

A workflow-inspired, modular and robust approach to experiments in distributed systems

Tomasz Buchert, Lucas Nussbaum, Jens Gustedt

▶ **To cite this version:**

Tomasz Buchert, Lucas Nussbaum, Jens Gustedt. A workflow-inspired, modular and robust approach to experiments in distributed systems. [Research Report] RR-8404, 2013. hal-00909347v2

HAL Id: hal-00909347

https://inria.hal.science/hal-00909347v2

Submitted on 28 Nov 2013 (v2), last revised 23 Jun 2014 (v5)

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.

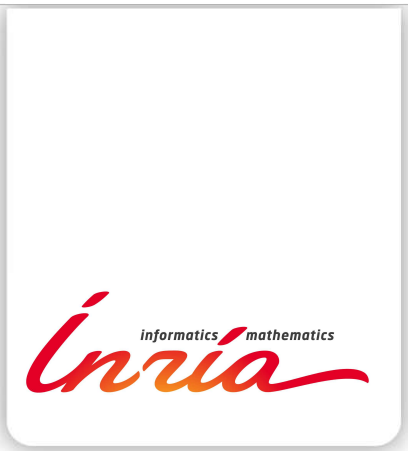


A workflow-inspired, modular and robust approach to experiments in distributed systems

Tomasz Buchert, Lucas Nussbaum, Jens Gustedt

RESEARCH REPORT
N° 8404
November 2013
Project-Team Algorille

ISSN 0249-6399 ISRN INRIA/RR--8404--FR+ENG



A workflow-inspired, modular and robust approach to experiments in distributed systems

Tomasz Buchert, Lucas Nussbaum, Jens Gustedt

Project-Team Algorille

Research Report n° 8404 — November 2013 — 18 pages

Abstract: Experimentation in large-scale distributed systems research is very challenging due to the size and complexity of modern systems and applications spanning domains of high performance computing, P2P networks, cloud computing, etc. Some obstacles that each researcher must face are: the difficulty of properly structuring experiments due to their complexity, the inflexibility of existing methodologies and tools, and the scalability problems resulting from the size of studied systems.

In this paper, we propose a novel method of representing and executing experiments that solves these problems. To this end, we present an interdisciplinary approach to the control of large-scale experiments in distributed systems research that draws its foundations from workflow management and scientific workflows. This workflow-inspired approach distinguishes itself by its representation of experiments, modular architecture and robust error handling. We show how the aforementioned problems are solved by our approach in an exemplary performance study of an HTTP server.

Key-words: experimentation; control of experiments; workflows; large-scale distributed systems; performance analysis

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

A workflow-inspired, modular and robust approach to experiments in distributed systems

Résumé : Expériences dans la domaine de systèmes distribués à grande échelle sont très difficile à conduire à cause de la complexité et de la taille des systèmes et des applications modernes (le calcul de grande performance, les réseaux P2P, le cloud computing, etc.). Entre les problèmes qui empêchent la recherche sont: la difficulté du design d'expériences, l'inflexibilité des méthodologies et des outils existants, et l'ensemble de problèmes liés au passage à grande échelle.

Dans cet article on propose la nouvelle méthode de structuration d'expériences qui résout ces problèmes. Notre approche interdisciplinaire de la gestion d'expériences dans la domaine de systèmes distribués à grande échelle utilise des idées et des méthodes efficaces de la gestion de workflows et des workflows scientifiques. La méthode se distingue par la représentation d'expériences en langage de workflows, l'architecture modulaire et la gestion d'erreurs robuste. On montre que les problèmes susmentionnés sont bien résolus grâce à notre approche en présentant l'étude de la performance d'un serveur HTTP.

Mots-clés : expérimentation; gestion d'expériences; workflows; systèmes distribués à grande échelle; analyse de la performance

1 Introduction

1.1 Context and motivation

Computer science started as a formal, theoretical science, but very rapidly became one of the most important of applied sciences. In parallel to the developments of computer science, computer systems were becoming more and more complex and traditional, formal analysis of them became unfeasible to the point that methods based on purely analytic, mathematical methods are almost non-existing (amounting to less than 4% [1]). There is, therefore, a trend of turning to scientific method and experimentation, both used routinely with much success in physical and life science research [2].

In the domain of distributed systems, the situation is arguably even worse. The objects of study, distributed systems built from loosely related components, with independent sense of passing time and connected using unreliable networks, are intrinsically non-deterministic. Moreover, the modern systems depend on countless software packages, each bringing its own influence on the overall behavior of the system. Large-scale distributed systems comprising thousands of independent machines are an extreme case where non-methodological and manual approach simply cannot work. Even a motivated scientist may be unable to produce high-quality results when confronted with such problems.

The difficulty is partially due to how the experiments are conducted. The complexity of existing distributed systems requires that the experiments are well-documented (as software should be), reproducible (so that other researchers can verify and use findings easily) and modular (to encourage reuse and standardize the domain). Moreover, they should be general, independent from any particular platform for example. Finally, they should be robust and address the faulty nature of distributed systems. Sadly, these issues are rarely addressed due to the difficulty and tediousness of such an undertaking.

The current methods and tools, even if they improve the situation considerably, are clearly not enough. We need dedicated methods to experimental research in large-scale distributed systems that address issues that the domain struggles with. The advances in the domain of *experiment control systems* would be welcomed and bring more credibility and quality into the research.

1.2 Contribution

In this paper, we describe a novel approach to control of experiments, which borrows ideas from experiment control systems, scientific workflows and workflow management systems. This workflow-inspired approach redefines the common activities of experimentation in one consistent, holistic methodology. We illustrate our findings with XPflow, a workflow engine tailored to the requirements of distributed systems research.

The rest of this article is structured as follows. First, in Section 2, we describe our approach and its main features:

1. a flexible workflow representation of experiments that promotes good practices, code reuse, improves understanding and paves a way to formal analysis,
2. a modular architecture that supports interoperability with low-level tools, instrumentation of experimental workflows and composability of experimental patterns,
3. robust error handling that includes special workflow patterns and checkpointing of workflow execution.

