Towards Trustworthy Testbeds thanks to Throughout Testing

Lucas Nussbaum

▶ **To cite this version:**

Lucas Nussbaum. Towards Trustworthy Testbeds thanks to Throughout Testing. REPPAR - 4th International Workshop on Reproducibility in Parallel Computing (with IPDPS'2017), Jun 2017, Orlando, United States. pp.9. hal-01538682

HAL Id: hal-01538682
https://inria.hal.science/hal-01538682

Submitted on 13 Jun 2017

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.

Towards Trustworthy Testbeds thanks to Throughout Testing

Lucas Nussbaum

Inria, Villers-lès-Nancy, F-54600, France
Université de Lorraine, LORIA, F-54500, France
CNRS, LORIA - UMR 7503, F-54500, France
Email: lucas.nussbaum@loria.fr

***Abstract*—When using testbeds in the context of experimental computer science, the ability to produce trustworthy and reproducible experiments results depends greatly on the trustworthiness of the infrastructure itself. Unfortunately, several factors many issues such as software misconfiguration, hardware heterogeneity, or service failures, can remain undetected and affect the quality of experimental results. This paper presents the design and implementation of an automated testbed testing framework. This framework was deployed in the context of the Grid'5000 project, and uncovered more than one hundred of issues.**

***Keywords*-experimentation; testbeds; automated testing; regression testing; reproducible research**

I. Introduction

Reproducibility has been the focus of a lot of interest over the recent years, in science in general, and in computer science specifically. But most of this focus has been targeted at the *reproducibility of data analysis*, which is usually handled by a pipeline [1] of several steps involving various tools, starting from *measured data* and going up to *figures* and *tables* included in an article. The various steps of that pipeline involve code for pre-processing the measured data, data analysis, and data presentation. However, this focus on *reproducibility of data analysis* ignores the important question of how *measured data* is produced.

Experimental computer science generally involves two main methods to acquire data about *systems under test*: simulation, and experimentation on testbeds. Experimenting on a testbed is a challenging task, and usually involves many different tools: the testbed itself, of course, but also experiment orchestration solutions [2] ranging from shell scripts to complex frameworks, load or failure injectors, emulation solutions, measurement tools, etc. Each of those components has a huge impact on the experiment and the results that will be obtained from it. In theory, experimenters should include a qualification and calibration phase in their experiments, and confirm that this whole stack meets its specification. But unfortunately this is very rarely done in practice, probably due to lack of time or adequate tools.

Still, assessing the correctness of software, e.g. with software testing techniques, is a relatively well-understood process [3]. But the bottom layer of the stack, that is, the testbed itself, raises specific challenges: testing infrastructure is an entirely different story. The complex mix of software and hardware, deployed at scale, provides potential for many difficult-to-detect issues, such as hardware misconfigurations or failures, or software bugs that happen randomly or only at scale.

This paper describes work that was carried out in the context of the Grid'5000 testbed in order to systematically test the infrastructure and its services, with the goal of increasing the reliability and the trustworthiness of the testbed by uncovering problems that would otherwise harm the repeatability and the reproducibility of experiments.

The paper is structured as follows. Section II provides the necessary context about the Grid'5000 testbed. Section III details motivations for this work, and specific challenges that were encountered. Then, Section IV describes the solution that was implemented, before some results are discussed in Section V. Finally, related work is presented in Section VI before the paper is concluded in Section VII.

II. Context: the Grid'5000 testbed

This section provides some background information about the Grid'5000 testbed, and about the way it is being operated, in order to support the design choices explained later.

The Grid'5000 project was initiated in 2003 with the goal to provide a testbed to experiment on Grid computing. The focus later moved to become a versatile testbed serving other areas of distributed computing (P2P, HPC, Cloud, Big Data, networking). The testbed itself was open to users in 2005. Each year, the testbed sees about 550 active users (users making at least one resource reservation) that produce about 100 publications.

Grid'5000 is currently composed of 8 sites (Figure 1) located in France and in Luxembourg. Each Grid'5000 site is composed of one or more sets of homogeneous machines called *clusters*. All machines from the same *cluster* are usually bought at the same time. At the time of writing, Grid'5000 has a total of 32 clusters, composed of a total of 894 nodes. At the networking level, all sites are interconnected with a dedicated 10 Gbps backbone network.

