HAL open science

Reconstructing the Software Environment of an Experiment with Kameleon

Joseph Emeras, Olivier Richard, Bruno Bzeznik

To cite this version:

Joseph Emeras, Olivier Richard, Bruno Bzeznik. Reconstructing the Software Environment of an Experiment with Kameleon. [Research Report] RR-7755, INRIA. 2011. inria-00630044

HAL Id: inria-00630044

https://inria.hal.science/inria-00630044

Submitted on 10 Oct 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Reconstructing the Software Environment of an Experiment with Kameleon

Joseph Emeras — Olivier Richard — Bruno Bzeznik

N° 7755

Octobre 2011

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--7755--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Reconstructing the Software Environment of an Experiment with Kameleon

Joseph Emeras , Olivier Richard , Bruno Bzeznik

Theme :
Équipes-Projets Mescal

Rapport de recherche n° 7755 — Octobre 2011 — 17 pages

Abstract: In the scientific experimentation process, an experiment result needs to be analyzed and compared with several others, potentially obtained in different conditions. Thus, the experimenter needs to be able to redo the experiment. Several tools are dedicated to the control of the experiment input parameters and the experiment replay. In parallel concurrent and distributed systems, experiment conditions are not only restricted to the input parameters, but also to the software environment in which the experiment was carried out. It is therefore essential to be able to reconstruct this type of environment. The task can quickly become complex for experimenters, particularly on research platforms dedicated to scientific experimentation, where both hardware and software are in constant rapid evolution. This article discusses the concept of the reconstructability of software environments and proposes a tool for dealing with this problem.

Key-words: Scientific experimentation, Environment, Reproducibility, Reconstructability, Data provenance.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Reconstruction des environnements logiciels des expériences avec Kameleon

Résumé : Dans le processus d'expérimentation scientifique, un résultat d'expérience requiert d'être analysé et comparé avec plusieurs autres, potentiellement obtenus dans des conditions différentes. Ainsi, l'expérimentateur a besoin de pouvoir refaire l'expérience. Plusieurs outils, dédiés au contrôle des données et paramètres d'entrées et au rejeu des expériences, existent déjà. Dans le domaine des systèmes parallèles, distribués, concurrents et des réseaux, les conditions de l'expérience ne se limitent pas à ses paramètres, il faut aussi considerer l'environnement logiciel dans lequel elle a été exécutée. Il est donc indispensable de pouvoir reconstruire cet environnement. Ceci peut vite devenir complexe pour les expérimentateurs, notamment dans les plateformes de recherche dédiées à l'expérimentation, où le materiel comme le logiciel évoluent très vite. Cet article propose une réflexion sur cette notion de reconstructibilité de l'environnement. Il propose ensuite un outil, dont le but est de répondre à cette problématique.

Mots-clés : Expérimentation, Environnement, Reproductibilité, Reconstructibilité, Provenance.

1 Introduction and Motivations

Research platforms dedicated to experimentation, like Emulab[1], Grid'5000 [5] and PlanetLab [6], offer to their users mechanisms allowing them to deploy and run their experiment on a set of nodes. In infrastructures such as Grid'5000, users can deploy their own software environment in which the experiment will be executed. This versatility gives experimenters a great flexibility but it is usually expensive. The user has to manage both his application and the environment in which it will be executed. In such a scientific experimentation context, it is necessary to ensure experiments results reproducibility. Without that, the experimenter cannot compare experiments [8, 1]. Indeed, it is not acceptable to draw a conclusion based on non-reproducible results, i.e. if the same experiment, run several times (several instances) in the same conditions, doesn't give the same result each time, or at least a result statistically comparable. Thus, the experiment in itself has to be reproducible and it is essential to be able to recreate its observation conditions because they will determine the results. These conditions are:

- the experiment instances parameters: their input data and configuration settings;
- the environment in which the experiment will be executed: software and hardware (taking into account its state at the run time).

The different parameters can be retrieved by using an experimentation plan, that log all the parameters used along the experiment instances, or by using an experiment conduct tool. The hardware environment condition requires that the experiment has to be run on the same set of nodes each time or at least on a set of nodes with identical characteristics. The problem of the hardware qualification before the experiment still remains. The user has to be sure that the same hardware is in the same state for each experiment. On the other hand, the software environment can be difficult to reconstruct. This environment reconstructability problem will be the main topic of this article.

For A.Iosup in [11], experiments reproducibility has two conditions: ensuring that the experiment can be run in the same conditions each time, and ensuring that the provenance of data used in the experiment (where data comes from) is known and that data can always be retrieved. Data provenance is a common problem in computing science [12]. More particularly in domains related to computing platforms such as Databases [4, 3] and Grid Services [13]. This article proposes to extend the data provenance concept by considering also data that is used to build the experiment software environment. This is justified by the fact that data (software, versions, configurations, and files) that make an environment determine some part of it and are necessary for its reconstruction, and so, to the results reproducibility of the experiment [8]. This article proposes to link data provenance to reconstructability.