In Section 3, we evaluate our approach on a real-life experiment. Then, in Section 4, the existing solutions and methods for control of experiments in distributed systems research are presented and compared with our approach. Finally, Section 5 concludes the paper and outlines future work.

```
1  process :setup_experiment do |clients, server|
2      parallel do
3          forall clients do |node|
4              run :setup_client, node
5          end
6          sequence do
7              run :setup_server, server
8              run :start_server, server
9              log "Server started"
10         end
11     end
12 end
```

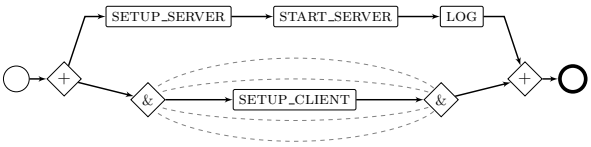


Figure 1: An example of a mapping between DSL and a workflow. Different language-level constructs (*parallel*, *forall*, *sequence*) result in respective workflow patterns (parallel execution of subworkflows, a parallel loop and sequential execution of activities). The workflow notation is based on Business Process Model and Notation (BPMN) with some extensions (the "ampersand notation" for a parallel loop, for example).

2 Workflow-Inspired Approach

Our approach is an interdisciplinary merger of three domains: experiment control systems, scientific workflows and workflow management. First, our main motivation is the control of experiments. Second, we share common concepts and goals with scientific workflows, the workflow representation of experiments and provenance in particular. Finally, we borrow the basic model of execution and workflow patterns from the domain of workflow management (a supporting science of Business Process Management).

The approach consists in using a domain-specific language to describe experiment workflows and execute them using a workflow engine tailored to control of experiments (called an *experiment control system* for short). The workflow engine launches, controls and monitors all aspects of experiment execution. A detailed log is kept which includes timing information, low-level and user-defined debugging information, etc. Additionally, it provides low-level features which cannot be implemented as workflows (e.g., checkpointing and error handling). Precise information about execution of the experiment can be exported, analyzed afterwards and stored as an experiment log.

Our implementation of that approach, XPflow, is written in Ruby which was chosen because of its popularity in configuration management tools (Puppet[1], Chef[2]) and its flexible syntax (for the purpose of implementing a high-level, domain-specific language). XPflow is a workflow engine that focuses on the control of experiments in distributed systems research. To achieve our goals a variety of different software and low-level tools are used: SSH (to control nodes), TakTuk [3] (for a scalable command execution), among others.

In the remaining part of this section, the three features of our approach are presented: the advantages of workflow representation, the modularity of the approach and the robust workflow execution. These features cover the main activities the experimenter has to cope with, that is: designing an experiment, implementing it and executing it, respectively. Each section starts with a problem that is addressed by that feature and then details of the solution follow.

2.1 Workflow-based description of experiments

One of main problems with published research is the lack of experimental verification of the presented ideas – less than 73% of papers that would need such a verification contained none

[1] `http://puppetlabs.com/`
[2] `http://www.opscode.com/chef/`

[1] (data from 2009). Even if experiments are presented in a publication, they are often poorly documented using informal language. The lack of a formal description of experiments prevents any formal analysis or verification of correctness. Moreover, the existing practices (use of low-level scripts and manual control of experiment execution) make it very difficult to monitor the progress or instrument the execution of experiments.

Our approach to this problem consists in representing experiments using workflows. To achieve that, we provide a domain-specific language to define workflows built from *activities*. Activities can be implemented imperatively (using a low-level programming language), or declaratively as *processes* that use various patterns to group activities and other processes into more complex workflows. In other words, the processes orchestrate the execution of activities, including other processes, whereas imperative activities implement final actions. The workflow engine parses the description and builds an internal graph representation of workflows which maintains one-to-one relation with their original description.

2.1.1 Domain-specific language representation

In our approach, workflows are described using a dedicated, domain-specific, declarative language. The building blocks are *activities* which can be grouped together into more complex workflows using special patterns for: sequential and parallel execution, looping over a set of variables, error handling, etc. (see Fig. 1 for an example). Almost every high-level feature of our approach is provided as a set of activities (including: acquiring resources for experiments, communication with nodes, analysis of data). For handling low-level tasks, the user is free to use traditional, imperative code.

The advantage of representing experiments in the textual form is that it does not diverge from existing practices which consist in describing experiments as a set of text files. Moreover, the human-readable text files can be easily copied, modified and published without any heavyweight or proprietary software.

2.1.2 Workflow representation

The experiment written in the domain-specific language is a workflow which can be represented as a directed graph with annotated nodes. The expressive power is close to that of workflow management systems as it uses standard patterns that were identified in the literature [4]. Workflows can be built from user-defined activities or activities from a core library that includes: node management, data collection and statistics, logging, etc.

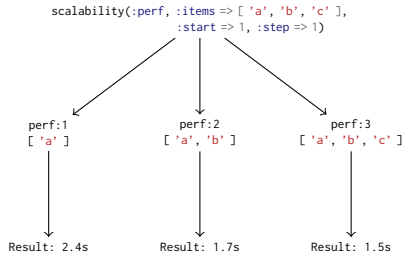
A workflow has a structure of a graph and is convenient to work with. Such a representation can be stored, modified and analyzed using well-known methods and algorithms. In particular, a static analysis of workflows can be done in order to identify their consistency and correctness (e.g., to ensure that all referenced activities exist). Moreover, workflows can be represented in graphical form which may be easier to understand than traditional ways to structure experiments.

The workflow representation enjoys a variety of monitoring possibilities. Information about execution times of activities is stored and can be analyzed to find bottlenecks, suspicious behavior or causal relations that affected the experiment. The monitoring data can be presented in a graphical form as well, for example as a Gantt chart.