Some specific hardware is also available on Grid'5000: various generations of HPC networks (mostly Infiniband),

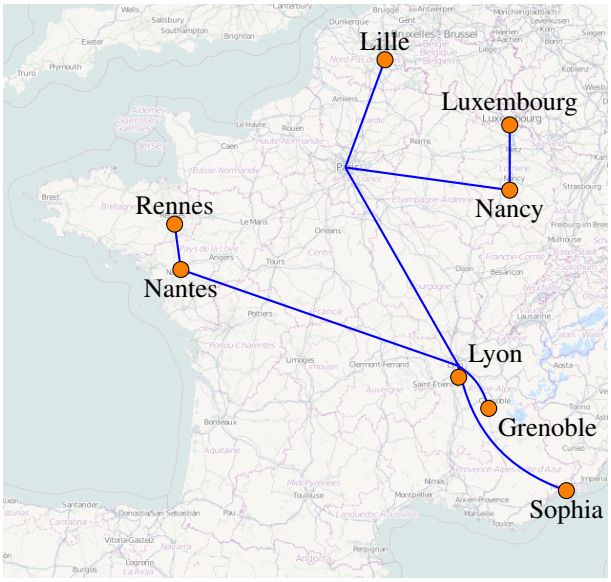

Figure 1. Grid'5000 sites and interconnection network

of GPUs, and Xeon Phi co-processors.

The main Grid'5000 features and services are:

- *Resources description and discovery*: a REST API (called the Reference API) provides detailed information about all resources in the testbed. Each time a node boots, its description is verified by a tool called *g5k-checks* that collects information using various inventory tools [4], to ensure that the Reference API contains correct and up-to-date information.
- *Resources reservation*: a resource manager (OAR [5]) is used by users to reserve resources for a specified duration. A rather complex Usage Policy[1] ensures fair sharing of resources between users during the day, while large reservations (generally made in advance) can use all resources at night and during weekends.
- *Nodes reconfiguration*: the testbed provides a default (called *standard*) environment where users do not have root access, similarly to what is available on traditional HPC clusters. On top of that, bare-metal deployment using Kadeploy [6] provides users with the ability to run custom operating system images and get root access. The Grid'5000 team provides a number of images (called *reference environments*), and users can also create custom environments. Out-of-band consoles to nodes are also available to users.
- *Network reconfiguration*: the KaVLAN tool provides the ability to reconfigure the network. Each node's network interface (and nodes have up to four network interfaces) can be put in a different VLAN, which are reserved using OAR. This is typically used for cloud experiments [7] or networking experiments. Additionally, those VLANs can also be propagated inside the backbone links, providing isolation for multi-site experiments.
- *Network and power monitoring*: the Kwapi tool [8] provides an API, a live visualization interface and an archive of measurements of network traffic and power consumption of all nodes of the testbed, captured at high frequency.

While the testbed is distributed, the Grid'5000 engineers work as a single team that manages all sites in a single administrative domain. Each site is still assigned to a specific engineer (mainly in order to build knowledge about local specificities, and to perform maintenance in machine rooms), but the configuration of services is managed centrally, through the use of a configuration management system (Puppet).

III. Motivations

A. Very few bugs are reported

Reporting bugs or asking technical questions correctly is a difficult process [9], [10]. Typical users of testbeds (PhD students or post-doctoral students) rarely have that skill, or lack the confidence to report a bug about an infrastructure that they do not fully understand. In the context of Grid'5000, the fact that the team is geographically distributed also makes it hard to talk to a local contact that could confirm the problem informally. As a result, we receive very few bugs from users, despite, as we will show later, a large number of issues that could and should be reported.

Testbed operators would be in a good position to find and report such issues, as they clearly have the expertise about how the testbed should behave. But they often have limited experience of *using* the testbed, especially of the large variety of services that are offered to users, and as a result they are unlikely to face all the problems that users encounter.

B. But many bugs should be reported

Testbeds such as Grid'5000 can produce a large number of different and interesting issues.

A first obvious factor is its scale: 8 sites, 32 clusters and 894 nodes provide plenty of potential for subtle problems. Also, while methods to standardize software configuration at scale are reasonably well understood (e.g. configuration management), hardware is much more difficult to deal with: its configuration sometimes requires manual steps (inside BIOS for example), it tends to fail much more randomly than software (for example, due to aging), and exhibits silent and subtle failure patterns (as a real example: vibrations causing screws attaching hard disk drives to become loose, causing additional vibrations, that can have a performance impact [11]).