This paper has presented the software environment reconstructability as an important part of results reproducibility. Indeed, software updates, hardware evolutions, platform reconfigurations damage software environment reconstruction and experiment reproducibility. Thus, it is difficult to make a software environment durable. A tool is necessary to solve this problem.

[1] http://www.emulab.net/pubs.php3

This article is organized as follows, in Section 2 we give more details on why it is necessary to be able to reconstruct a software environment and how data provenance has to be considered in this process. In Section 3, we introduce Kameleon, a tool that aims at easing the environment reconstruction. Then, in Section 4 we present the validation of our tool on the Grid'5000 platform through use cases taken from real users' experiences. In Section 5, we present a short state of the art of related tools. Finally in Section 6 we conclude and state some evolutions perspectives.

2 Problem Details and Background

In the experimentation process, the different instances of an experiment have to be run in the same software environment to ensure that the conditions are the same, thus allowing results comparison. Or so, the experimenter has to be able to know what changed and to control these changes between each run. Indeed, not all the software stack of an environment is implied in the result but only a subpart, depending on the experiment itself. If we ensure that the software part which determines the experiment behavior and thus its result is common to all of its instances, the results can be processed. Moreover, controlling what changed between several environments may be wanted by the experimenter for comparison purpose. He may want to create several versions of an environment, with different versions of a particular library, software or configuration and observe how his experiment behaves in this environment. Thus we have to be sure that the only differences between the environments were the ones wanted. In the research platforms dedicated to experimentation, both hardware and software are in constant rapid evolution. Thus, a software environment that is currently working and up to date may be obsolete very quickly, thus backing up the software environment and restoring it before each experiment run is not sufficient. The user custom software environment can become useless and will have to be created again, from scratch. In this case, the experimenter must have kept a trace of the environment construction to be able to reconstruct a new working environment, similar to the first version.

2.1 Environment Construction Steps

Generally, constructing an experiment software environment is composed of many short operations (usually shell commands), that are not linearly distributed over time. At the beginning, the user sets up the environment "basis" with a set of commands. This "basis" will be all the software stack needed by the experiment to run properly. E.g. the Operating System (OS) that will be installed on the nodes running the experiment, software required that have to be installed, their configurations, network and users setup. All these "basis" set up commands can be easily merged into one single script destined to reconstruct the environment later. However, this is generally not enough and requires a debug and setup stage. During this phase, the basis environment is modified by small patches and their integration in the global script gives a result hard to maintain and impossible to be reused by the users' community. Moreover, this method has the disadvantage of being less robust to errors. The risk of unlogged modifications to the environment, not reported in the final script, is significant.

Another drawback of this method is the lack of flexibility. The environment has to be generated directly on one of the nodes of the experimentation platform and thus monopolizes one node, generally for several hours just for the construction purpose.

2.2 Data Provenance

When reconstructing a duplicate environment, it is necessary to look at two matters. First, the environment creation process, i.e. the different steps that lead to the environment generation. These steps are the set of commands and related input data that allowed to setup the software environment and its configuration. Second, data contained in the environment have to be considered too. The reconstructed environment must contain the same data as the original one. This idea of data has to be as large as possible and has to take into account the files that can be present on the environment, but also installed software and their versions. This idea of data provenance, central in the reconstructability context, states the problem of software changes and evolutions in the packages repositories or direct download links. One cannot guarantee that a particular download link will exist forever or always point to the same software with the same version and build tag. It is the same thing for packages repositories whose softwares versions evolve along the updates. If the packages repositories are updated between two environment generations, the two environments will inevitably be different, and the experiment reproducibility may be affected. There is thus a need of a method that guarantees that both original and duplicated environments use the same data sources. That is what was previously identified as a data provenance problem.

2.3 Features Useful for Reconstruction

The reconstructability concept should also answer to several more technical problems, such as:

- **Debugging**. The availability to manually operate whenever during the environment creation has to be eased and easily traceable.
- **Composition of several environments parts.** To pool efforts, it should be possible for the users to share the whole or some parts of their environments to enable the experimenters' community to reuse them.
- **"Snapshotting"**, or restart after backup. Developing a software environment is rarely a simple and linear process. The possibility to restart from a former existing environment is a feature which brings a lot of comfort and saves a considerable amount of time to the user.

3 A Tool for Reconstructability: Kameleon

3.1 General Principle

Kameleon[1] is a tool that has been developed to answer the problem of software environment reconstruction. The general principle is simple: the Kameleon

[1] http://wiki-oar.imag.fr/index.php/Kameleon (Work In Progress)

engine enables composing chunks of scripts, boosted with special commands. Their goal is to mask some technical complexities to the user. Kameleon can thus be seen as a command sequencer, supplied with tools to ease some operations. During an environment (re-)construction, Kameleon will read the environment description and the stages that make up its creation in a "recipe" file described in YAML[1], a human friendly data serialization standard for all programming languages. YAML has been chosen due to its user-friendliness, its simplicity, and because it is implemented in several programming languages, easing the coupling between Kameleon and higher level tools. The "recipe" is made of two parts. First, the global variables definition part. Second, an ordered list, detailing the different steps of the environment creation. Each item in the list corresponds to a YAML "step" file, whose name will be unique in the whole recipe. Each step corresponds to one semantic action like installing a software, modifying a particular configuration file or recording the environment as a particular output format. A step is thus a set of commands leading to one or several technical actions. The sum of all the steps that compose the recipe will generate the software environment.