2.2 Modular and extensible architecture

A common shortcoming of existing methods and tools is difficulty to extend their functionality or reuse them in a different context. Most of the tools support a single testbed and a require a dedicated software stack to function, for example. Similarly, many approaches suffer from limited

```
1   experiment :scalability do |exp, opts, *args|
2       items = items_of opts
3       n = length_of items
4       start = start_of opts
5       step = step_of opts
6       sizes = range(start, n, step)
7       foreach sizes do |size|
8           subset = item_range(items, size)
9           result = run(exp, subset, *args)
10          log "Result for #{size} is #{result}"
11          value([ size, result ])
12      end
13  end
```



Size	Performance
1	2.4
2	1.7
3	1.5

Figure 2: An example of a composable experiment that measures performance while running with more and more "items" (DSL representation on the left, an example how scalability experiment executes `:perf` experiment with a list of 3 items, and final results of the example on the right). A sub-experiment passed as an argument is executed with more and more *items* (e.g., clients). Additional options specify an initial size, a step size, etc. The final value is a mapping between number of items and results obtained with them.

possibilities to extend them, either by interoperating with external solutions or extending the capabilities of the framework itself. In both cases, the composability of them is important so that the modules can be used together in an arbitrary, flexible way.

In that regard, the workflow representation gives a powerful way to abstract any existing functionality, including one offered by external tools. The activities represent actions in a transparent way and can be grouped together to form more complicated workflows. Moreover, the workflows can be parameters for the execution of other workflows giving a powerful method of composing experiments from well-defined experimental patterns. Thanks to that architecture, most of the features of our approach are provided in a completely modular way.

2.2.1 Interoperability with testbeds and external tools

To illustrate the interoperability of our approach, we show how the control over nodes of the experiment is implemented. In particular we show its independence from an underlying testbed and a way to achieve a scalable method of command execution.

A primary abstraction is that of a node which on the workflow level is treated just as any other variable. To interact with the node, the user uses a set of activities that provide a portable implementation of common operations. These operations include: command execution (both simple and highly-scalable), copying files, retrieving files, etc. The method of accessing the nodes and software used to perform these operations is abstracted and transparent to the user.

The first benefit of this architecture is that the testbed used in the experiment can be easily changed by providing an alternative set of activities to perform all necessary actions on the new testbed. Another benefit is that it is equally easy to work with nodes at any level of the infrastructure. It does not matter if a node is accessible directly or via a gateway node of a testbed – from the workflow point of view both situations are exactly the same.

2.2.2 Composable experiments

The experiments are workflows which form a hierarchical structure. There is always a topmost, root experiment that encompasses the process of the experiment. During its execution sub-experiments can be started, which may contain their sub-experiments and so on. Each experiment contains its own log, set of files and properties and remains mostly independent from its parent experiment. An example of such an experiment is presented in Fig. 2.

An experiment can take another experiment as a parameter and execute it, possibly multiple times. For example, a study of scalability may consists in measurement of performance of an application as the number of clients varies. Instead of modeling this experiment as a monolithic loop that exercises more and more nodes, we can model it as two distinct problems: (1) implementing a process that runs a benchmark with a given set of nodes and (2) implementing an experiment that executes it with more and more nodes. The high-level experiments can be used in an arbitrary way depending on the needs of a user. This encourages reuse of code and good scientific practices.

2.3 Workflow execution and failure handling

The ultimate goal of experiment execution is to finish it with a success, even in the presence of failures. Unfortunately, the execution of experiments in large-scale experiments is difficult due to their complexity and size. Large-scale experiments involving hundreds or thousands of nodes reach scalability limits of a platform (e.g., overload network hardware) and fail not only for understandable reasons (e.g., timeouts in communication protocols), but also in completely unexpected ways. Failures cannot be left unnoticed, because they could bias measurements and conclusions. However some failures may be intermittent and therefore not fatal to the experiment execution and its correctness. These characteristics of the domain of large-scale experiments call for proper analysis of how the experiments can fail and what can be done about that.

Error handling in our approach consists of special workflow patterns to handle failures and checkpointing. However, before discussing them, we must start with a discussion about failures during experiment execution. There are two types of failures:

- platform failures – caused by external technical problems, software or hardware bugs, etc., in an idealized environment platform failures would not happen; an example of a platform failure is a dropped HTTP connection due to an overloaded server,
- workflow failures – a result of a wrong structure of the experiment or wrong assumptions which, contrary to the platform failures, happens even in an idealized environment; an example is an attempt to copy a file that does not exist.

The workflow execution can result in various outcomes: a success, a failure caused by the platform or a failure caused by the workflow. The platform failures can be further divided into intermittent and fatal ones. In practice, these two types of failures cannot be easily distinguished and standard approach consists in using timeouts and retries to decide if an intermittent failure is actually a fatal one.

The workflow representation offers many different patterns to group workflows, but the primary one is a sequence of workflows to execute in a fixed order. A sequence of workflows finishes with a success if all of them finish with a success. If any of them fails, the result of the sequence is a failure that caused its subworkflow to fail. Other workflow patterns have different semantics, for example a parallel loop fails with a *compound failure* if any of its parallel iterations fails. The compound failure groups all failures that caused the parallel iterations to fail (as there may be many of them).

Further on, we will say that two executions of workflows are *equivalent* if they lead to the same effect on the platform. It does not imply exactly the same state of the platform, but two states whose differences have no impact on the execution of the workflow and on the overall result of the experiment. The precise definition depends on the object of study and technical details. For instance, if hard disk performance is studied, any disk access may affect the state and hence experimental results. Therefore, executions that differ by the disk access patterns in general are not equivalent. However, if it is the network that is studied, the state of the hard disk can be arguably ignored.

```
process :prepare_experiment do |machines|
    forall machines do |machine|
        try :retry => 5
            run :install_software, machine
        end
    end
end
```

Figure 3: Error handling during software installation. If there is an intermittent failure during installation of software on each node, the process will be retried 4 times more until it will become an unrecoverable failure.

We will say that a workflow is *restartable* if it can be executed repeatedly (with a success or a failure) in sequence without causing a workflow failure. In other words the workflow execution cannot break its own assumptions as a result of the execution. The example of an action that is not repeatable is moving of a file – the file will not be present the second time the action is run, breaking its very own assumption necessary for proper execution.

