Reproducible Software Appliances for Experimentation

Cristian Camilo Ruiz Sanabria, Olivier Richard, Joseph Emeras

Reproducible Software Appliances for Experimentation

Cristian Ruiz[1],Olivier Richard[1], Joseph Emeras[2]

[1] INRIA Grenoble, Grenoble France.
[2] INRIA Nancy, Nancy France.
{cristian.ruiz,olivier.richard,joseph.emeras}@imag.fr

Abstract. 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 paper we present our approach oriented to setup complex environments for experimentation, environments that require a lot of configuration and the installation of several software packages. The main objective of our approach is to enable the exact and independent reconstruction of a given software environment and the reuse of code. We present a simple and small software appliance generator that helps an experimenter to construct a specific software stack that can be deployed on different available testbeds.

Key words: Reproducible Research, Testbed, Virtual Appliances, Cloud Computing, Experiment Methodology.

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 [8]. This is due to the numerous details that have to be taken into account. The process of repeating an experiment was carefully studied in [7] and among the many conclusions drawn, the difficulty of repeating published results was highly relevant.

With the advent of testbeds such as Grid'5000[5] and FutureGrid[15], cloud-based testbeds like BonFIRE[1], the ubiquity of Cloud computing infrastructures

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

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 [12] the author gives 13 ways that replicability is enhanced by using virtual appliances and virtual machine snapshots. Another close approach is shown in [9] 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 [2], which relies on the use of virtual machines and aims at improving the interactions between authors, reviewers and readers with reproducilibity purposes.

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 this paper, we present our approach to reproduce a software environment for experimentation. The approach is based on a software appliance generator called Kameleon. We present the implementation of a persistent cache mechanism that stores every piece of data (e.g., software packages, configuration files, scripts, etc.) used to construct the software appliance. It presents a lightweight approach which enables the construction and exact post reconstruction of a given software appliance from text descriptions. 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.

This paper is structured as follows: In Section 2, some approaches to reproduce a given environment for experimentation are discussed. Then, our approach to set up the environment required for experimentation is described in Section 3. In Section 4, we show some experimental results and validation of our approach. Finally the conclusions are presented in Section 5.

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.

2.1 Tools for capturing the environment of experimentation

CDE[11] and ReproZip[6] 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.

2.2 Methods for setting up the environment of experimentation

Manual The experimenter deploys a *golden image* 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 with testing purposes. Possibly, the experimenter has to deal with the contextualization 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 [9] and [2]. 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 [1] 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*[2] or *Puppet*[3]) 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 [14] it is shown the viability of reproducible eScience on the cloud through the use of configuration management tools. A similar approach is shown in [3].

Software appliances Experimenters can opt for software appliances that have to be contextualized at deployment time. In [13] the viability of this approach was shown. Those images can be either built or downloaded from existing testbed

[2] http://www.opscode.com/chef/
[3] https://puppetlabs.com/

infrastructures (e.g Grid'5000, FutureGrid) or sites as TURNKEY [4] or Cloud market[5] 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. The process of image building relies on widely available tools that will be analyzed in the next subsection.

2.3 Software appliances builders

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.

Vagrant[6] and Veewee[7] are complementary tools to create and configure lightweight, reproducible, and portable development environments. Veewee automatically builds virtual machine images of different Linux distributions. Those images can be exported as so called *Boxes* that are run on top of the most popular virtualization technologies (e.g., VirtualBox, VMware, etc.). Vagrant provision these *Boxes* using industry-standard configuration management tools such as shell scripts, Chef or Puppet that will automatically install and configure software. The idea of Vagrant is the creation of disposable and consistent environments that can be re-built from scratch. BoxGrinder [8] creates appliances from simple plain text descriptions for various platforms. Unlike previous tools, it uses the host system to perform the image creation which results in a faster process. Those tools are widely used in Cloud infrastructures for generating customized virtual appliances. In theory any experimenter could reconstruct the virtual appliances using the same tool and the same specifications provided by other experimenter. However, the main hurdle is the dependency on external repositories, for instance, 30% of Veewee definitions files point to repositories that not longer exist or some packages are missing for a complete installation.