Inside a step, it is possible to define subparts. They will be used in the recipe, instead of calling the full step, the user can suffix the name of the step with one or several of its subparts, thus only these ones will be taken into account in the environment construction. The global variables defined in the recipe can also be accessed in the steps (by preceding their names with "$$").

In the global variables definition part of the recipe, a particular field is mandatory and reserved for Kameleon: "distrib". This field will be used by Kameleon to know which kind of OS will be setup in the output environment (e.g. debian, fedora, SL5... are possible values). As the different OS distributions can be very different, installation and configuration steps of a particular software may differ according to the OS type. Kameleon will use the steps corresponding to the right OS during the environment creation. Thus, with Kameleon, a recipe that creates a Debian virtual image can be easily transformed into one that creates a Fedora virtual image. The changes to make in the recipe would be to replace the "distrib" dedicated field in the header of the recipe from debian to fedora. The result will be an environment with the same software and configurations but whose OS will be Fedora.

The goal of the slicing between recipe and steps is to ease the code reuse and distribution. A step, once tested and validated, will be used in several recipes and by several users. The idea is that it is more efficient in a developers community, to pool their qualities, and that everyone works on a subpart of the problem, in his competence field. Kameleon tries to encourage this approach.
An other benefit of this approach is the ease of the process of generating several environments "flavors". If the experimenter needs three environments with three different versions of his application (or a particular library), he will create three different steps corresponding to the installation/configuration of these three versions. When generating the environments, just setting the corresponding step in the "recipe" file will allow Kameleon to generate the environment with the wanted application version. Thus the result will be three environments whose only differences are the versions of the experimenter's application.

[1]YAML Ain't Markup Language, see http://www.yaml.org

An example of a Kameleon recipe and some of its steps is presented in Figure 1. Here, the recipe will generate a basic "debian squeeze" environment in the "raw" output format (a generic format which is directly exploitable into a virtual machine manager or that can be converted later). Concerning the two highlighted steps, "bootstrap" will create the basis debian installation through the "debootstrap" command, which is a common method for installing a debian system. "kernel-install" will write a configuration file for the kernel to install, then install it.

The whole environment generation process using the recipe and steps files is shown in Figure 2. The Kameleon engine loads a recipe file and parses it to retrieve the step files called in the recipe. Then, the steps are processed in their order of appearance in the recipe. Finally, Kameleon will produce as output the environment generated and a cache area containing data used to build the environment. The reason of the presence of the cache area will be explained later in this article.

With such a method, The only modifications that have to be done in this recipe are changing the system installation part from Debian to Fedora. This is done by replacing in the recipe the few steps responsible for the OS installation from Debian to Fedora (both are provided with Kameleon). In our example of modifying the recipe to get a Fedora instead of a Debian distribution, the changes to make to the recipe would be: replacing the "distrib" field located in the "global" variables part, in the header of the recipe from debian to fedora. This will make Kameleon use the Fedora "`bootstrap`" and "`kernel-install`" steps that are specific to the system installation instead of the Debian ones. Basically, just replacing "debian" to "fedora" in the recipe and running Kameleon with this recipe would create a second image with the same software installed and in the same output format. When setting the "distrib" field in the recipe, the user tells to Kameleon where to find the steps to use in the steps hierarchy. Thus, Kameleon will automatically load the steps corresponding to the distribution specified in the header of the recipe and will execute them.

3.2 Execution Contexts

Kameleon steps are composed of a set of shell commands prefixed with specific "Kameleon-commands". Their role is multiple.
Some of the "Kameleon-commands" enable the Kameleon engine to know in which context the corresponding action has to be executed. This notion of context is important here. It illustrates the isolation between the machine running Kameleon (host) and the environment being created by Kameleon. Depending on the goal of the step, the action to execute can be run in the environment itself or in the context of the machine that runs Kameleon. A typical example of the first case is a software installation. If the action is the installation of a particular software in the environment, the installation context has to be the environment and not the machine running Kameleon. Indeed, the goal here is certainly not installing the software on the host. An example for the second case is the following: a file has to be copied from the host File System to the environment. Here, the copy of the file has to be done in the context of the host in order to have access to the file. Due to the isolation mechanism between the host and the environment, the environment cannot access the host File System. The Kameleon tool uses the "chroot" isolation method (system

