK_DIRECTORY}/#{TLM_TARBALL}")
    run("cd #{WORK_DIRECTORY}; tar -xf #{TLM_TARBALL}")
    run("cd #{WORK_DIRECTORY}/TLMME_lb/tlm/; make ITERATIONS=200")
    run("cd #{WORK_DIRECTORY}/TLMME_lb/tlm/; make ITERATIONS=200 MAIN=main_lb_test EXESUFFIX=load_test")
  end
  put("/tmp/nodes.deployed","#{WORK_DIRECTORY}/TLMME_lb/tlm/")
end

task :tlm_lb, :target => resources.first, :sync => true do
  RUNS.times do
    run("cd #{WORK_DIRECTORY}/TLMME_lb/tlm/; ./grid_run_lb #{SIMULATION_PARAMETERS}")
  end
end

task :tlm, :target => resources.first, :sync => true do
  RUNS.times do
    run("cd #{WORK_DIRECTORY}/TLMME_lb/tlm/; ./grid_run #{SIMULATION_PARAMETERS}")
  end
end
```

Listing 14: Description file of an experiment that compares the gains obtained when applying load balacing to a large simulation based on TLM. Some tasks were omitted due to space constraints.

```
require 'g5k_api'
set :user, "root"
set :gw_user, "cruizsanabria" ## replace with your user
set :resources, "MyExperiment.resources"
reserv = connection(:type => "Grid5000")
reserv.resources = { :lyon => ["nodes=2"] }
reserv.environment = "http://public.nancy.grid5000.fr/~dlehoczky/newimage.dsc"
reserv.name = "mpi trace collection"

##### Tasks Definition ######################################
task :run_reservation do
   reserv.run!
end

### Generating password less communication
task :config_ssh do
   msg("Generating SSH config")
   File.open("/tmp/config",'w+') do |f|
      f.puts "Host *
      StrictHostKeyChecking no
      UserKnownHostsFile=/dev/null "
   end
end

task :generating_ssh_keys do
   run("mkdir -p /tmp/temp_keys/")
   run("ssh-keygen -P '' -f /tmp/temp_keys/key") unless check("ls /tmp/temp_keys/key")
end

task :trans_keys, :target => resources do
   put("/tmp/config","/root/.ssh/")
   put("/tmp/temp_keys/key","/root/.ssh/id_rsa")
   put("/tmp/temp_keys/key.pub","/root/.ssh/id_rsa.pub")
end

task :copy_identity do
   resources.each{ |node|
      run("ssh-copy-id -i /tmp/temp_keys/key.pub root@#{node.name}") #,:target => gateway)
   }
end

### Getting the benchmark
task :get_benchmark, :target => resources do
   unless check("ls /tmp/NPB3.3.tar") then
      msg("Getting NAS benchmark")
      run("cd /tmp/; wget -q http://public.grenoble.grid5000.fr/~cruizsanabria/NPB3.3.tar")
      run("cd /tmp/; tar -xvf NPB3.3.tar")
   end
end

task :compile_benchmark_lu, :target => resources do
   compile = "export PATH=/usr/local/tau-install/x86_64/bin/:$PATH;"
   compile += "export TAU_MAKEFILE=/usr/local/tau-install/x86_64/lib/Makefile.tau-papi-mpi-pdt;"
   compile += "make lu NPROCS=8 CLASS=A MPIF77=tau_f90.sh -C /tmp/NPB3.3/NPB3.3-MPI/"
   run(compile)
end

## Generating machinefile
task :transfering_machinefile, :target => resources.first do
  put(resources.nodefile,"/tmp/machinefile")
end

task :run_mpi, :target => resources.first do
   mpi_params = "-x TAU_TRACE=1 -x TRACEDIR=/tmp/mpi_traces -np 8 -machinefile /tmp/machinefile"
   run("/usr/local/openmpi-1.6.4-install/bin/mpirun #{mpi_params} /tmp/NPB3.3/NPB3.3-MPI/bin/lu.A.8")
end

## Gathering traces and merging
task :gathering_traces, :target => resources.first do
   resources.each{ |node|
      msg("Merging results of node #{node.name}")
      run("scp -r #{node.name}:/tmp/mpi_traces/* /tmp/mpi_traces")
   }
   cmd_merge = "export PATH=/usr/local/tau-install/x86_64/bin/:$PATH;"
   cmd_merge += "cd /tmp/mpi_traces/; tau_treemerge.pl"
   run(cmd_merge)
   run("cd /tmp/mpi_traces/; /usr/local/akypuera-install/bin/tau2paje tau.trc tau.edf 1>lu.A.8.paje 2>tau2paje.error")
end
```

Listing 15: Description file of an experiment that traces a NAS benchmark with TAU.

Appendix B

Experiment management tools comparison

The following descriptions were used for comparing *Expo* against *XpFlow* and *Execo*. The conclusions of this comparison were shown in Chapter 3.