A workflow is *idempotent* if a sequence of its multiple successful executions is equivalent to a single successful execution. Observe that making a copy of a file is idempotent, but appending to a file is not. A workflow composed of idempotent constituents does not have to be idempotent. For example, the sequence (copy a b; copy c a) is not idempotent if files b and c differ. Similarly, an idempotent workflow is not necessarily composed of idempotent actions – the sequence (copy a b; move b c) is idempotent as a whole even though moving a file is not.

If a workflow is restartable and in presence of intermittent failures will nevertheless eventually succeed in a state equivalent to a single successful execution, then we will say that it is *eventually successful*. If it is also idempotent, then will say it is *eventually idempotent*.

The aforementioned properties are crucial in a discussion that follows and describes failure handling patterns and checkpointing. We show that with an adequate way of handling failures, one can still properly execute an experiment despite presence of failures.

2.3.1 Failure handling patterns

A user can take a workflow and use it in a failure handling pattern that will retry its execution a predefined number of times. The inner workflow should be *eventually successful* so that it can be restarted multiple times until finally it succeeds. If the limit of retries is reached, the problem is treated as fatal and the workflow finishes with a failure.

The primary use for failure handling of workflows is to cope with intermittent failures which are a common problem in large-scale experiments. If, for example, many nodes use a shared resource (e.g., connection bandwidth, broadcast domain, etc.), some requests may fail due to timeouts, operating system limits, software and hardware bugs, etc. A common case is a highly parallel installation of software on many nodes – the bottleneck is the shared access to a single server with packages. Fortunately, the software installation process can be treated as restartable and hence our approach applies.

To sum up, if the experiment is built from eventually successful workflows and proper error handling is used, then the experiment can be eventually finished even in the presence of intermittent failures. Moreover, the results of the execution are equivalent to execution that experienced no failures at all. An example how the pattern is used in practice is shown in Fig. 3.

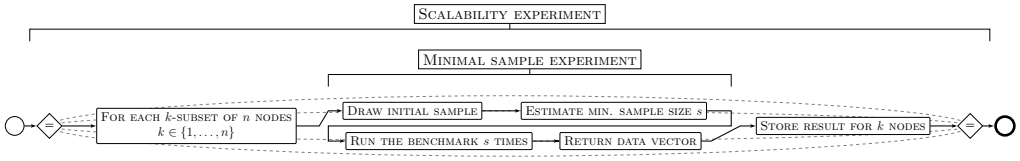


Figure 4: The principal workflow of the experiment consisting of 2 composable experiments and the main experiment (i.e., the HTTP server benchmark). The input of the experiment is a set of nodes provided by a testbed-specific workflow that can be replaced as we did in this paper. One node is designated as a server, the remaining act as clients. The *scalability experiment* takes a varying number of clients in a sequential loop (denoted with "=") and passes them to the *minimal sample experiment*. The *minimal sample experiment* executes the benchmark as many times as necessary to achieve a required statistical precision.

2.3.2 Checkpointing

A state of the experiment workflow can be saved in strategic, predefined places and restored later. A common use of that feature is to restart the execution from already reached checkpoints in a case of a failure, saving time and other resources. The experiments are often implemented iteratively, that is, by repeatedly designing and running the experiment until the researcher is satisfied with it. Thanks to checkpointing, the parts that are already finished do not have to be re-executed every time. Moreover, even if a complete experiment fails unexpectedly, the cause of the problem can be found, fixed and the experiment restarted.

Although checkpoints can be used anywhere in the main workflow, the properties of failures suggest that the placement of them is not arbitrary. More precisely, a use of checkpoint assumes that the remaining part of the workflow, executed after the checkpoint, is *eventually idempotent*.

Thanks to techniques like snapshotting of virtual machines or of the underlying file system, the checkpointing may include a state of the underlying physical platform where the experiment is executed. In that case, restarting a checkpoint consists in restoring the underlying platform before the state of the experiment workflow. The primary advantage is that the workflow following the checkpoint may assume a particular state of the platform and does not have to be even restartable. This functionality is not yet available in our implementation.

3 Evaluation

In this section, we will evaluate our approach with its implementation called XPflow. To this end, we designed an experiment that covers all relevant features and shows how they are used together. Our experiment is a basic performance analysis of a single server instance of nginx[3] HTTP server (version 1.2.1) with 4 worker threads, while serving a varying number of clients. As a benchmark, we use ApacheBench[4]. A single run of the experiment consists in measuring a throughput of a single HTTP server while serving many clients simultaneously for a timespan of 10 seconds. The result is measured in requests per second that the server was able to successfully deliver. The requested document is a short web page (that easily fits filesystem cache) and therefore we mostly measure the scalability and reliability of a request loop in the server. The operating system used to run all experiments is Debian 64-bit (wheezy) running a Linux kernel (version 3.2).

[3] `http://nginx.org/`

[4] `http://httpd.apache.org/docs/2.4/programs/ab.html`

3.1 Workflow representation

The workflow representation of the experiment exploits various patterns to build experiments. We used various sequential and parallel loops to iterate over sets of nodes, parallel execution patterns to perform actions in parallel, error handling patterns to harden the execution of the experiment, etc. The workflow representation is verified before execution to check if there are any problems that can be detected before runtime.

Moreover, the workflows can be visualized in a graphical form which is some cases provides an advantage over a plain, textual representation.

The overview of the experiment is presented as a workflow presented in Fig. 4. That part is independent from the underlying testbed and normally is preceded by testbed-specific part that reserves physical nodes and prepares a testbed.

3.2 Modularity and extensibility

Our experiment is built from reusable, modular components. The exemplary experiment is built from 3 sub-experiments with increasing depth of nesting:

- scalability experiment – runs a given experiment with an increasing number of objects (in our case these objects are clients),
- minimal-sample experiment – given an experiment adaptively tries to find a minimal number of samples necessary to reach a desired precision,
- the benchmark – the main experiment that benchmarks the performance of an HTTP server.

The nested structure of these components is clearly visible in Fig. 4.

Moreover, we have shown that thanks to our abstractions it is easy to change the underlying testbed. All variations of the experiment share most of the structure, only the part related to deployment and bootstrapping the experiment differs.