Recipe sample: Creation of a KVM Debian image

```
# ### debian.yaml Kameleon recipe sample ###
global:
  distrib: debian
  # Debian specific
  debian_version_name: squeeze
  distrib_repository: http://ftp.fr.debian.org/debian/
  #
  # Architecture
  arch: amd64
  kernel_arch: "amd64"
  #
steps:
  - bootstrap
  - system_config
  - root_passwd
  - mount_proc
  - kernel_install
  - strip
  - umount_proc
  - build_appliance:
    - clean_udev
    - create_raw_image
    - create_nbd_device
    - mkfs
    - mount_image
    - copy_system_tree
    - install_grub
    - umount_image
    - save_as_raw
  - clean
```

Debian Kernel installation step

```
kernel_install:
 - kernel-img_conf:
  - write_file:
   - /etc/kernel-img.conf
   - |
     do_symlinks = yes
     relative_links = yes
     do_bootloader = yes
     do_bootfloppy = no
     do_initrd = yes
     link_in_boot = no
 - kernel_install:
  - exec_chroot: bash -c "DEBIAN_FRONTEND
    =noninteractive apt-get -y --force-yes
    install linux-image-$$kernel_arch"
```

Debian basis installation step

```
bootstrap:
 - debootstrap:
  - exec_appliance:  debootstrap --arch=
    $$arch $$debian_version_name
    $$chroot/ $$distrib_repository
```

Figure 1: Kameleon recipe and steps example.

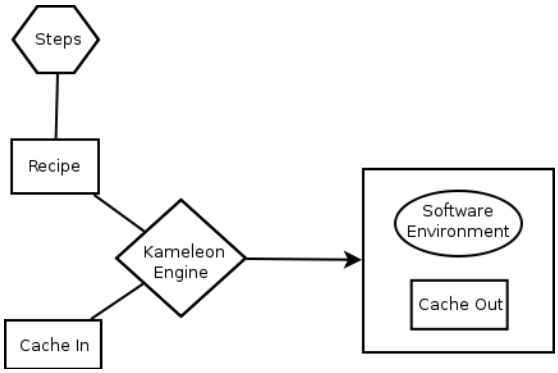

Figure 2: Environment generation with Kameleon.

root change) for all the actions that have to be executed in the environment context. This method has been chosen because it is a common technique for Linux distributions installation bootstrap.

The others special commands provide abstraction methods to elementary operations. They allow to:

- include another step file. Everything described in the included step will be executed;
- verify the presence of a particular command (in the environment or in the host context);

- manipulate files (append, write);
- define local variables. Instead of the ones defined in the recipe, that are global, these have only a local (step-level) scope;
- get a shell (with the possibility to invoke a particular step by its name).

3.3 Provided Ingredients

Kameleon is distributed with a set of recipes and steps. Their goal is to provide basic bricks and examples for the user and help him developing his owns. Among provided steps, Kameleon offers environment export methods in several output formats. Currently, the output formats are VirtualBox[1], Xen[2], KVM[3], disk image, Grid'5000[4] image and archive. This allows the user to generate his environment on the format he needs, and thus, to operate the environment creation on an other machine than the one dedicated to the experiment.
A virtual appliance, configured to use Kameleon, is available for download on the official website. A Kameleon recipe is dedicated to generate this appliance. Kameleon also provides a recipe and its steps to generate a Debian environment. RPM packages based distributions are under development.
The goal of providing all these "ingredients" with Kameleon is to enable the user to easily reuse what have been done by the community. A dedicated repository is available to share the steps and recipes between Kameleon users. Users are obviously welcome to submit their own steps and recipes for them to be integrated in the repository. Of course, a lot of steps are very specific to a particular platform or OS but this problem is inherent to dealing with multi-platforms and OSes during the experiments and no simple method will work for all of them. The most convenient, maintainable and comprehensive method is thus to provide many little specific basic bricks, easily adaptable for particular cases. Thus, the Kameleon tool helps a lot in organizing the environment construction. Besides, all the steps that are not platform specific can be directly used in several recipes. This is a substantial saving in terms of time.

3.4 Snapshotting

Kameleon provides a way to use recipe snapshotting (restart after backup). In the recipe, it is possible to specify an archive, that will be used as a base for the environment construction. This technique is particularly used during the iterative phase of the recipe/steps development. Instead of restarting from scratch each time, Kameleon reuses what has been validated as working, and being backup. The Kameleon engine loads the snapshot archive and will set it as the new environment context (in the chroot). Figure 3 illustrates how snapshotting can be used to shorten the image generation process by reusing what is known as working. In this example, a first environment "E1" has been built with Kameleon and the "Recipe 1". Then, from this environment archive, the user can create a second "E2" environment, based on "E1". The new environment

[1] http://www.virtualbox.org
[2] http://www.xen.org
[3] http://www.linux-kvm.org
[4] https://www.grid5000.fr/mediawiki/index.php

will contain everything that was in "E1" plus what was described in the "Recipe 2".

