Managing large-scale, distributed systems research experiments with control-flows

Tomasz Buchert

UNIVERSITÉ DE LORRAINE

École doctorale IAEM Lorraine

MANAGING LARGE-SCALE, DISTRIBUTED SYSTEMS RESEARCH EXPERIMENTS WITH CONTROL-FLOWS

THÈSE

présentée et soutenue publiquement le 6 janvier 2016

pour l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE LORRAINE
(MENTION INFORMATIQUE)

par

TOMASZ BUCHERT

Composition du jury :

Rapporteurs

CHRISTIAN PÉREZ, Directeur de recherche Inria, LIP, Lyon

ERIC EIDE, *Research assistant professor*, Univ. of Utah

Examinateurs

FRANÇOIS CHAROY, Professeur, Université de Lorraine

OLIVIER RICHARD, Maître de conférences, Univ. Joseph Fourier, Grenoble

Directeurs de thèse

LUCAS NUSSBAUM, Maître de conférences, Université de Lorraine

JENS GUSTEDT, Directeur de recherche Inria, ICUBE, Strasbourg

Laboratoire Lorrain de Recherche en Informatique et ses Applications – UMR 7503

ABSTRACT (ENGLISH)

Keywords: *distributed systems, large-scale experiments, experiment control, business processes, experiment provenance*

Running experiments on modern systems such as supercomputers, cloud infrastructures or P2P networks became very complex, both technically and methodologically. It proved difficult to run experiments correctly and understand obtained results, even with the background on the employed technology and methods. Moreover, large-scale experiments suffer from erroneous and the unpredictable behavior of underlying software and hardware, undermining the scientific principles of experimental computer science. This worrisome state of research on large-scale distributed systems calls for new approaches to design, run and interpret experiments.

This work explores the use of control-flows (business processes) as a model for representing the large-scale experiments in research on distributed systems. We set out to find advantages, disadvantages and limitations of this approach, and practical considerations for future implementers.

We make 3 main contributions. First, we analyze the current state of experiment management tools, their limits and features to better understand difficulties that lay ahead. We construct a general framework to evaluate tools of this type. Second, we design and implement an experiment management tool which is based on the model of control-flows. We show that this methodology can be implemented and used in practice to run challenging and large-scale experiments while offering a wide set of features, some of them missing in the previous approaches. Finally, we analyze the use of provenance in computer science, and in particular in experimental research on distributed systems, and propose a provenance collection system that emerges from the control-flow model used as the representation of experiments. The design is implemented and shown to collect provenance in efficient and automatic way.

Our results show that workflows are a viable model for the design and execution of experiments in distributed systems research. With these positive conclusions in mind, we also sketch future research directions for improving our work.

ABSTRACT (FRENCH)

Titre : *Gestion d'expériences à grande échelle dans la recherche sur les systèmes distribués à l'aide de control-flows*
Mot-clés : *systèmes distribués, grande échelle, control-flows, démarche scientifique, processus métier, workflows scientifiques*

L'expérimentation sur les systèmes modernes comme les superordinateurs, les infrastructures cloud ou les réseaux P2P, est devenue complexe à cause des difficultés techniques et méthodologiques. La réalisation correcte d'expériences et l'analyse des résultats obtenus est difficile, même en possédant toute l'expertise nécessaire sur le domaine d'étude et la technologie utilisée. De plus, les expériences à grande échelle échouent souvent en raison du comportements aléatoires du matériel et du logiciel, menaçant les principes de la recherche expérimentale comme la fiabilité et la reproductibilité des résultats. Cette situation inquiétante de la recherche sur les systèmes distribués à grande échelle nécessite la découverte de nouvelles approches pour la structuration, le contrôle et l'interprétation d'expériences.

Ce travail explore l'utilisation de control-flows (processus métier) comme un modèle pour la représentation d'expériences à grande échelle dans le domaine des systèmes distribués. Il analyse les avantages, inconvénients et limitations de cette approche, ainsi que des considérations pratiques pour leur implantation future.

Trois contributions principales peuvent être distinguées. D'abord, nous analysons l'état actuel des outils pour le contrôle d'expériences. Nous montrons les fonctionnalités manquantes et permettons de comprendre les difficultés partagées par toutes les approches. Cette analyse se termine avec la construction d'une hiérarchie des propriétés qui peut être utilisée pour l'évaluation des outils qui contrôlent les expériences. La deuxième contribution consiste en un design et une implantation d'un système de contrôle d'expériences qui se base sur le modèle de control-flows. Nous montrons que cette méthodologie est capable du contrôle efficace et robuste des expériences à grande échelle et offre des fonctionnalités nécessaires, dont certains ne sont pas présentes dans les approches existantes. La dernière contribution porte sur la conception et l'implantation d'un système pour la collection de provenance pendant l'exécution d'expériences sur les systèmes distribués. Elle utilise intensément le modèle de control-flows et améliore l'approche présentée précédemment. Le prototype de ce système est capable d'une collection de provenance de manière efficace et automatique.