[1] https://www.grid5000.fr/w/Grid5000:UsagePolicy

Another, less obvious factor is the software stack and the ecosystem of services deployed on the testbed. Some core services are heavily used, but next to them, testbeds are always trying to design new services to address new experimentation needs. When they are first made available to the public, they rarely immediately pick up a strong user base, able to detect issues as soon as they occur. As a result, it is not unusual for a new service to be broken for a long period of time without testbed operators being aware, which is, of course, detrimental to attracting users.

C. And bugs can have dramatic consequences

In a testbed where most of the users are interested in measuring performance, subtle bugs can have a huge impact. For example, a misconfigured service or node could reduce performance by 5% or 10%, and thus lead experimenters to wrong conclusions about the solutions they are comparing, which could result in the need to retract a paper. Example cases where this could happen, all based on real facts, are:

- Different CPU settings, such as power management (C-states), hyperthreading, or turbo boost;
- Different disk firmware version;
- Different cache settings in a disk drive;
- Cabling issue that would cause wrong measurements by testbed-managed monitoring services (e.g. measuring the power consumption of another one).

In addition, there are also many problems that can be found at the software side, causing services to be unreliable and making it much harder to automate experiments.

IV. Design of our testbed testing framework

In order to design a testbed testing framework, we leveraged the Jenkins automation server to run testing scripts. But we had to work around several limitations of Jenkins through external developments, most notably for tests scheduling, and analyzing and summarizing results. The following paragraphs explore each of those aspects.

Overall, we tried to build on the widely accepted best practices in software engineering about test suites, continuous integration (CI) and continuous delivery (CD), but had to adjust because of specifics of the context, and of our goals.

A. Jenkins automation server

Jenkins [12] is the de facto standard for automating processes. In a nutshell, it can be seen as the *cron* Unix service on steroids. Using Jenkins, one can define *jobs* (tasks) that are executed in a specified environment, started by various means. The result and output of jobs are stored by Jenkins (as well as historical data).

Jenkins can be extended through plugins. A central plugin for our work was the Matrix Project Plugin, that adds support for defining jobs as matrices of several options. We used that for most tests in order to cover all possible configurations. For example, the `test_environments` job is in charge of testing the deployment of each of the 14 reference environments provided by the technical team, on each of the 32 clusters, resulting on 448 configurations for that test alone. Another related useful plugin was the Matrix Reloaded Plugin, that adds support for restarting a subset of configurations in a Matrix job.

Several other plugins also proved useful, such as the Build Timeout Plugin (to work around unexpected problems in some test scripts).

However, Jenkins alone proved insufficient for our needs, mainly for two aspects: fine-grained job scheduling, and analyzing and summarizing results. How we addressed those is detailed in the following sections.

B. Job scheduling

The scheduling of test jobs on the testbed was difficult to design (and a challenge that caused a previous iteration of work on this topic to fail). The requirements are fairly complex. First, different kinds of tests need to be addressed: some that focus on software, and only require one node per cluster; other that focus on hardware, and require all nodes from a cluster. Second, the scheduling must handle the fact that the resources might not be available (because of resources already reserved, or resources reservations made in advance for the future, that will prevent the allocation of resources for the required duration). Third, the scheduling must avoid disrupting other usages of the testbed.

Submitting test jobs as normal resources reservations, and waiting until resources are allocated, is not an option because: (1) Jenkins has a limited number of *workers* (job slots), and a pending reservation would use such for slot, possibly for days; (2) The test jobs would compete with user-made resources reservations, and possibly block resources when users would want to use them.

Also, submitting jobs at the same time every day or week is not an option either, because it would be unlikely that the resources would be available.

The solution that was implemented was a tool external to Jenkins that would, for each configuration of each test job, and on a regular basis (every 10 minutes): (1) query the status of the configuration, and make a decision about whether a new run should be attempted, based on the current state of the test (successful, failed) and on a per-job delay between attempts; (2) evaluate the status of the testbed to determine, using a basic analysis, if the job could start immediately; (3) start the job, and if it ends up not being scheduled by the testbed's resource manager after a few minutes (due to conditions that were not considered in step 2, cancel it (and mark it as *unstable* in Jenkins).

As a result, test jobs are only scheduled when resources are available, and as frequently as possible.