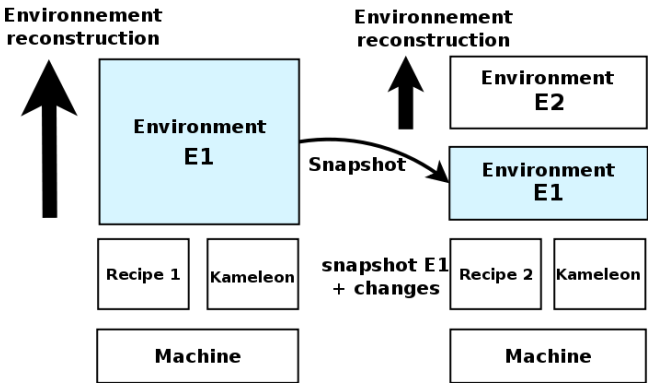

Figure 3: Kameleon snapshot feature.

3.5 Data Provenance

Later Kameleon developments tends to bring an answer to the data provenance problem. In the recipe, Kameleon allows to specify a cache area and an associated policy. This policy will allow the user to define if data accessed during the environment creation will come from the cache, from outside, or both. Thus, after the environment generation, Kameleon will have in its output zone on the machine that ran the recipe: the environment itself and the cache being used for its creation as shown in Figure 2.

Kameleon provides two methods to use this caching idea. First, at launching, the tool configures an HTTP proxy to cache all the files retrieved through this protocol (which represents most of outside access in a common Kameleon run). Thus, at the end of the Kameleon run, every data accessed through HTTP during the environment creation will be available in the output cache zone. For other protocols, a command allows to specify a file and the way to retrieve it, if not present in the cache. When this command is called in a step, data will be either read from the cache or retrieved from the outside and then fill the cache (according to the cache policy and contents). So basically, it is possible to cache every data that have been used in the environment construction. The user will decide which data is important for the reconstruction and thus needs caching. An interesting problem here, is data with restricted distribution (data that should not be distributed for various reasons like private or classified data). Such data retrieved during the environment creation can end into the cache automatically. This is a problem because when the user will want to redistribute his environment and its associated cache, the data that should be considered as "private" (e.g. ssh keys) will be found in the environment cache. To solve this problem, Kameleon offers to specify it as a parameter in the previously described command, and will split the cache in two parts: "public" and "private". The user will thus decide which data is "private" and which is "public". Kameleon offers a method allowing to export the whole recipe, steps, cache and environment

generated, for redistribution. This takes into account the private data and, instead of exporting data itself, only the data fingerprint and metadata will be exported. This allows to know if data that composed the environments during the different experiments was the same.

4 Use Cases

All the cases presented here are taken from **real users experiences** on Grid'5000 (in short: G5K). This platform has been chosen to validate our approach for its flexibility and its richness. An advanced use of Grid'5000 often requires from the users to deploy their own software environment on a set of nodes. Such a situation often occurs and thus illustrates the gain in using Kameleon for a repetitive action. Grid'5000 provides a wide range of use cases, allowing to illustrate several points of interest of Kameleon. However, Kameleon is absolutely not linked to Grid'5000 in any way and aims at being used on any system where the user can install his own software stack, from the simple desktop machine to the reconfigurable computing platform or even on embedded systems. The use of Grid'5000 in this section is only made for validating the Kameleon approach. Grid'5000 is a scientific tool dedicated to the study of large scale parallel and distributed systems. Geographically distributed over 9 sites in France and 1 in Brazil, it aims at providing a highly reconfigurable, controllable and monitorable experimental platform to its users. This platform has the particularity to allow them to deploy their own software environment, which makes it perfectly adapted to the Kameleon use.

Figure 4 presents the environment deployment process on Grid'5000 (here, we don't need Kameleon, this figure only illustrates the deployment process). The user reserves a node, once it is allocated the software environment is sent to the node through a specific tool. Basically, this tool copies the environment archive to a dedicated disk partition, which is bootable and then the node reboots on this partition. Once this is done, the user can connect the deployed node through ssh with the root account. Several default environments are available on Grid'5000 and can be deployed. The user can both deploy his own environment or a default one.

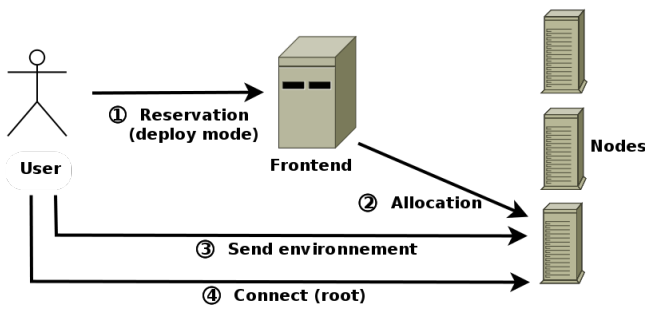

Figure 4: Environment deployment on Grid'5000.