Les résultats obtenus montrent que le modèle proposé est une approche viable du contrôle d'expériences dans les systèmes distribués. De plus, les améliorations possibles sont mentionnées à la fin du document.

PUBLICATIONS

Some ideas, figures and results have appeared previously in various scientific publications:

- The parts of Chapter 2 dedicated to the state of the art of Business Process Management (Section 2.5) and the early results on using control-flows in management of experiments (Chapter 4) have been published as two conference articles [25, 27].
- Most of Chapter 3 has been published as the main subject of a journal article [32].
- The large part of Chapter 4, including the elements of evaluation, has been presented as a conference article [30].
- Some experimental results presented as the part of evaluation in Chapter 4 have been presented in a conference paper [26].
- Similarly, some results presented in Chapter 4 have been previously published during a workshop [123].
- The topics discussed in Chapter 5 have been published as a workshop paper [31].
- The early version of the presented experiment control engine was used by Attila Döme Lehóczky in his MSc thesis entitled *A testing framework for validation and improvement of the SMPI simulation framework for MPI applications*[1].

Hence the contributions of this thesis are represented by 7 publications of the author. Four of them [25, 27, 30, 31] discuss the subject directly, whereas the remaining ones [26, 32, 123] use the methods presented here as research methodology. The archive containing all assets accompanying the thesis can be downloaded[2].

Finally, although not a scientific publication, another contribution is represented by software that was written during the work on this thesis. It is freely available for anyone interested[3] (contact the author if it is not the case).

1 https://github.com/lehoo/thesis
2 http://xpflow.gforge.inria.fr/thesis/all.tar.xz
3 http://xpflow.gforge.inria.fr

Overcome by example, grumbling to himself, nevertheless,
Porthos stretched out his hand, and the four friends repeated
with one voice the formula dictated by d'Artagnan:

"All for one, one for all."

— Alexandre Dumas, *The Three Musketeers*

ACKNOWLEDGMENTS

This work has a single author, but it would be foolish to think that it could have been finished by the author alone. So many people are involved directly or indirectly that it is almost impossible to credit all of them, but one can try nevertheless.

I would like to thank my family for the support they gave me. They filled me with hope, energy and perseverance to pursuit this goal. I love them all.

I thank my advisors for the time they invested and the undeserved patience on their part. This work would be much worse without them.

I offer gratitude to my coauthors, coworkers and friends. They gave me a great environment to work and live and taught me more things that anybody could ask for. This is the best gift that one can get.

Special thanks go to the Grid'5000 staff without whom I would not be able to validate my ideas.

Last but not least, all best to thousands of people who develop and care for Free Software. Not only is it crucial for the reproducibility of research, but also for the society as a whole.

Thank you all!

CONTENTS

LIST OF FIGURES

LIST OF TABLES

The beginning is the most important part of the work.

— Plato, *The Republic*

1 INTRODUCTION

1.1 CONTEXT AND MOTIVATION

Computer science is a relatively young domain of science which has seen a rapid and unabated evolution since its infancy which can be situated somewhere in the middle of twentieth century. The objects of its study, computers as well as other information-processing components, have also seen unstoppable development thanks to research and technological advances. The computing systems became much faster, but at the cost of increasing complexity: indeed, the performance of the most powerful computing systems seems to double every year, as well as their size [185] – *faster* than what the Moore's Law would predict.

As a result, the current computer systems are of enormous complexity and are nearly ungraspable to human beings in their entirety. Such systems are examples of *distributed systems*, since they can no longer be regarded as independent elements, but must be seen as one, complex system. To put it simply, a distributed system is a computer system "in which the failure of a computer you didn't even know existed can render your own computer unusable", an informal definition given by Leslie Lamport, one of the pioneers of distributed systems research.

The role of research in distributed systems is hence to make these systems more robust and allow them to function despite various problems. The difficulty is due to the fact that the constituents of distributed systems are fallible, either because of imperfect production process or because of software that is unprepared for exceptional situations or simply incorrect. In fact, hardware and software errors are by far the most common source of problems in large computer systems [93].