3.2.1 Lightweight containers-based platform

This platform was used to rapidly develop the experiment without necessity of using a real, shared testbed. The advantages are numerous: the ideas can be tested rapidly, the "testbed" resources are always available, the connection is much more reliable and fast. This feature can be described as a *virtual testbed* - a sandbox to test ideas, iterate rapidly and spot early problems.

On a technical level, this feature is implemented using Linux Containers[5]. All instances share a common base image using Copy-On-Write features of Btrfs filesystem, whereas their virtual network interfaces are bridged together and with the host network card. The process of launching containers is implemented as a separate XPflow workflow.

The machine used to host containers and develop the experiment was a typical desktop machine running Linux. The results for this "sandbox" testbed are not presented, since they are neither comparable with other testbeds nor can achieve a similar size of an experiment as them.

3.2.2 Grid'5000-based platform

This experiment is the most common case that we are interested in. The nodes used in the experiment come from two different clusters. One consists of nodes that are equipped with Intel Xeon X3440 processors, 16 GB RAM, magnetic hard disk and Gigabit Ethernet network cards. The second one consists of machines that only significantly differ in that they have Intel Xeon L5420 processors. We used a total number of 199 nodes during the experiment.

[5] `http://lxc.sourceforge.net/`

This experiment uses a small set of activities to interact with Grid'5000 testbed. The activities are used to submit user reservations, obtain information about associated nodes and install an operating system used during the main experiment. This reusable part amounts to few hundred lines of code, whereas the non-reusable boilerplate added to the original experiment is less than 30 lines.

3.2.3 Cloud-like platform

This variation of the experiment uses KVM-virtualized instances of Linux operating system. The physical nodes used to host the virtual machines are the same as those used in the previous experiment and are running Linux as well. The experiment is the most challenging one and the most time-consuming due to a fairly complicated installation process. A complete listing of workflows used to run the deployment is presented in Fig. 5.

The multi-level setup complicates communication with nodes at different levels of the installation, as they may not be reachable directly from the host running XPflow. In our case, we use one Grid'5000 node as a gateway to all virtual machines (it is also a gateway that acts as a router for virtual machines). Thanks to our abstractions the whole process is completely transparent (see Fig. 6).

The physical infrastructure that hosts the cloud-like installation is the same as in the Grid'5000-based experiment. Each physical node hosts two VMs per each core available on that node (that is, each core is shared by 2 instances). During the experiment we started 2034 virtual machines (and another VM dedicated as a web server).

Each virtual machine has a one virtual core and 1 GB of RAM available with the exception of the virtual machine that hosts the HTTP server – this node has 4 cores and 4 GB of RAM. Network interfaces of the virtual machines are bridged to the physical Ethernet network. This cloud-like infrastructure is in many respects similar to one of possible configurations offered by OpenStack[6] cloud software and arguably serves as a good model to study clouds with many deployed instances.

This variation of experimental scenario required to design and implement workflows to deploy the virtualized infrastructure. This (reusable) process amounts to 315 lines of code. Moreover, we had to prepare a KVM image used by the virtual machines. It required an installation of Debian Linux with necessary software. The image can be reused in different experimental scenario.

3.3 Failure handling

Our method takes a strict, almost paranoid approach to failure handling – by default every single problem will cause the experiment to fail. However, the user can manually define a retry policy for parts of the workflow forcing them to restart automatically if a failure happens during experiment execution, or by restarting the experiment from a checkpoint by manual intervention.

Our approach to error handling allowed to cope with some intermittent failures that happened during the experiments. In particular, the massively parallel installation of software on all nodes caused problems on some nodes. Thanks to failure handling these problems have been mitigated.

The experiment was structured according to the workflow properties described before. For example, the checkpoints were used in proper places to save time and ease the process of debugging. During development of the experiment, they were occasionally used to fix a bug in an experiment that caused it to fail and then restore a state from a previously attained state.

[6] `http://www.openstack.org/`

```
use :g5k
import :http, "../http/common.rb" # import the main exp.

IMAGE = "/home/tbuchert/public/wheezy.qcow2"

# shadow package installation
process :"http.deploy" do |master, slaves|
    execute master, "service nginx start"
end

process :install_master do |master|
    distribute master,
        "files/master.sh", "/tmp/"  #! text="Copy files"
    execute_many master,
        "bash /tmp/master.sh"       #! text="Configure the node"
end

process :install_slaves do |slaves|
    distribute slaves, "files/*.sh", "/tmp/"   #! text="Copy files"
    distribute slaves, $ssh_key, "/tmp/"       #! text="Copy SSH key"
    chain_copy slaves, IMAGE, "/tmp/"          #! text="Copy KVM image"
    execute_many slaves, "bash /tmp/client.sh" #! text="Conf. nodes"
end

activity :get_instances do |master, slaves|
    results = run :execute_many_here, slaves, "cat /tmp/__instances__"
    mega_result = results.to_list.map(&:stdout).join
    nodes = mega_result.strip.lines.map do |line|
        name, ip, mac, group = line.split
        run :proxy_node, master, "root", ip
    end
    nodes
end

process :run_nginx do |slaves, master|
    try :retry => 3 do
        run :"http.conf_performance", master, slaves
    end
end

process :perform_experiment do |master, slaves|
    run :"http.deploy", master, slaves
    step = var(:step, :int)
    results = run :"http.scalability", :"/run_nginx",
        { :items => slaves, :start => step, :step => step }, master
    save_yaml "./results.yaml", results
end

process :deploy_grid5000_nodes do
    job = g5k_auto_raw :site => var(:site)
    vlan = g5k_kavlan_id(job)
    vnodes = g5k_kavlan_nodes(job)
    vnodes = code(vnodes) { |xs| xs.map(&:host) }
    nodes = g5k_kadeploy(job, "virsh-image",
        "--vlan #{vlan}", :real_nodes => vnodes)
    bootstrap_taktuk(nodes)
    value(nodes)
end

process :start_nginx_server do |server|
    execute server, "bash /tmp/flush.sh"  #! text="Stop all VMs"
    execute server, "bash /tmp/server.sh" #! text="Start server VM"
end

process :launch_instances do |nodes|
    execute_many nodes, "bash /tmp/flush.sh"  #! text="Stop all VMs"
    execute_many nodes,                       #! text="Start client VMs"
        "bash /tmp/instances.sh $TAKTUK_RANK"
    sleep 60
end

process :retrieve_instances do |master, server, hosts|
    nginx = get_instances(master, server)
    clients = get_instances(master, hosts)
    bootstrap_taktuk(nginx)
    bootstrap_taktuk(clients)
    value([ first_of(nginx), clients ])
end


process :deploy_cloud do
    nodes = deploy_grid5000_nodes()  #! text="Deploy Grid'5000 nodes"
    log "Using #{length_of nodes} G5K nodes."
    checkpoint :nodes_kadeployed
    master, slaves = shift nodes
    log "The master is #{master}, slaves are #{slaves}"
    parallel do
        install_master(master)
        install_slaves(slaves)
    end
    checkpoint :configured
    server, hosts = shift slaves
    parallel do
        start_nginx_server(server)
        launch_instances(hosts)
    end
    nginx, clients =
        retrieve_instances(master, server, hosts) #! text="Configure VMs"
    value([ nginx, clients ])
end

process :cloud_experiment do
    nginx, clients = deploy_cloud :deploy_cloud
    checkpoint :testbed_prepared
    log "The nginx server is #{nginx}."
    log "There are #{length_of clients} clients."
    test_connectivity(nginx)
    test_connectivity(clients)
    checkpoint :before_main_experiment
    perform_experiment(nginx, clients)
end

main :cloud_experiment
```