To further reduce the impact on users, two additional rules have been implemented. First, the scheduler avoids starting test jobs during *peak hours* (8 am-9:30 am, 12:30 pm-2:30

pm), that is, when most users are likely to start working on Grid'5000 and reserving resources. Second, the scheduler never starts two test jobs simultaneously on the same site.

On the busiest sites, this policy sometimes caused test jobs to be delayed for several days as resources were permanently reserved by users. For some jobs where partial results where particularly useful, specific test configurations for running the test on a single node or on all available nodes were added.

C. Analyzing and summarizing results

Another need that was not well served by Jenkins alone is the ability to provide a useful summary of the results. There are several different requirements: (1) per test status (for all sites or clusters); (2) per-site or per-cluster status (for all tests); (3) historical perspective.

A per test status is reasonably well provided by Jenkins, but is unfortunately made rather unusable due to the *unstable* status for jobs that were started but could not be scheduled. Jenkins does not provide a way to build a per-site status. Jenkins provides some historical perspective, using weather-like icons for each job, but it was insufficient for our needs.

To meet those needs, we designed an external status page by exporting data from Jenkins using its REST API[2]. That page (Figure 2) provides a table summarizing the status (current success percentage) for each site and test, and then a list of all failures that can be filtered using basic Javascript. Each failure can be annotated by engineers, for example to mention the corresponding bug.

Using the same method, we wrote a plugin for the Munin monitoring tool to keep historical data about each job and each site (Figure 3).

D. Why Jenkins, after all?

Due to the large number of Jenkins limitations that were worked-around, one could wonder if using Jenkins as a basis was really a good choice in the first place. We still believe it is, because (1) Jenkins provides a clean execution environment for scripts, with a queue to avoid overloading, the ability for users to trigger manual builds through a web interface; (2) Jenkins provides a storage system for test logs (and optionally, to store artifacts about test runs, such as raw data files, that could be useful in the future), the history of build results, and the ability to browse those test logs through a web interface.

Re-developing those features could have been an option, but would have resulted in significant development work. Additionally, Jenkins is also used for more traditional continuous integration tasks, and it makes sense to keep those CI tasks and testbed testing in the same tool in order to combine them. For example, there are CI tasks that build development versions of software, and then run testbed tests with those development versions installed, in order to evaluate whether those versions are suitable for release.

[2]Which, it is worth noting, is extremely well designed: it provides a way to get data about all jobs with a configurable level of detail in just one HTTP request.

E. Test scripts

The goal of test scripts should be to exhibit issues, but also to provide sufficient information to testbed operators to understand and fix the issue. One important limitation of bug reports submitted by users is that issues are described as the users see them, and usually not with all the information that testbed operators would like to have. This problem can be avoided using an automated testing framework only if the scripts are carefully designed in a way that provides that information.

For that reason, we wrote the test scripts using rather simple tools, and performing steps that would be close to what one would use when trying to reproduce a problem manually (which is considered a good practice for automation [13]).

As Grid'5000 is documented through a series of tutorials, an option that was considered was to use the content of those tutorials as the list of actions that would be automated and tested. If successful, it would have ensured that tutorials continue to work over time (which has been a problem in the past). However, this idea was rejected because (1) tutorials are designed with pedagogy in mind, not with test coverage; (2) most tutorials have several options (paths) at one point or another, making it difficult to automate. Therefore, for now we focused on simpler tests that cover the features of the testbed that are known to fail the most frequently, independently from how they are used in tutorials.

At the time of writing, the following tests have been developed and are in production:

- *refapi*: Check (1) the conformance of the description of each node in the Reference API compared to a schema; (2) That nodes of each cluster are homogeneous in terms of hardware configuration, according to their description in the Reference API.
- *oarproperties*: Check that the properties of nodes in the OAR resource manager match what would be generated based on the information in the Reference API.
- *oarstate*: Check the current state of nodes in the OAR resource manager (in particular, check that disabled nodes – due to hardware failure – are correctly documented).
- *cmdline*: Perform basic operations using command-line tools (reservation, deployment). All other tests rely on the Grid'5000 REST API.
- *sidapi*: Perform basic operations using the development branch of the Grid'5000 REST API. All other tests use the stable branch.
- *environments*: Check that each environment maintained by the Grid'5000 team can be provisioned on each cluster, and further check functionality of various features after provisioning (e.g., Internet access from the node).