[4] http://www.turnkeylinux.org
[5] http://thecloudmarket.com/
[6] http://www.vagrantup.com/
[7] https://github.com/jedi4ever/veewee
[8] http://boxgrinder.org/

3 Reproducible software appliances

We extended our previous work Kameleon[10] which is a very simple software appliance generator that enables the construction and exact post reconstruction of a given software appliance from text descriptions. It is targeted to make easier the reconstruction of custom software stacks in HPC, Grid, or Cloud-like environments. Kameleon takes care of the following steps in the process of software appliance generation:

- **Operating system**: Construction of the respective *O.S* file system layout, which encompasses the necessary binaries, libraries, configuration files in order to run. This depends on the distribution chosen for the software appliance.
- **Provision**: Installation of different software packages required for the appliance. This can be done through the package manager of the distribution chosen, from source tarballs, or using configuration management tools.
- **User's code**: This step will add user's modifications or applications that the user wants to experiment with.
- **Save output**: Save the generated image into a particular format: Virtual machine format, LiveCD, raw disk image, etc.

Kameleon approach is based on two contexts, namely *execution context* which is where Kameleon engine is executed (e.g., user's machine) and *construction context* in charge of generating the file system layout of the appliance. Kameleon can use different operating system-level virtualization techniques such as: *chroot* (the less isolated but the lightness one) or *Linux Containers* as well as full virtualization (*e.g., VirtualBox, kvm*) and real machine (the most isolated but the heaviest one). Each context has its own advantages and disadvantages. As exposed before, using the user's machine to build the appliance could result in a faster build process. Kameleon enables to take advantage of the most convenient approach given the user's requirements. The process of construction or reconstruction has to take care of some possible issues caused by, for example, isolation and portability. Special needs can be specified in Kameleon metadata.

Our previous work was extended mainly in two points: (1) Requirements for a reproducible software appliance were identified, (2) The implementation of a persistent cache mechanisim. Both points will be described next.

3.1 Requirements for a reproducible reconstruction

The approach for software appliance construction and reconstruction is based on four requirements:

1. A recipe (Fig 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. For more details [10]

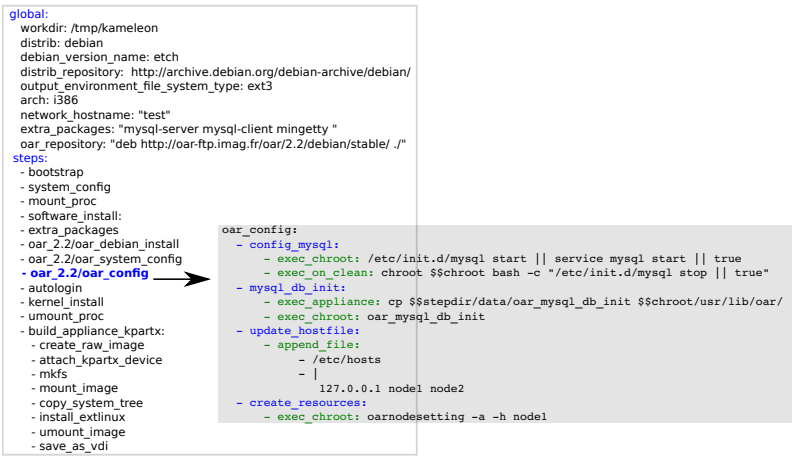

Fig. 1: Recipe and step example

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 paper, it will refer to this.
3. Kameleon engine consist in 700 lines of ruby code which parses the recipe and carry out the building. This part includes as well the persistent cache mechanism that will be described later on. This is the user interface to Kameleon.
4. Metadata that describes the compatibility and requirements between *execution context* and *construction context*.

Therefore, the problem of guaranteeing the exact reconstruction of software appliances is reduced to keeping the parts of Kameleon unchanged: (1) the recipe, (2) DATA (3) Kameleon engine. Two different experimenters having those three exact elements and fulfilling the requirements of context interactions (4) will generate the same software appliance. Kameleon can generate in an automatic and transparent way a cache file 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 low size of Kameleon engine and Polipo (less than 1MB) makes feasible the distribution of the exact versions used to create the environment, avoiding the incompatibility between versions. The whole process is depicted in Fig 2. More information can be found in [10] or in Kameleon web site[9].