Figure 5 shows the typical usage of a deployed environment customization on G5K without the use of Kameleon. Once the environment is deployed on the node and the node has rebooted on this environment, the user connects to the node and can modify the deployed environment. When the modifications are terminated, the new environment has to be saved for a further deployment.

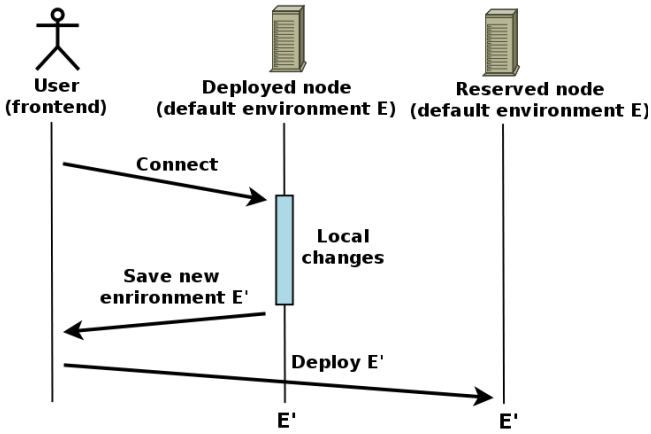

Figure 5: Regular environment customization on Grid'5000.

In the Figure 6, the process is completely different. The user starts from a default environment already available for deployments on Grid'5000, loads it thanks to the Kameleon snapshot feature and then modifies it locally before saving it. The great benefit here, is that the user doesn't need to reserve a node of the platform for the environment customization but does it locally, on his workstation and the platform is not required during this phase.

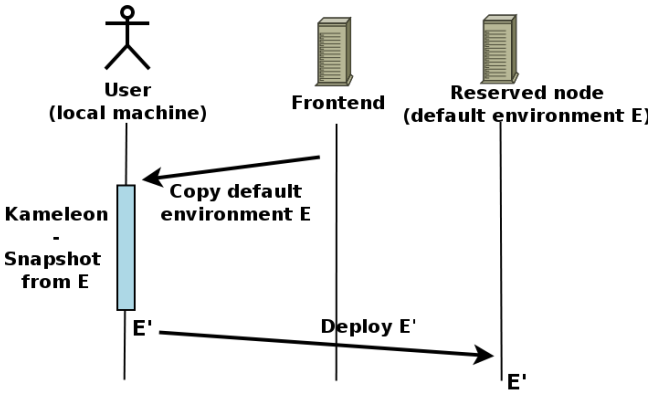

Figure 6: Environment customization on Grid'5000 with Kameleon.

The Figure 7 shows another different use case: here the user creates from scratch his own environment, and then saves it for a later deployment. This might be very useful if the user needs another Operating System (OS) that the

ones provided for deployment. E.g. on Grid'5000 there are no base images provided with the SL5 (Scientific Linux 5.0) OS. This OS is required for installing a gLite[1] grid middleware. Thus for testing purpose of the gLite software, the user might need Kameleon to construct from scratch the SL5 OS image required.

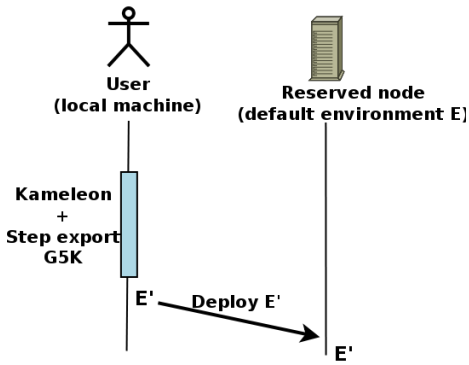

Figure 7: Environment creation on Grid'5000 with Kameleon.

The Figures 8 and 9 present a typical usage where Kameleon allows the user to take control over what changes between two environment reconstructions. Let's take the case of a platform upgrade. New network cards are installed in the nodes, providing an other type of network (i.e. Infiniband) to the users. In such case, the deployable environments provided by the G5K technical team are changed: a new kernel is required to support the new device. These environments are reconstructed by the administrators, leading to a set of complete new environments with new software versions.

In Figure 8, the user has customized, built and installed several softwares and saved his own environment. He got several results from his experiments. When the platform is upgraded, the new network cards are not supported by the old kernel of the user's environment. His custom environment has no network access and thus is unusable. The user will have to recreate it from the new updated environment provided by the technical team. He has to reconfigure it and spend a lot of time to make it work as previously. But worst, now his new custom environment is different from his former one. Software versions differ (and particularly gcc and all the compiling tools) and he might have troubles when building the softwares his experiments requires. Moreover, the user is not totally sure about the validity of the comparison between his previous experiments (in the first custom environment) and the ones done in the new environment, the installed software having different versions.