Site	Average	cmdline	console	disk	environments	kavlan	mpigraph	multideploy	multireboot	oarproperties	oarstate	paralleldeploy	refapi	sidapi	stdenv
grenoble	77%	100%	33%	33%	87%	33%	100%	66%	100%	100%	100%	0%	0%	100%	66%
lille	85%	100%	60%	40%	98%	80%	100%	100%	80%	60%	0%	100%	0%	100%	80%
luxembourg	95%	100%	100%	100%	100%	50%	100%	100%	100%	100%	100%	0%	100%	100%	100%
lyon	78%	100%	25%	50%	100%	50%	75%	100%	25%	0%	0%	100%	0%	100%	100%
nancy	92%	100%	100%	55%	99%	66%	95%	88%	33%	100%	100%	100%	100%	100%	88%
nantes	88%	100%	100%	0%	100%	0%	100%	100%	0%	100%	100%	100%	100%	100%	100%
rennes	89%	100%	60%	40%	98%	60%	100%	100%	60%	100%	100%	100%	60%	100%	100%
sophia	80%	100%	75%	0%	100%	25%	100%	75%	50%	100%	0%	100%	0%	100%	25%
Average	86%	100%	69%	42%	98%	54%	96%	90%	54%	81%	62%	75%	45%	100%	81%

Showing 1 to 9 of 9 entries

hide bugs with comments | reset

Search:

Job	Configuration	Status	Last successful	Last failed	Streak	Last attempt	Next	Comment (from pad)
test_disk	site_cluster=nancy-graphene	Fail		2017-01-18 07:50:45	7	2017-01-24 15:18:39	2017-01-24 16:18:39	NORETRY graphene-[45,48] ont des disques différents
test_disk	site_cluster=grenoble-edel	Fail		2017-01-19 19:30:38	11	2017-01-24 15:18:39	2017-01-24 16:18:39	NORETRY Bug 7696 Les disques du cluster Edel ne sont pas homogènes
test_disk	site_cluster=nancy-griffon	Fail		2017-01-25 15:00:56	11	2017-01-25 15:00:56	No retry	NORETRY Bug 7675 griffon : disks are not homogeneous
test_console	site_cluster=rennes-paravance	Fail		2017-01-27 07:30:57	7	2017-01-27 07:30:57	2017-02-03 07:30:57	Bug 7770 - kaconsole failed on paravance-56
test_disk	site_cluster=rennes-paravance	Fail		2017-01-20 07:00:41	10	2017-01-27 07:31:10	2017-01-27 07:00:41	Bug 7737 test-disk on paravance
test_multireboot	environment=jessie-x64-min,site_cluster=sophia-uvb	Fail		2017-01-02 15:10:40	1	2017-01-02 15:10:40	2017-01-09 15:10:40	Bug 7686 - uvb - some nodes fail on reboot
test_kavlan	site_cluster=grenoble-genepi	Fail	2016-12-02 02:10:45	2017-01-24 20:30:57	12	2017-01-24 20:30:57	2017-01-31 20:30:57	Bug 7685 - kavlan fail to put node in VLAN 100: Configuration session timed out!
test_refapi	site_cluster=rennes-parapluie	Fail		2017-01-27 12:01:15	34	2017-01-27 12:01:15	2017-02-03 12:01:15	Bug 7585 Homogénéité des clusters de Rennes

Figure 2. Status page for all tests and sites

Figure 3. Historical status for each job, as provided by Munin

- *stdenv*: Similarly to *environments*, perform functional checks in the *standard* environment that is available on nodes without provisioning.
- *paralleldeploy*, *multireboot*, *multideploy*: Perform stress tests on the testbed, ensuring that it is possible to initiate several nodes provisioning operations concurrently, that nodes do not randomly fail to boot by rebooting them several times in a row, and that nodes do not randomly fail to deploy by deploying them several times in a row.
- *console*: Check that out-of-band consoles work on every node.
- *kavlan*: Check that network reconfiguration works on every node and network interface.
- *kwapi*: Check that power monitoring works on every node.
- *mpigraph*: Perform a MPI matrix bandwidth test to check connectivity and performance for all nodes of each cluster, and for each network technology (Ethernet, Infiniband, IPoIB).
- *disk*: Check homogeneity of disk configurations (read and write caches) and performance among nodes of the same cluster.
- *dellbios*: Check that BIOS parameters are homogeneous inside a cluster, and that some parameters follow testbed-wide rules (e.g. CPU configuration). Due to public procurement rules in France, most of the Grid'5000 clusters use Dell hardware, which justifies this vendor-specific test.