Since powerful distributed systems are a necessary enabler for other sciences, this quest is of crucial importance. Biology, chemistry, astrophysics, medicine, meteorology, physics (the analysis of tremendous amount of data from the Large Hadron Collider led to the discovery of Higgs Boson in 2012), to name just a few, are life sciences that routinely profit from the computing power of modern distributed systems. But they are not the only ones - large IT companies like Google, Microsoft, Yahoo, Facebook or Amazon use distributed computing to serve millions of users every day.

Naturally, these complex systems require complex evaluation. When formal verification is possible, there is often a great discrepancy between what theory promises and what the real world requires [46]. Therefore, the gen-

eral trend is to turn to *experimentation* as the mean of validation and evaluation [184].

There are many methodologies, methods and tools that help with experimental evaluation of distributed systems. First, there is *simulation*, where the model of a system is evaluated instead of a real system. Although *simulation* is based on simplified assumptions about the reality, it can lead to useful conclusions, yet the discrepancy between theory and reality still applies. Then, there are *in-situ* experiments which evaluate the system on a real platform. This approach, although virtually unrestricted, brings multiple difficulties.

The inherent complexity of the real-life system must be tamed by the experimenter, including numerous factors that can influence the behavior of the system. Unfortunately, there is no way so far to find what can influence the results and even quite innocuous factors can play a weighty role [133]. Again, there are many approaches proposed to control this aspect: emulation, virtualization and cloud computing for example [74], which are a middle ground between simulation and in-situ experiments. What is more, experiments with distributed systems must explicitly take into account their faulty nature. If not, failures may go unnoticed and obtained results will be wrong.

All these difficulties threaten the *reproducibility* and as we know from the famous philosopher of science, Karl Popper, *non-reproducible single occurrences are of no significance to science*. Contrary to the previous problems, however, the lack of reproducibility is partially a fault on our own. Indeed, the experiments themselves are conceptually complicated and there is no efficient or formal way to disseminate them. More insights into the process of experimentation and the ways of understanding it are necessary.

Scientific workflows are an approach in computational sciences (e. g., biology, medicine, genetics or astrophysics) that successfully addresses this difficulty. Experiments represented as workflows have a formal structure that can be analyzed, shared and understood much easier than the unstructured ones [208]. Unfortunately, such methods do not apply, at least directly, in distributed systems research due to their data-centric model.

Not-that-distant cousins of scientific workflows, *business processes modeled with workflows*, have been applied with a similar success in more "real-life" situations like production and process management, or orchestration of web services [116]. They focus much more on the *control* of the process, than on the efficient data shuffling and processing as scientific workflows do. Moreover, as they are used to model fallible, real-life systems, they are explicitly designed to cope with exceptional situations.

It seems that the complex, unreliable, real-life systems are in fact quite similar to the complex systems that need the sophisticated methods of control to work properly, efficiently and intelligibly. David Wheeler, a famous computer scientist and the first one to complete a PhD in computer science is quoted saying that *all problems in computer science can be solved by another level of indirection*. Is Business Process Management this indirection that we look for? This is a question that we will turn to in the following pages.

1.2 PROBLEM STATEMENT

The problem that is addressed in this thesis is the worrisome state and the difficulty of experimenting with large-scale distributed systems. The most important intermediate problems can be observed:

- How can the experiments be reliably controlled and finished despite their large size and delicate nature?
- How should the experiments be represented so that scientists can understand and improve them?
- How should the experiment data and provenance be gathered, stored, structured and queried so that useful observations can be made swiftly and reproducibly?

Our approach is based on the domain of Business Process Management and hence further questions must be distinguished:

- Are workflow patterns found in business processes a viable model for experiments in distributed systems?
- What advantages and disadvantages, if any, do they have?
- How does this method differ from existing approaches?
- How can a system based on this idea be implemented? Can it be efficient?