In such a situation, Kameleon is very helpful. Figure 9 shows how a user can take advantage from the Kameleon snapshot mechanism. Here, the user has customized his environment with Kameleon from a Grid'5000 default environment. He has loaded this default environment in Kameleon, modified it according to his needs and saved it. After the network cards upgrade on the machines, the user will be able to update locally (on his own computer) only the kernel part of his environment by loading it in Kameleon and applying it a recipe that up-

[1] Lightweight Middleware for Grid Computing (http://glite.cern.ch)

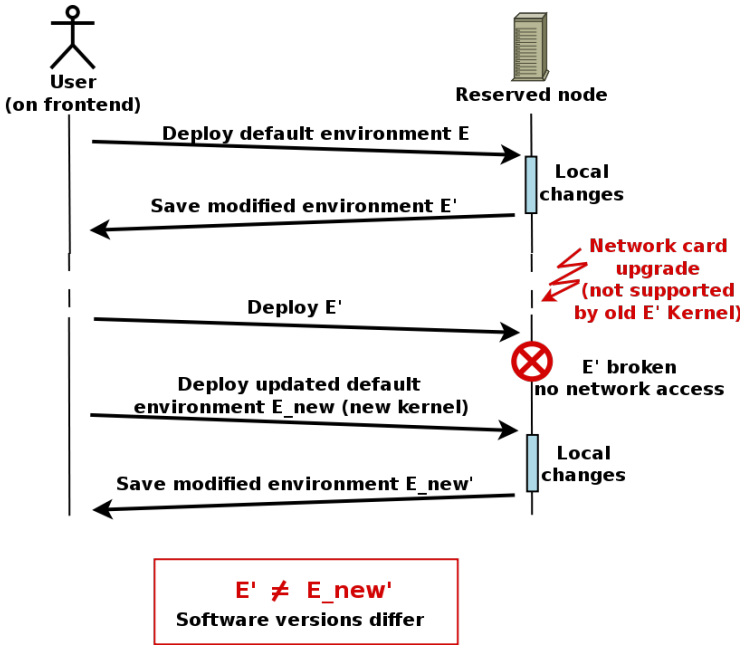

Figure 8: User Environment aging problem – without Kameleon.

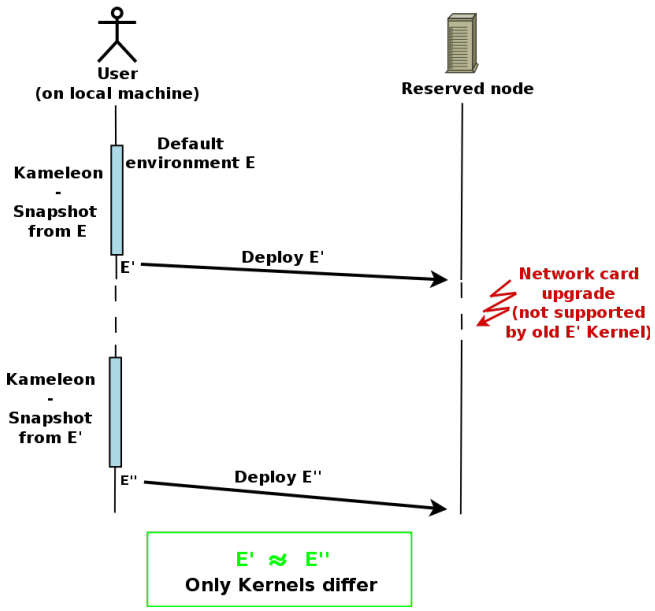

Figure 9: User Environment aging problem – with Kameleon.

grades the kernel. Thus, with this solution, the new environment can access the network and only the kernel will differ between the original environment and

the new updated one. The results obtained with the old environment have less chances of being trashed because of software version incompatibility.

5 Related Works

Several configuration automation and systems administration software currently exist. Among the most recognized, there are CFEngine[1], Puppet[2], and Chef[3]. Historically, Puppet came after CFEngine and has been strongly inspired from it. More recently, Chef takes up their concepts and offers an Open Source solution. Their goal is to ease the setup, the administration and the maintenance of an infrastructure. These 3 tools use a Domain-Specific Language (DSL)[10], a dedicated language, relative to an application domain, to describe an environment software configuration. This type of language allows to express, in an abstract way, thanks to an executable language, all the semantic of the domain it refers to. This is both an advantage because it allows to define simple programs, easy to understand, but also a major drawback, the effort to master it at start, is considerable. Thanks to their DSL, these tools allow the infrastructures administrators, for a given machine (or type of machine), to describe which applications and services will be present and what will be their configurations. Then, the automation software engine will allow to install and configure this machine, as well as managing its potential software updates. For more information concerning the use of a DSL in these tools, a substantial study is presented in the technical report [9]. A state of the art and a global comparative study of such tools is proposed in [7]. These tools allow to answer partially to the experiment environment reconstructability problem. Indeed, they propose to describe, in a machine administrator point of view, a software environment. However, the real goal of these tools is not reconstructability. Thus, they are not really convenient in that case. The use of a DSL restricts the versatility, what can be expressed is determined by the tool itself and its DSL. These tools are focused on "configuration" for administration, and not for experimentation. Being designed to manage a platform and not generate software environments from scratch, the system creation bootstrap is a problem. What can be done in such a case, is the environment configuration part once the basis system is already installed. The user has to initiate this part in an other manner, manually or by a script, but in a external way. Moreover, these tools are quite heavy, in terms of infrastructure, to set up, and work on the Client/Server mode. Only Chef proposes a lighter version: Chef-Solo, which is autonomous. Finally, the data provenance problem is not taken into account with such solutions.