Overall, this currently results in a total of 751 test configurations. Other tests will be added in the future. The *kwapi* test should be extended to cover network traffic measurements. Other aspects of the testbed's network configuration could be tested, such as MTU settings or support for multicast. Also, at the software level, the tools for

automated deployment of OpenStack and Ceph should be tested on all testbeds.

V. Results and discussion

During the course of this work, 118 bugs were filed in the Grid'5000 bug tracking system, of which 84 have already been fixed. This includes issues that were directly found by the tests, but also problems that were discovered while writing tests, as writing scripts that should run on every cluster and site proved to be a good way to uncover various usability issues (e.g. differences between sites that are not described in the Reference API).

Here are a few examples of issues that were found in the process[3]:

- Several cases of heterogeneous configuration or documentation (different ways to document the same property on different sites) were uncovered by test *refapi* and some other tests (e.g. the *disk* and *dellbios* test): heterogeneous BIOS versions or configuration inside a cluster, disk drives configuration (read or write caching), CPU power configuration settings (C-states), etc. (bugs 7473, 7465, 7675, 7407, 7585, 7370, 7371, 7584, 7586)
- The *kavlan* test exhibited a number of cabling errors, where two nodes would be inverted in the KaVLAN configuration (bugs 7669, 7580, 7767, 7735, 7381, 7663, 7598, 7515, 7290), and cases where the driver for our network equipment would fail to properly handle error conditions (bug 7637, 7685);
- A number of other weak spots in our infrastructure (unable to handle load, or to work reliably) were uncovered by tests *paralleldeploy*, *multireboot*, *multideploy* (bugs 7482, 7403, 7415, 7686, 7502, 7503). Specifically, the out-of-band consoles service, which was thought to be reliable, was actually failing frequently (bugs 7770, 7362, 7570, 7325, 7466, 7574, 7575, 7411, 7576).
- Some configuration problems were detected in the images that we provide. The network configuration was invalid on one cluster for two images (bug 7342, 7302). The Infiniband stack was randomly failing to start on boot due to an interesting bug (Figure 4).
- A hardware issue causing random reboots on one of the older clusters (the Grenoble adonis cluster). As the warranty had expired, the cluster was shut down.

Many of the above issues are either issues that are difficult to detect as they do not cause nodes or services to stop functioning, but rather affect experiments in subtle ways; or issues that include an amount of randomness. Very few of those kinds of issues were reported by users.

Two issues in particular are worth explaining in more details.

[3]The corresponding bugs are available in the Grid'5000 bug tracker (https://intranet.grid5000.fr/bugzilla/). A Grid'5000 account is required, but could be obtained through the Open Access program.

Boot failures due to a race condition in the kernel (bug 7347): Our work also uncovered bugs that affected more critical pieces of software. While analyzing test failures, we noticed that nodes were taking much longer to boot in about 5% of cases. The problem was tracked down to LVM2 initialization waiting for `systemd-udev-settle` execution, which is a command in charge of waiting for all physical devices to be initialized that is included as a dependency for LVM2 in the Debian 8 boot sequence. However, due to a race condition in the kernel, a CPU core initialization was hanging, causing `systemd-udev-settle` to also hang until it reached a timeout. This has been reported to Debian[4] and will be fixed in a future update of the 3.16 branch of the kernel (it was already fixed in Linux 3.19, but Linux 3.16 is the version in Debian 8). This kind of issue shows that the devil is in the detail, and that no piece of software should be considered bug-free.

Heterogeneous disk performance on supposedly identical hardware (bug 7658): The Nantes Grid'5000 site hosts the *econome* cluster, composed of 22 Dell PowerEdge C6220 nodes (a single 2U chassis hosts four servers) with the exact same hardware configuration. On this cluster, the *disk* test showed lower performance on four nodes (from the same chassis): the test workload exhibited an average sequential read bandwidth of 79.3 MB/s on the slow nodes, vs 87.7 MB/s on the fast nodes (a 10% performance difference). Also, the SATA rate advertised by the disk was different (SATA 2.6 on the slow nodes, SATA 3.0 on the fast nodes). It turns out that the slow chassis was bought five months before the rest of the nodes (June 2012 vs November 2012), and, while it was indeed the exact same hardware configuration, it shipped with a different set of BIOS and firmware versions. Due to the lack of tests, this was not detected at the time of the cluster installation. Upgrading the BIOS version on the older nodes did not solve the problem, but upgrading the disk firmwares did.