1.3 CONTRIBUTIONS

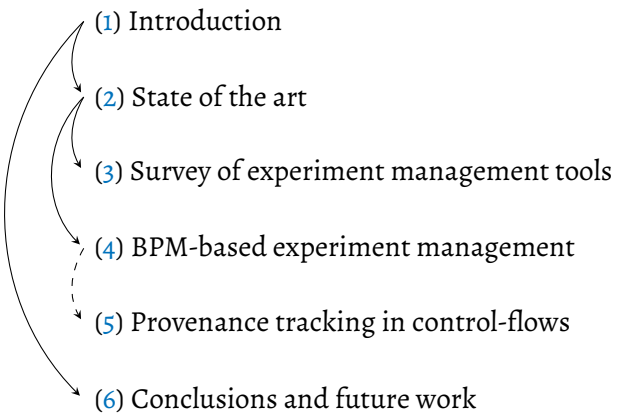
The thesis makes 3 principal contributions to the state of the art:

1. It analyzes the current state of experiment management tools for distributed systems research to understand their limits and features. This analysis leads to deeper understanding of difficulties laying ahead, as well as a general framework to evaluate such tools. This work is the main subject of Chapter 3.
2. It studies how experimental research in distributed systems can profit from applying control-flows and Business Process Management, and how a system based on them can be designed. We show how to use this methodology in practice by presenting three case studies based on it. More importantly, we observe that it can lead to better understanding, robustness and scalability of experiments by verifying these claims experimentally in Chapter 4.
3. It proposes a solution to the difficult problem of collecting provenance in experiments expressed as control-flows. Different types of provenance are distinguished and the design and implementation of a system that collects it efficiently is presented. This is the main subject of Chapter 5.

1.4 STRUCTURE OF THE DOCUMENT

The document consists of 6 chapters. Apart from the current introductory part and the last one dedicated to conclusions, the remaining chapters can be divided into two non-disjoint parts. The first part discusses the prior art and the context of the work and consists of Chapter 2 and Chapter 3. The second part consists of Chapters 3, 4 and 5 and is dedicated to the contributions made in this thesis. As can be seen, the role of Chapter 3 is twofold: it analyzes the prior work, but is a contribution on its own.

The chapters can be to some extent read separately and each chapter starts with an introductory text that should help the reader. That said, assuming that the introduction has been finished, the prerequisites for each chapter are as follows. Chapter 2 generally requires no prior knowledge and is also strongly recommended for Chapter 3 and Chapter 4, unless the user has experience with experimental research in the domain of distributed systems. Chapter 5 is more general than others and most of it can be reasonably approached on its own (although the knowledge of workflow models is strongly recommended). For deeper understanding, Chapter 4 is indispensable as it gives the technical context. This weak requirement is shown with a dashed arrow in the following summary:



The overview of each chapter is presented below.

CHAPTER 1 – INTRODUCTION

In this chapter we introduce the subject and the motivation of the thesis in general terms. The problem considered is precisely stated, as well as the contributions made in this thesis. Finally the structure of the document is presented, so that the reader can easily navigate. This is the chapter that you read right now.

CHAPTER 2 – STATE OF THE ART

In this chapter we analyze the current state of the art in the domain of research in distributed systems research. This includes domains such as the methodology of experimental research, the scientific workflows and the busi-

ness process management. Note that discussion of methods for provenance collections is postponed to Chapter 5.

CHAPTER 3 – SURVEY OF EXPERIMENT MANAGEMENT TOOLS

This chapter is the analysis of the experiment management tools for research in distributed systems. First, the features of such tools are identified, defined and explained. Then all relevant experiment management tools are evaluated according to it, showing their strong and weak points. Moreover, important conclusions are made about the general state of such tools. It is the first major contribution of the thesis.

CHAPTER 4 – BPM-BASED EXPERIMENT MANAGEMENT

In this chapter, we introduce XPFLOW, an experiment management tool for distributed systems research, which, distinguishably, is based on Business Process Management. We motivate our approach and show how it helps with the problems encountered in distributed systems research. Then we describe the design of our tool and evaluate it with various case studies to validate our claims. It is the second major contribution of the thesis.

CHAPTER 5 – PROVENANCE TRACKING IN CONTROL-FLOWS

This chapter is devoted to the tracking of provenance in control-flows. First, we analyze the existing work in the domain of provenance collection. Then, we discuss the importance of provenance in research and identify the distinct types of it. Later, we design a provenance system that can track and collect each of them in an efficient way. This system is also evaluated to show its usefulness regarding provenance collection. It is the third major contribution of the thesis.

CHAPTER 6 – CONCLUSIONS AND FUTURE WORK

In the last chapter, we sum up the contents, the contributions and the results of the thesis. The limitations of our methods and methodologies are discussed as well, followed by the overview of the future work that lays ahead in this domain.

My Mama always said you've got to put
the past behind you before you can move on.

— Forrest Gump, *Forrest Gump*

2

STATE OF THE ART

This chapter is dedicated to the analysis of prior art in domains supporting the theses of this work. Such an analysis is necessary to understand the difficulties of performing research on distributed systems, as well as the current position of researchers tackling these problems.