```
require 'plain_api'

set :resources, "MyExperiment.resources"
set :user, "root"

reserv = connection(:type => "Plain",
                    :nodes_file => "vboxnodes")

PIPE_LENGTH = 800
RUNS = 5

task :install_packages, :target => resources do
  packages = "make g++ openssh-server openmpi-bin openmpi-common openmpi-dev"
  run(" apt-get -y --force-yes install #{packages} 2>&1")
  run("ifconfig eth1 down")
end

task :compiling_tlm, :target => resources do
  put("/home/cristian/Dev/C++/TLM_2013/tlm_clean_version.tar","/root/")
  run("cd /root/ && tar -xf tlm_clean_version.tar")
  run("cd /root/TLMME/tlm/ && make")
end

task :conf_mpi, :target => resources.first do
  put(resources.nodefile, "/root/TLMME/tlm/bin/")
  put("run_cluster", "/root/TLMME/tlm/")
  run("cd /root/TLMME/tlm/ && chmod +x run_cluster")
end

task :run_tlm, :target => resources.first do

  [2,4,6].each do |num_procs|
    RUNS.times{
      run("cd /root/TLMME/tlm/;./run_cluster #{num_pr3ocs} 100 #{PIPE_LENGTH/num_procs} 86 43 matched")
    }
  end
end
```

Listing 16: Experiment that measures the best performance of TLM code using *Expo*

```
from execo import *
from execo_engine import *

class tlm_performance(Engine):

    def run(self):
        hosts= [Host('192.168.56.101', user = 'root'),Host('192.168.56.102', user = 'root')]
        logger.info( "Starting Experiment")
        logger.info("Installing packages")
        Remote(" apt-get -y --force-yes install \
                make g++ openssh-server openmpi-bin openmpi-common openmpi-dev 2>&1",
                hosts).run()
        logger.info("transfering code")
        Put(hosts,
            ["/home/cristian/Dev/C++/TLM_2013/tlm_clean_version.tar"],
                "/root/").run()
        logger.info("Compiling")
        Remote("tar -xf tlm_clean_version.tar",hosts).run()
        Remote("cd /root/TLMME/tlm/ && make ",hosts).run()

        logger.info("MPI configuration")
        f = open("machines", "w")
        for node in hosts:
            f.write("%s \n"  % node.address)
        f.close()

        Put(hosts[0],["machines"],"/root/TLMME/tlm/bin/").run()
        Put(hosts[0], ["run_cluster"], "/root/TLMME/tlm/").run()
        SshProcess("cd /root/TLMME/tlm/ ; chmod +x run_cluster",hosts[0]).run()

        logger.info("starting tlm execution")
        PIPE_LENGTH = 800
        RUNS = 5
        result_file = "execution_time_tlm.txt"
        f = open(result_file, "w")
        for num_procs in [2,4,6]:
            for run in range(RUNS):
                tlm_parallel = SshProcess(
                          "cd /root/TLMME/tlm/;./run_cluster"
                          " %d 100 %d 86 43 matched" %(num_procs,PIPE_LENGTH/num_procs),
                          hosts[0])

                tlm_parallel.run()
                tlm_parallel.wait()
                execution_time = tlm_parallel.end_date - tlm_parallel.start_date
                logger.info("Execution time is : %d" % execution_time)
                #f.write("\n")

if __name__ == "__main__":
    engine = tlm_performance()
    engine.start()
```

Listing 17: Experiment that measures the best performance of TLM code using *Execo*

```
activity :install_package do |nodes, packages|
  log("Installing packages")
  r = execute_many(nodes, "apt-get -y --force-yes install #{packages} 2>&1")
  r = execute_many(nodes,"ifconfig eth1 down")
end

activity :compile_tlm do |nodes|
  r = execute_many(nodes, "cd /root/ && tar -xf tlm_clean_version.tar")
  r = execute_many(nodes, "cd /root/TLMME/tlm/ && make")
end

activity :tlm_execution do |nodes,runs,pipe_length|
  [2,4,6].each do |num_procs|
    runs.times{
      r = execute_one(nodes.first, "cd /root/TLMME/tlm/;./run_cluster #{num_procs} 100 #{pipe_length/num_procs} 86 43 matched")
      log(r)
    }
  end
end

activity :conf_mpi do |nodes|

  log("MPI configuration")
  File.open("machines", 'w') do |f|
    nodes.each{ |node|
      f.puts(node.host)
    }
  end

end

process :main do
  log "Installing packages"
  PIPE_LENGTH = 800
  RUNS = 5

  log "loading nodes"
  ip_adresses = YAML::load(File.read("vboxnodes"))
  hosts = []
  ip_adresses.each{ |ip|
    hosts.push(simple_node("root@#{ip}"))
  }

  run(:install_package,hosts,"make g++ openssh-server openmpi-bin openmpi-common openmpi-dev")
  f = file(localhost, "/home/cristian/Dev/C++/TLM_2013/tlm_clean_version.tar")
  distribute f, hosts, "/root/tlm_clean_version.tar"

  compile_tlm(hosts)
  log "Finished of setting up TLM"
  conf_mpi(hosts)
  copy "machines", hosts.first, "/root/TLMME/tlm/bin/machines"
  copy "run_cluster", hosts.first, "/root/TLMME/tlm/run_cluster"
  r = execute_one(hosts.first, "cd /root/TLMME/tlm/; chmod +x run_cluster")
  tlm_execution(hosts, RUNS, PIPE_LENGTH)

end
```

Listing 18: Experiment that measures the best performance of TLM code using *XPFlow*