Obviously, for all experiments performed on this cluster where storage performance had an impact, this puts into question the results that were obtained (as the heterogeneous performance might create results that depend on the placement of data on specific nodes), the repeatability of results (as different nodes from the same cluster would provide different results) and overall, the reproducibility of experiments.

VI. Related work

There has been very little work on testbed testing, verification and quality control.

In the context of the Fed4FIRE project, a monitoring system [14], [15] was designed and later extended [16], but it focuses on API-level compliance testing and authentication testing for testbeds members of the Fed4FIRE federation.

[4]https://bugs.debian.org/841171

```
local apps="opensm osmtest ibbs ibns"
for app in $apps
do
    if ( ps -ef | grep $app | grep -v grep > /dev/null 2>&1 ); then
        echo "Please stop $app and all applications running over InfiniBand"
        echo "Then run \"$0 $ACTION\""
        exit 1
    fi
done
```

Figure 4. Excerpt of the `openibd` script (part of the OFED Infiniband stack, and responsible for starting it during boot). the use of `grep $app` caused random failures to start, as any unrelated process with e.g. `libnss` in this command line would cause the test to succeed, and the service to abort start up. The image was updated to a newer version of the OFED script, which switched to using `pgrep` for more reliable process matching.

Similarly, GENI developed a test suite to verify AM API compliance [17].

On Grid'5000 itself, it was already mentioned in Section II that each time a node boots, a tool called *g5k-checks* downloads the node's description from the Grid'5000 Reference API, and then verifies using various inventory tools that the visible hardware configuration on the node matches the description from the Reference API [4]. This is complementary to the work described in this paper, as *g5k-checks* is not in a position to verify properties at the cluster, site or testbed level (e.g. that clusters are homogeneous). Also, the fact that it runs when the node boots make it time-critical, and thus it cannot run longer performance measurements or tests requiring the collaboration of several nodes. The Emulab-based testbeds (e.g. CloudLab) use a tool that is similar in scope to *g5k-checks* called CheckNode [18].

Also on Emulab, LinkTest [19] can validate the network characteristics of an Emulab experiment (connectivity, latency, bandwidth, link loss, routing).

More generally, this work builds on the products of good practices in software engineering, like software testing [3], continuous integration (CI), continuous delivery (CD) and automation in general. It is similar in some ways to performance regression testing, which is more and more advocated in various communities, e.g. in the Linux kernel development community [20].

VII. Conclusions and future work

In this paper, we presented the automated testing framework[5] that was designed in the context of the Grid'5000 testbed. This framework is in production and helped to identify 118 issues so far, with more to be found. It will serve as a basis for future work: the 15 tests that have been implemented could be extended to cover more cases; and additional tests will be added to cover additional services.

One limit of the current architecture is the scheduler. It currently schedules jobs on a per-cluster basis, and thus requires all nodes to be available. While the Grid'5000 Usage Policy makes this an acceptable compromise, it is still a problem with the busiest clusters. It would be better if the scheduling could be done on a per-node basis, with nodes being re-tested on a regular basis, when they are available. Unfortunately this would mean tracking the status of nodes outside of Jenkins, as Jenkins would not scale to a one-job-per-node model (which would also not make sense for the nodes provisioning, which is more efficient when done using groups of nodes).

The number of detected problems might seem important. It should be put in perspective with the fact that Grid'5000 sees a lot of users, and that those users are generally satisfied about the testbed. Many of the problems are corner cases that only affected some experiments, some nodes, or could stay unnoticed for a long time. Also, until one does not actually start looking for problems in a systematic way, it is difficult to estimate the number of problems that will be detected on a given testbed.

Generally, this work raises the question of the respective roles of testbed operators and experimenters. Shouldn't experimenters assume that the testbed needs to be verified or qualified before their experiments? Based on the number of uncovered issues, not many of the Grid'5000 users have been doing so. on the other hand, doing some amount of verification at the testbed level makes sense, of course, but it will never be exhaustive and cover all users' needs. Also, designing these kinds of solutions as part of a given infrastructure makes them testbed-specific, and makes them more difficult to port an experiment to another testbed, where guarantees will not be identical.

[5]Due to the large amount of hardcoded Grid'5000-specific things, the framework code itself is of little value for other testbeds, and thus we did not make the effort of making it publicly available. However, we are totally open to sharing it (or parts of it) if this is considered useful.