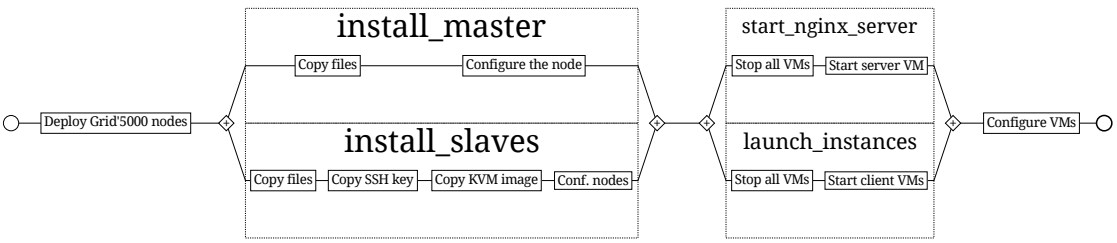


Figure 5: A full listing of workflows used to deploy a cloud-like testbed and an autogenerated graphical representation of the `deploy_cloud` workflow (thanks to the annotations inside comments). The blocks defined with `process` keyword describe workflows, whereas a block defined with `activity` contains a low-level Ruby code. Various patterns discussed in the paper are presented: failure handling, checkpoints, parallel execution and composable experiments.

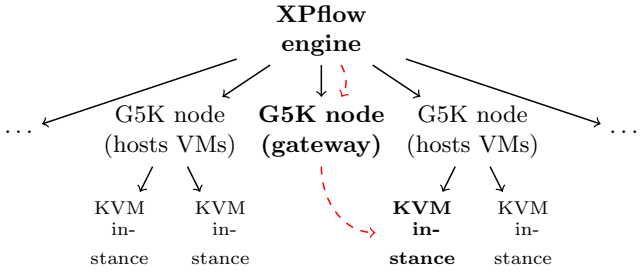


Figure 6: A hierarchical structure of nodes used in the cloud-like experiment. Three nested levels can be distinguished (XPflow engine, Grid'5000 nodes, KVM instances). The red, dashed path describes a connection used to access and control virtual machines.

3.4 Results

The obtained results are presented in Fig. 7. For each data point, the experiment was repeated at most 10 times to obtain a sample mean value that is at most 3% from the real mean value with 95% confidence. The data points that failed to converge are presented with intervals of the same confidence, but of a larger size.

Unsurprisingly, the results obtained on a physical testbed are better than on a virtualized environment. The peak performance reached more than 30000 requests per second for 180 clients and to our knowledge would only improve if more nodes were used. We were however limited by the size of physical clusters available to us.

The results obtained in a virtualized environment are not as good. The performance reached around 10000 requests per second in the range that we experienced with. The lower performance is due to overhead of CPU and network interface virtualization. Moreover, one has to remember that we measure the performance from the perspective of clients – their performance of submitting requests has an effect on the performance as well. We noticed as well that above a number of 900 simultaneous clients some requests were not served successfully, but the overall performance remained the same.

All in all, the nginx HTTP server offers very good performance and scalability while serving a large number of clients. Thanks to a cloud-like infrastructure we were not limited by the size of physical infrastructure and were able to run our experiment with more than 2000 virtualized nodes. The raw results, XPflow software and the experimental workflows are available at `http://www.loria.fr/~buchert/xpflow2013.tgz`.

4 Related Work

XPflow is a result of an interdisciplinary merger of three domains: it is an *experiment control system*, it shares common ideas with *scientific workflows* and borrows from *workflow management*.

4.1 Experiment control systems

Experiment control systems have an objective of managing the execution of experiments so that the process is easier and more reliable. Among many ways to differentiate between them, one can look in particular at: the way they treat the problem of describing and structuring experiments,