The chapter consists of a few sections each devoted to a different and independent subject and therefore can be read separately. First, in Section 2.1 we will discuss the state of research in distributed systems, its challenges, methodologies and supporting tools. Then, in Section 2.2, tools helping all researchers in computer science will be presented. This will be followed by Section 2.3 where we will narrow our focus to tools supporting research in distributed systems. Finally, we will discuss the domain of scientific workflows in Section 2.4 and Business Process Management in Section 2.5. Provenance collection and its state of the art is postponed to Chapter 5.

2.1 EXPERIMENTAL DISTRIBUTED SYSTEMS RESEARCH

Each domain of research comes with its own challenges and peculiarities and this is obviously true for experimental distributed systems research. We analyze methodologies, trends and tools used in the domain, hoping to better understand its difficulties and ways by which it can be improved. In Section 2.1.1, we will introduce the domain and motivations for studying distributed systems. Then in Section 2.1.2, the status of the domain as a science will be discussed, followed by the presentation of standard research methodologies in Section 2.1.3. Then the reproducibility of experiments, especially those in distributed systems research, will be discussed extensively in Section 2.1.4. Finally, some high-level methodological difficulties of experiment design will be discussed in Section 2.1.5.

2.1.1 *Motivation*

Distributed systems became indispensable in the modern, interconnected world. Their study becomes increasingly important as they form the skeleton of modern data processing, communication and countless services that people depend and rely on.

Distributed systems are used on a daily basis, sometimes even without their users noticing it. Domain Name Service [130], for example, is a globally

distributed name service used on the Internet. Together with electronic mail, they are arguably the two oldest distributed systems in existence. More recently peer-to-peer file sharing (P2P) became another prominent distributed system which is responsible for up to 70 % of Internet traffic in some regions of the world [167]. Some of these systems are vital to correct and the continuous operation of the Internet. Their proper design and evaluation is of utmost importance, as consequently is the research in distributed systems.

Distributed systems became also a necessity for highly available, globally distributed services for coordination, data storage, transaction processing, communication, indexing and monitoring. In particular, large IT companies implemented many complex systems, for example: Spanner [50] (a globally distributed SQL database), Percolator [144] (an indexing service behind the Google's search engine), ZooKeeper [104] (a coordination service) and MapReduce [59] (a parallel programming model), among countless others.

What permeates the research of these well-funded bodies is the unanimous use of the experimental verification of their systems. Formal methods are only used to verify some parts of systems if at all. Moreover, some researchers honestly admit that it is very difficult to apply theoretical results in practice and there is no practical methodology to develop distributed systems [46]. For that reason researchers look not only for new approaches, but also for simpler solutions to problems that has been, at least in theory, already solved [141].

2.1.2 *Scientific method in computer science*

The beginnings of computer science were purely theoretical and the reason for that is quite simple: there were no computers at that time. When Alan Turing published his famous paper *On Computable Numbers, with an Application to the Entscheidungsproblem* in 1936, paving the road for future computer revolution, there where no practical realizations of his *Turing machines*. There were machines that were able to *compute* (such as the Enigma machine), but the first programmable machine capable of the *universal computation* was constructed only a few years later, in 1941, by Konrad Zuse [160].

In this light, it may be considered incorrect to call computer science a *science* since *science* is *the intellectual and practical activity encompassing the systematic study of the structure and behavior of the physical and natural world through observation and experiment* (Oxford Dictionary). This image has been reinforced by some computer scientists (cf. attributed to E. Dijkstra: *computer science is no more about computers than astronomy is about telescopes*), but also contested by others [184]. Actually, computer science is probably all of that: mathematics, science, engineering, even art, and all combinations of them [64].

It seems, however, that computer science turns to non-formal methods and a purely analytical, mathematical approach is almost non-existing (i. e., amounts to less than 4 % of publications [193]). Even if we grant the status of *science* to computer science, a question remains whether computer scientists actually follow the scientific method and its principles. It has been reported

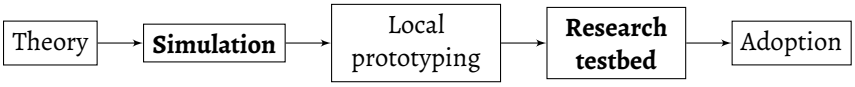


Figure 1: Model describing progression from research to adoption [4] (with some minor adjustments). As can be seen, it involves both simulation and *in-situ* experimentation on research testbeds.

that approximately 50 % of computer science papers proposing models or hypotheses do not test them [184], whereas in other fields of science this number is as low as 10 % [64]. Clearly the scientific method has not become yet the mainstream paradigm in computer science.