Acknowledgements

We would like to warmly thank the Grid'5000 technical team for fixing many of the issues that were discovered during the course of this work, thus providing an extremely valuable feedback loop.

The work presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER

and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

[1] R. D. Peng, "Reproducible research," Coursera lecture, Week 1, Part 2. [Online]. Available: https://www.coursera.org/learn/reproducible-research

[2] T. Buchert, C. Ruiz, L. Nussbaum, and O. Richard, "A survey of general-purpose experiment management tools for distributed systems," *Future Generation Computer Systems*, vol. 45, pp. 1 – 12, 2015. [Online]. Available: https://hal.inria.fr/hal-01087519

[3] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing*. John Wiley & Sons, 2011.

[4] D. Margery, E. Morel, L. Nussbaum, O. Richard, and C. Rohr, "Resources Description, Selection, Reservation and Verification on a Large-scale Testbed," in *TRIDENTCOM - 9th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, Guangzhou, China, May 2014. [Online]. Available: https://hal.inria.fr/hal-00965708

[5] N. Capit, G. Da Costa, Y. Georgiou, G. Huard, C. Martin, G. Mounié, P. Neyron, and O. Richard, "A batch scheduler with high level components," in *Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on*, vol. 2. IEEE, 2005, pp. 776–783.

[6] E. Jeanvoine, L. Sarzyniec, and L. Nussbaum, "Kadeploy3: Efficient and Scalable Operating System Provisioning for Clusters," *USENIX ;login:*, vol. 38, no. 1, pp. 38–44, Feb. 2013. [Online]. Available: https://hal.inria.fr/hal-00909111

[7] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding virtualization capabilities to the Grid'5000 testbed," in *Cloud Computing and Services Science*, ser. Communications in Computer and Information Science, I. Ivanov, M. Sinderen, F. Leymann, and T. Shan, Eds. Springer International Publishing, 2013, vol. 367, pp. 3–20.

[8] F. Clouet, S. Delamare, J.-P. Gelas, L. Lefèvre, L. Nussbaum, C. Parisot, L. Pouilloux, and F. Rossigneux, "A Unified Monitoring Framework for Energy Consumption and Network Traffic," in *TRIDENTCOM - International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, Vancouver, Canada, Jun. 2015, p. 10. [Online]. Available: https://hal.inria.fr/hal-01167915

[9] S. Tatham, "How to report bugs effectively," 1999. [Online]. Available: http://www.chiark.greenend.org.uk/~sgtatham/bugs.html

[10] E. S. Raymond and R. Moen, "How to ask questions the smart way." [Online]. Available: http://www.catb.org/esr/faqs/smart-questions.html

[11] B. Gregg, "Visualizing system latency," *Commun. ACM*, vol. 53, no. 7, pp. 48–54, Jul. 2010. [Online]. Available: http://doi.acm.org/10.1145/1785414.1785435

[12] "Jenkins – build great things at scale." [Online]. Available: https://jenkins.io/

[13] T. A. Limoncelli, "Automation should be like iron man, not ultron," *Queue*, vol. 13, no. 8, pp. 50:50–50:59, Sep. 2015. [Online]. Available: http://doi.acm.org/10.1145/2838344.2841313

[14] "Fed4fire – monitoring." [Online]. Available: https://flsmonitor.fed4fire.eu/

[15] B. Vermeulen, "Fed4fire d2.8 – third integration and testing roadmap," 2016. [Online]. Available: https://www.fed4fire.eu/wp-content/uploads/2016/10/d2-8-third-integration-and-testing-roadmap.pdf

[16] "Fed4fire federation monitor." [Online]. Available: https://fedmon.fed4fire.eu/

[17] "Geni am api acceptance tests." [Online]. Available: http://trac.gpolab.bbn.com/gcf/wiki/AmApiAcceptanceTests

[18] "Emulab – checknode." [Online]. Available: https://wiki.emulab.net/wiki/checknode

[19] "Emulab – linktest." [Online]. Available: https://wiki.emulab.net/wiki/linktest

[20] D. Bueso, "Performance monitoring in the linux kernel," in *Linux Plumbers Conference*, 2015. [Online]. Available: http://events.linuxfoundation.org/sites/events/files/slides/dbueso-lpc-2015-kperfmonitor.pdf