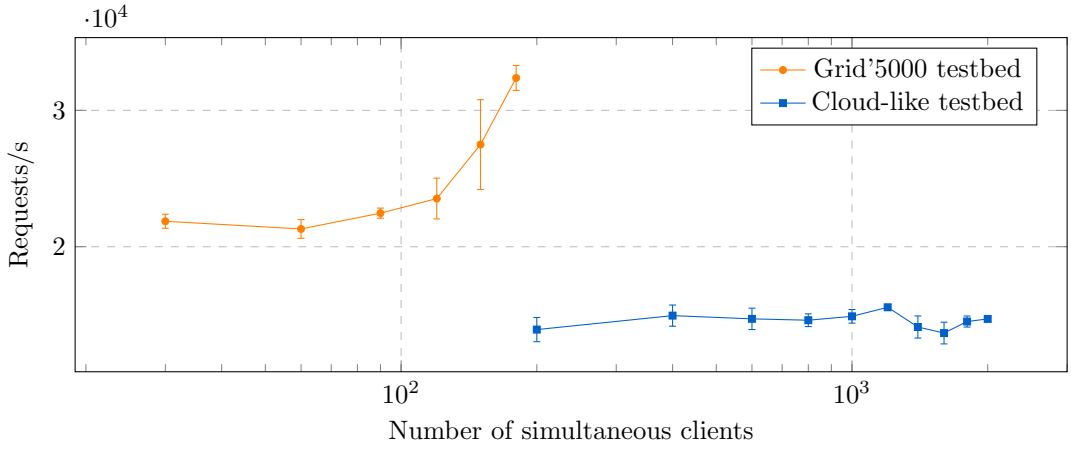


Figure 7: Performance (measured in requests per second) of the nginx HTTP server while serving a varying number of clients (X axis has a logarithmic scale). Results for two different testbeds are shown: one built using two physical clusters and one cloud-like with virtualized nodes. Each data point is presented with its 95% confidence interval.

how modular and extensible their architecture is and how the reliability and correctness of experiments are achieved.

4.1.1 Experiment description

The methods of representing experiments range from purely imperative representations to declarative ones based on high-level models and languages. The former approach usually extends an existing programming language with a set of routines to control experiments, effectively turning the experiment description into a program. This representation, although theoretically equivalent to any possible computation, suffers from few problems: it is low-level, cannot be thoroughly analyzed and is unsuitable for expressing complex patterns. Therefore most of the tools use a dedicated high-level representation to describe experiments. Although it solves some problems of low-level representation, it does not come without some of its own: one has to learn to use it (at the expense of time), the language may be limited in its expressiveness or be difficult to extend it with existing software or tools.

The former group is represented by Expo [5], a tool that extends an imperative programming language (Ruby) with a set of abstractions to control nodes. The experiment is more readable, but has no high-level structure that can be analyzed. Moreover, failures are managed on a level of a programming language, which is difficult and may be inconvenient. Many tools use a low-level programming language to describe high-level constructs, for example Emulab Workbench [6] uses it to describe an experimental platform, and OMF [7] to provide an event-based description. Another approach is featured by Splay [8] which offers a description language which is intentionally similar to pseudo-code used commonly in scientific papers in distributed systems research that hides common difficulties of experimentation. Such a description allows to rapidly prototype an experiment and run it on a model of a platform, but is mostly limited to one-time studies. Finally, solutions like Weevil [9] or Plush [10] turn to high-level, declarative descriptions of experiments.

Our approach uses a special language to build workflows, but is conceptually independent from any programming language. The high-level workflow representation is easy to understand,

expressive, easy to analyze and to work with.

4.1.2 Modular architecture

The means of extending a given approach vary greatly and are related to the way the experiments are represented. Approaches based on high-level representations may be constrained by this very fact, whereas low-level tools may offer no convenient way to extend them.

Approaches concerned mostly with one-time studies (like Splay [8]), or tools dedicated to a particular platform (Emulab Workbench [6] or Plush [10]) do not tackle this problem directly. Solutions similar to Expo [5] offer theoretically unlimited capabilities, but are not modular by design and extending them is difficult. High-level solutions tend to be self-contained and difficult to extend as well, but also offer much more extensive built-in functionality (e.g., Weevil [9] offers a powerful workload generation functionality). Finally, an example of OMF [7] shows that even a platform-dedicated and relatively high-level approach may be extensible, as is shown by its instrumentation, logging and interoperability capabilities.

Our approach proved to be very flexible as it can be extended in various ways. Not only the functionality can be extended with external software in a modular way, but even the testbed can be replaced if necessary. Moreover, common patterns can be reused as has been shown with composable experiments.

4.1.3 Robustness

Due to non-deterministic behavior of distributed systems the failures must be taken into consideration by every approach. The failure handling can be automatic or manual, the former typical to methods that are dedicated to a particular platform and latter to methods constrained by their general applicability. Nevertheless, every solution has some means to cope with problems with scalability and intermittent failures.

Among other features, Plush [10] can ensure proper connectivity between nodes, Emulab Workbench [6] and OMF [7] can verify that the platform complies to a specification – a set of features that would be otherwise difficult to provide if not for the tight integration with the platform. For similar reasons, Splay [8] can hide file management and related problems, whereas general solutions like Expo [5] turn to mostly manual error handling.

To our knowledge no approach has defined precisely the way the failures are handled and coped with. Although some solutions claim to handle failures and scalability problems, neither the ways to achieve this are explicitly stated nor the effects on results of the experiment discussed. Our workflow-inspired approach provides a well-defined framework built around the idea of correct execution and gives a set of properties and methods necessary to achieve this objective.

4.2 Scientific workflows

Scientific workflows are a broad term encompassing tools aiming at automation of scientific processes. They are mainly used to run complex computations on preexisting data using grids, and are not used to manage experiment execution, especially experiments in distributed systems research. Normally, those systems are provided with GUIs that enable non-expert users to easily construct their workflows as a visual graph. Such graph describes the complete flow of the computation and the dependency between tasks and data generated (i.e., provenance). A prominent feature of scientific workflows is their independence from the computing infrastructure. Thanks to that the failures of the platform are handled in an automatic and invisible way (other types of failures may not be, however). Some examples of scientific workflows are: Kepler [11], Taverna [12] and Vistrails [13].

The scientific workflows are data-oriented and a distributed system underneath (usually a computational grid) is merely a tool to efficiently process data, not an object of a study. Although our approach represents the experiments as workflows as well, there are incompatible differences. First, scientific workflows are data-oriented whereas in our approach they represent actions performed on the platform. Second, the platform is not hidden from the experimenter, but is a central and explicit element of the workflow execution. Last, the workflows in our approach are based on a different formalism.