The situation is worsened by the fact that the term *experiment* is often ambiguous. At least five uses of *experiment* exist: *feasibility experiment*, *trial experiment*, *field experiment*, *comparison experiment* and *controlled experiment* [182]. Each of them represents a different activity with slightly different standards and rules for running it. Although *controlled experiments* are the gold standard of scientific research in many fields of science, it is the other meanings that prevail in computer science.

2.1.3 *Research methodologies*

We can distinguish two entities involved in experimental research in distributed systems: an *application* and an *environment*. An application is a system under the test itself, whereas an environment is everything that makes the context of the experiment, but is not the application itself. It may be tempting to associate the application with *software* and the environment with *hardware*, however this is not the case. Software may well be a part of the environment (e. g., libraries installed in a system) and hardware may be a part of the application (e. g., sensors in a wireless mesh network).

Both the application and the environment can be *real* (i. e., similar to production systems) or a *model* (i. e., a simplified view of a real system). By intersecting these two orthogonal aspects of the application and the environment, the four following research methodologies in distributed systems can be defined: the *in-situ* experiments, benchmarking, emulation and simulation [101] (see Table 1). Some existing solutions do not fit in this classification: Splay [119], for example, can be considered both a simulator and an emulator.

It is worth noting that the members of National Science Foundation argue in their report [4] that there is a natural progression from a research idea to the commercialization which involves the aforementioned methodologies. This is presented in Figure 1.

In the following sections we will outline each methodology in detail.

2.1.3.1 *In-situ experiments*

A methodology of experiment consisting in running a real application in a real environment, *in-situ* experimentation, is the main subject of this thesis.

Table 1: Research methodologies in distributed systems research.

		Application	
		Real	Model
Environment	Real	In-situ	Benchmarking
	Model	Emulation	Simulation

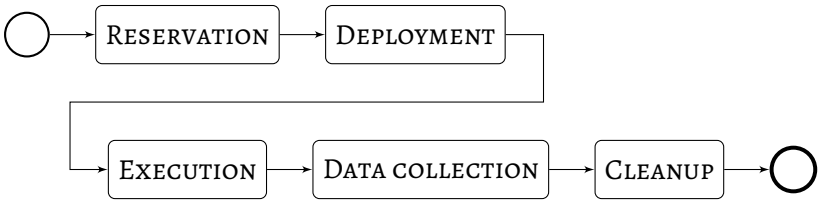


Figure 2: High-level model of a single instance of an *in-situ* experiment. It consists of 5 steps: Reservation, Deployment, Execution, Data collection and Cleanup. Note that these steps make up the step 3 (Perform the experiment) in Figure 3.

As the most general methodology it is also the least standardized and countless approaches and tools exist to support it. In fact, as standard environments and well-known software are used, such an approach can be and is used by people with common computer science knowledge.

The *in-situ* methodology gives the encouraging promise of the most relevant results. Indeed, the way the experiments are conducted does not differ significantly from a deployment in a production system. The same problems must be solved and the same issues arise in both cases. Nevertheless, the approach can be criticized for not being general nor reproducible, since its experiments depend on numerous factors that are difficult extract, control or may have erratic and hence unreproducible behavior.

As *in-situ* experiments are the main subject of this work, it will be useful to understand how they are structured. By observing habits of scientists and existing publications, the simplified, high-level model of steps involved in a single instance of an *in-situ* experiment can be obtained. It consists of 5 steps:

1. Reservation of resources (Reservation).
2. Deployment of necessary software and configuration (Deployment).
3. The experiment proper (Execution).
4. Collection of results (Data collection).
5. Cleaning up (Cleanup).

There are, of course, experiments that do not follow these steps to the letter. Moreover, the model presumes that each step is sequential, but it is possible, for example, to collect results as the experiment proper executes. We will refer to this model, presented graphically in Figure 2, throughout the thesis.

Tools supporting this methodology will be presented in more detail in Section 2.3. Existing tools to manage and control *in-situ* experiments will be analyzed extensively in Chapter 3.

2.1.3.2 *Benchmarking*

Benchmarking is a common way to evaluate a distributed system. It is performed by running a synthetic application on an existing, real system and evaluate its performance. The metric of performance used depends on the benchmark, but the most common examples are: time, the number of requests per second, energy consumption, etc.