3.2 Persistent cache mechanism

Our approach to achieve replicability is to use a persistent cache to capture all the *DATA* used during the construction. As we cannot guarantee that a

[9] kameleon.imag.fr

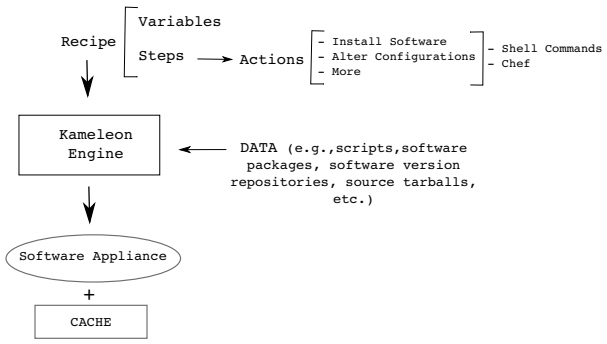

Fig. 2: Software appliance creation with Kameleon

particular download link will exist forever 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.

Design The caching mechanism has to be transparent and lightweight for the user in the two phases of the Kameleon approach: 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 parts of the *DATA* missing, 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[10] which is a very small, portable and lightweight caching web proxy. We chose Polipo because it can run with almost zero configuration.
- Ad hoc procedures that cache what could not be cached using the caching web proxy (e.g., version control repositores, https traffic) and all data from the local machine. These Ad hoc procedures are based on simple actions depending on the data to cache. Modifications on the fly of the steps involved on those Ah doc procedures are necessary.

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*.

[10] http://www.pps.jussieu.fr/ jch/software/polipo/

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,$\bar{C}$: 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,$\bar{C}$: it comes form local machine but can not be cached.

4 Experimental results and validation

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

OAR Appliances			
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
2.5.0	5 Dec 2011	Debian Squeeze	140
2.5.2	23 May 2012	Debian Squeeze	140

Table 1: Software appliances generated

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 1 that consist in different flavors of GNU/Linux (Debian, Ubuntu) and middleware: OAR[4] a very lightweight batch scheduler, Hadoop[11] and TAU[12]. It was possible to reproduce old environments of test back to 2009. A design goal was to achieve a self contained cache. Hence, we 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.

4.1 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

[11] http://hadoop.apache.org/

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

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.

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. All the examples presented in this paper can be reproduced accesing the Kameleon site[13].

5 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 paper, 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 show 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.

6 Acknowledgments

Experiments presented in this paper were carried out using the Grid'5000 experimental testbed, being developed under the INRIA ALADDIN development action with support from CNRS, RENATER and several Universities as well as other funding bodies (see https://www.grid5000.fr).

References

1. 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.

[13] http://kameleon.imag.fr/

2. Grant R. Brammer, Ralph W. Crosby, Suzanne Matthews, and Tiffani L. Williams. Paper mch: Creating dynamic reproducible science. *Procedia CS*, 4:658–667, 2011.
3. 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.
4. 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.
5. 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)*, November 2005.
6. 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.
7. 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.
8. A.P. Davison. Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science Engineering*, 14(4):48–56, 2012.
9. Joel T. Dudley and Atul J. Butte. In silico research in the era of cloud computing. *Nature Biotechnology*, 28(11):1181–1185, November 2010.
10. 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.
11. 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.
12. Bill Howe. Virtual appliances, cloud computing, and reproducible research. *Computing in Science and Engg.*, 14(4):36–41, July 2012.
13. 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.
14. J. Klinginsmith, M. Mahoui, and Y.M. Wu. Towards reproducible escience in the cloud. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 582–586, 2011.
15. G. von Laszewski, G.C. Fox, Fugang Wang, A.J. Younge, A. Kulshrestha, G.G. Pike, W. Smith, J. Vockler, R.J. Figueiredo, J. Fortes, and K. Keahey. Design of the futuregrid experiment management framework. In *Gateway Computing Environments Workshop (GCE), 2010*, pages 1–10, 2010.