4.3 Workflow management

Traditionally related to workflows as well, is the domain of Business Process Management and Modeling (BPM) [14]. Business processes are found in business organizations and the goal of BPM is to model, understand and improve them using appropriate techniques and tools, most of them computer-aided nowadays. A viewpoint that we adhere to is to think about BPM as a *management discipline* (so it's neither a technology nor has anything to do with computer science) and *workflow management* as a computer science discipline supporting it [15]. In that sense, we are more interested in the latter, but parallels between experimentation and BPM has been observed as well [16]. From the discipline of workflow management we primarily reuse the common workflow patterns and the execution model.

5 Conclusion and Future Work

In this paper we have shown how research in distributed systems research can profit from the workflow representation of experiments. Our interdisciplinary approach to control of experiments is based on workflow management and scientific workflows. Among its advantages is the flexible and modular representation that enables building experiments from reusable components and transparent interoperation with testbeds or external software without much additional work. Our approach has a well defined model of execution based on BPM workflow patterns and robust failure handling. To verify claims about our approach, we designed and conducted an exemplary, large-scale experiment.

In future work, there are a few aspects we would like to concentrate on. First, we would like to push the scalability of our experiments further (say, some tens of thousands of nodes). Second, various ways to extend our approach should be explored: recording provenance of data in experiments, adaptive methods for execution of experiments, advanced checkpointing, for example. Finally, we have to release XPflow so that scientists could profit from our approach and give us a valuable feedback.

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`, [17]).

References

[1] J. Wainer, C. G. Novoa Barsottini, D. Lacerda, and L. R. Magalhães de Marco, "Empirical evaluation in Computer Science research published by ACM,"

Inf. Softw. Technol., vol. 51, no. 6, pp. 1081–1085, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2009.01.002

[2] W. F. Tichy, "Should computer scientists experiment more?" *Computer*, vol. 31, no. 5, pp. 32–40, May 1998. [Online]. Available: http://dx.doi.org/10.1109/2.675631

[3] B. Claudel, G. Huard, and O. Richard, "TakTuk, adaptive deployment of remote executions," in *Proceedings of the 18th ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, ser. HPDC '09. New York, NY, USA: ACM, 2009, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1551609.1551629

[4] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow Patterns," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5–51, Jul. 2003. [Online]. Available: http://dx.doi.org/10.1023/A:1022883727209

[5] B. Videau, C. Touati, and O. Richard, "Toward an experiment engine for lightweight grids," in *Proceedings of the First International Conference on Networks for Grid Applications (GridNets 2007)*, ser. GridNets '07. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, pp. 22:1–22:8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1386610.1386640

[6] E. Eide, L. Stoller, T. Stack, J. Freire, and J. Lepreau, "Integrated scientific workflow management for the Emulab network testbed," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 33–33. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267359.1267392

[7] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "OMF: a control and management framework for networking testbeds," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 54–59, Jan 2010. [Online]. Available: http://doi.acm.org/10.1145/1713254.1713267

[8] L. Leonini, E. Rivière, and P. Felber, "SPLAY: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze)," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, ser. NSDI'09. Berkeley, CA, USA: USENIX Association, 2009, pp. 185–198. [Online]. Available: http://dl.acm.org/citation.cfm?id=1558977.1558990

[9] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf, "Automating Experimentation on Distributed Testbeds," in *Proceedings of the 20th IEEE/ACM International Conference On Automated Software Engineering (ASE)*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 164–173. [Online]. Available: http://doi.acm.org/10.1145/1101908.1101934

[10] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat, "Loose synchronization for large-scale networked systems," in *Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 28–28. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267359.1267387

[11] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao, "Scientific workflow management and the Kepler system," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006. [Online]. Available: http://dx.doi.org/10.1002/cpe.994

[12] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services." *Nucleic Acids Research*, vol. 34, no. Web Server issue, pp. 729–732, July 2006. [Online]. Available: http://view.ncbi.nlm.nih.gov/pubmed/16845108

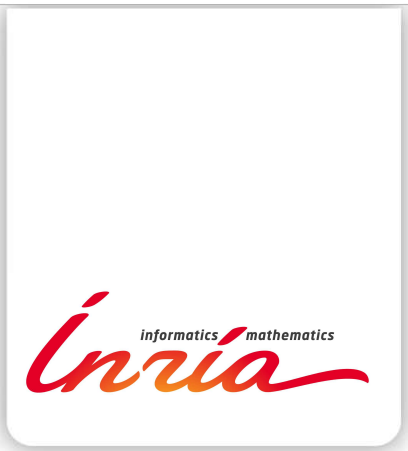
[13] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo, "VisTrails: visualization meets data management," in *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 745–747. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142574

[14] M. Weske, *Business Process Management: Concepts, Languages, Architectures*, ser. 978-3-540-73521-2. Springer, 2007. [Online]. Available: http://www.springer.com/computer/information+systems+and+applications/book/978-3-642-28615-5

[15] R. K. L. Ko, "A computer scientist's introductory guide to business process management (BPM)," *Crossroads*, vol. 15, no. 4, pp. 4:11–4:18, Jun. 2009. [Online]. Available: http://doi.acm.org/10.1145/1558897.1558901

[16] T. Buchert and L. Nussbaum, "Leveraging business workflows in distributed systems research for the orchestration of reproducible and scalable experiments," in *9ème édition de la conférence Manifestation des Jeunes Chercheurs en Sciences et Technologies de l'Information et de la Communication - MajecSTIC 2012 (2012)*, A. Etien, Ed. Lille, France: Nicolas Gouvy, Aug. 2012. [Online]. Available: http://hal.inria.fr/hal-00724313

[17] F. Cappello, F. Desprez, M. Dayde, E. Jeannot, Y. Jégou, S. Lanteri, N. Melab, R. Namyst, P. Primet, O. Richard, E. Caron, J. Leduc, and G. Mornet, "Grid'5000: a large scale, reconfigurable, controlable and monitorable Grid platform," in *6th IEEE/ACM International Workshop on Grid Computing (Grid)*, Nov. 2005. [Online]. Available: http://hal.inria.fr/inria-00000284/en/



RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399