Benchmarking is usually standardized within each domain. Indeed, there are benchmarks to measure the performance of high-performance computing (HPC) [147], memory bandwidth [125] or CPU performance [175]. The results from benchmarks are relatively easy to obtain and easy to compare between different experiments. For example, the TOP500 list [185] of the most powerful computer systems in the world is compiled according to the results of a single benchmark: the number of floating-point operations per second (i. e., FLOPS) that a given system can achieve.

It is of course not necessarily true that the system will achieve the same performance in a real-life situation. It is the main drawback of the approach, as the system is being represented by a single, possibly idealized and unrealistic metric. This bears similarity with representing a whole probabilistic distribution by its moments such as the expected value or its median.

2.1.3.3 *Simulation*

In simulation, both the application and the environment are modeled, which has a few advantages. First, it is in theory fully reproducible and separated from the effects of unreliable hardware. Second, models of the application and of the platform enable easier and faster development, as well as simplify the interpretation of results. Finally, some simulators let the researchers run their experiments at the scale that is unattainable in the reality [155].

On the other hand, simulators target a specific domain and may not meet the needs of research. Moreover, they represent a simplified behavior of a real system that may not reflect reality. Moreover, the simulated application is effectively a prototype that cannot be directly used in production systems. There is research, however, on transforming software models to real systems while keeping their essential properties. Finally, there is a surprising lack of standardization in simulation. Currently, nearly 75 % of P2P papers using simulation employ custom simulators and this fraction increases [17, 134].

Simulation is a popular approach in domains such as general distributed systems [44], P2P networks [17] or networking.

2.1.3.4 *Emulation*

Emulation consists in running a real, unmodified application in a model of platform. The main idea is to tame the inherent variability of *in-situ* experi-

ments by using a model of the environment, while still maintaining useful properties, such as the use of standard tools.

Most emulation techniques focus on the emulation of network properties such as latency and bandwidth (Dummynet [41, 42, 159], P2PLab [136], Study of Network Emulators [137]). Other techniques integrate emulation of CPU performance (Distem [28, 29, 165]), IO speed limitation (Wrekavoc [37, 38, 73]) and emulation of multiple nodes via containerization (PlanetLab [40], Distem [165], LiteLab [194], Mininet [201]). Network emulation is sometimes a core feature implemented in research testbeds (e. g., Emulab [165], PlanetLab [40]).

The next group of emulation techniques is based on *time dilation* which consists in changing the perception of time experienced by an application (DieCast [100], Time Jails [97], SliceTime [199] and others [209]). This powerful technique enables emulation of nonexistent hardware, such as high-performance CPUs, network devices and storage devices.

Another approach, lying on a crossing between emulation, virtualization and containerization, relies on emulation of a whole operating system [66, 67]. User-Mode Linux (UML) is a specially crafted Linux kernel that can run as a standard, unprivileged program that interacts using userspace libraries and interfaces. A recent work in this area led to a userspace library that fully emulates the Linux networking stack and can be used with ns-3 network emulator [181].

Emulation has been used to run large-scale network experiments in the environment resembling Internet [188]. This approach is another successful application of the Dummynet emulator.

Disadvantages of using emulation in research are inherited from both *in-situ* experiments and simulation. Depending on the particular case, a given emulator may suffer from low reproducibility, the difficulty of controlling the experiment (*in-situ* emulators), or from too simplistic assumptions, unrealistic emulation and restricted environment (emulators leaning towards simulation).

2.1.4 *Reproducibility*

Science is a process which accumulates knowledge by observing phenomena, hypothesizing about their causes and testing these hypotheses experimentally. If the results of an experiment turn out to be in favor of the hypothesis, confidence in it is strengthen, but a failure to support it is a reason to reject it altogether (assuming the experiment was correctly conducted). The more evidence, especially from independent sources, in favor of the hypothesis, the stronger is the reason to believe it is true.

Reproducibility is an ability to show the veracity of the given scientific hypothesis by a different group and preferably in different settings, for example with different measurement techniques. Reproducibility consists therefore of making sure that the phenomenon is general, not a mere artifact of an experiment, coincidence, luck or a result of a scientific fraud.