A more interesting tool, which is closer to our problem is CDE [2]. The idea is to allow users to eliminate the dependencies problem during the installation of a software or a library. To do this, CDE proposes to catch on the fly all the accessed files (binaries, libraries, configuration files) during a command execution, and to provide this set as a CDE package. This package can be redistributed and run on an other machine, as long as it has the same architecture (e.g. x86) and the same major kernel version (e.g. 2.6.X). More complete than the static links mechanism, which only allows a program to be linked to its libraries, this

[1] http://www.cfengine.org
[2] http://www.puppetlabs.com
[3] http://www.opscode.com/chef

tool allows to capture all the environment necessary to the execution of the command. In fact, related to the reconstructability problem, it can be said that this tool captures the state of the software before the experiment. This solution is incomplete for the problem since it doesn't capture all the experiment environment, but proposes an interesting approach. A CDE archive is more portable than a full system archive (as in Grid'5000) which is dependent on the physical machine construction (i.e. disks partitions, network cards configurations), it only depends on the kernel system calls. But with this solution, the construction plan of the environment is not kept as it is in Kameleon, with the recipe and steps. Moreover, the CDE mechanism doesn't allow to treat fully the data provenance problem neither.

6 Conclusion and Perspectives

This article stated one of the major problems encountered during scientific experimentation, and more particularly in the research platforms dedicated to this domain: experiments and results reproducibility. This paper explained the reproducibility conditions, and identified a particular notion related to reproducibility: the software environment reconstructability concept. This article combined to this the idea of data provenance, an essential counterpart to experiments reproducibility. Then, a tool to solve these problems is proposed and some use cases of this tool are given. Finally, this article stated several tools that can be possible alternatives and explained their advantages as well as their limitations.

Kameleon is a recent tool being actively developed and maintained. Several future functionalities are under development. First, there is a need of recipes for more Linux distributions in order to provide most of the OS used in the High Performance Computing (HPC) domain. The HPC tools and techniques study is a central priority in research platforms dedicated to experimentation. Thus, it is necessary to provide most of the possible systems configurations to experimenters. Then, steps for distributed and parallel File Systems (as Lustre[1]) are in testing stage. The parallel File System is a central part in HPC. Because of that, there is a real need of such steps to help the experimenters to easily setup their own HPC configuration. Other research oriented recipes and steps, dedicated to Resources and Jobs Management Systems benchmarks and tools are also available.

References

[1] *Final Report of the NSF Workshop on Challenges of Scientific Workflows*, National Science Foundation, Arlington, VA, December, 2006.

[2] *CDE: Run Any Linux Application On-Demand Without Installation.* Proceedings of the 2011 USENIX Large Installation System Administration Conference (LISA), 2011.

[3] P. Buneman, S. Khanna, and W.-C. Tan. Data provenance: Some basic issues. In S. Kapoor and S. Prasad, editors, *FST TCS 2000: Foundations*

[1] http://wiki.lustre.org

of Software Technology and Theoretical Computer Science, volume 1974 of *Lecture Notes in Computer Science*, pages 87–93. Springer Berlin / Heidelberg, 2000.

[4] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and where: A characterization of data provenance. In J. Van den Bussche and V. Vianu, editors, *Database Theory - ICDT 2001*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer Berlin / Heidelberg, 2001.

[5] F. Cappello, E. Caron, M. Dayde, F. Desprez, Y. Jegou, P. Primet, E. Jeannot, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, B. Quetier, and O. Richard. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, GRID '05, pages 99–106, Washington, DC, USA, 2005. IEEE Computer Society.

[6] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Comput. Commun. Rev.*, 33:3–12, July 2003.

[7] T. Delaet, W. Joosen, and B. Vanbrabant. A survey of system configuration tools. In *Proceedings of the 24th international conference on Large installation system administration*, LISA'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[8] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24 –32, 2007.

[9] S. Gunther, M. Haupt, and M. Splieth. Utilizing internal domain-specific languages for deployment and maintenance of it infrastructures, fin-004-2010. Technical report, University of Magdeburg, School of Computer Science.

[10] P. Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 134–, Washington, DC, USA, 1998. IEEE Computer Society.

[11] A. Iosup. *A framework for the study of grid inter-operation mechanisms.* PhD thesis, Delft University of Technology, 2009.

[12] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34:31–36, September 2005.

[13] M. Szomszor and L. Moreau. Recording and reasoning over data provenance in web and grid services. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 603–620. Springer Berlin / Heidelberg, 2003.

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399