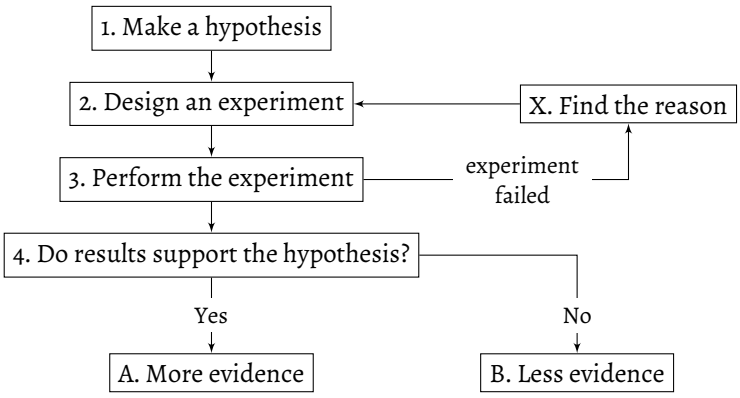


Figure 3: Experimental cycle. The steps 1, 2, 3, 4 and A make an ideal research process one can dream of. However, it may happen that the experiment failed for some reasons (X) or did not support the working hypothesis (B). Note that the step 3 is essentially represented by the model in Figure 2.

Reproducibility is considered a hallmark of science, since phenomena that cannot be reproduced cannot be considered general laws. It should not be confounded however with *replicability* that focuses on being able to obtain *the same conclusion* multiple times and *in a short period of time* (e. g., one day). *Repeatability*, on the other hand, is a more robust form of replicability and extends this period of time (e. g., more than a day). The proposed periods of time are just an example, the importance lies in the following facts:

- Replicability concerns itself with the experiments conducted in an immediate succession, with the same conditions and the same mindset.
- Repeatability assumes that some time has passed, the accumulated domain knowledge is probably lost, conditions or software may have changed or protocols have been revised. The design of the experiment remains nevertheless the same in principle as well as the group of researchers that experiments.

This can be summarized in the following way by referring to the idealized experimental cycle in Figure 3:

- *Reproducibility* consists in being able to successfully redo steps 2, 3, 4 and A. It means that the same scientific hypothesis is supported by a substantially different piece of evidence obtained by a different group, tools and methods.
- *Replicability* is about being able to successfully redo steps 3, 4 and A. In this case, the experiment design remains essentially the same. It is therefore rather *automation* of the experiment than its full reproduction. In the context of distributed systems research it leaves the following elements remain unchanged: software, external factors (e. g., workload and injected faults), configuration, etc.

- *Repeatability* is just as replicability above, but in time periods that are long enough for the domain knowledge about the experiment to become lost.

Replicability (and hence repeatability) still remains an ambiguous term and the different types of it have been distinguished in science [96]. Moreover, the usefulness of replicability in science has been questioned [71]. It may seem, that the lack of reproducibility in science may undermine its public perception and trust, but not all see that as a real threat to science [72].

In the following sections, a concise list of threats to reproducibility will be presented. We will start, however, with a short list of general principles that are widely considered useful or even necessary for reproducibility.

2.1.4.1 *General methods to ensure reproducibility*

Many authors in various domains advocated for actions and methodologies to ensure reproducibility of their research [91].

It is widely advocated and generally accepted as a good practice to release source code and other artifacts related to research and publications [106, 124]. Preferably a version control system such as Git or Subversion should be used as well, so that changes can be easily tracked.

Apart from the source code, it is important to document the details such as dependencies, platform configuration, information on how to run the experiment, all input data and the details of execution. Moreover, the researchers should publish raw data obtained from the experiments, as well as scripts used to postprocess them [43].

The use of existing tools and standards instead of reimplementing them is preferred as well. If non-standard procedures are used, the research is less likely to be reproducible. Various tools helping research will be presented in Section 2.2.

2.1.4.2 *Technical threats to reproducibility and countermeasures*

There are inherent, technical difficulties associated with experimental research in distributed systems and one has to identify them to know how to improve the process. In this section we discuss them and try to understand their impact on the difficulty of conducting experiments.

Large scale of experiments. Experiments are often of large scale which may introduce random or even erratic behavior due to surpassing inherent limits of experimental environment. This problem belongs not only to a system under the study, but also to all operations required to control the experiment as well.

This problem cannot be simply eliminated by not considering large-scale systems. It is often the very purpose of distributed systems to handle large-scale installations and to function correctly despite associated difficulties.

Experimental research in distributed systems does not have an integrated approach to deal with large-scale experiments, although simulation is *de facto* standard to achieve robust and repeatable experiments at large scale. In *in-situ* experiments, the early detection of failures (e. g., via monitoring) and checkpointing are other useful techniques shielding from the randomness of large-scale computer systems. One of the enablers of large-scale experiments are testbeds such as those described in Section 2.3.1.

Heterogeneity. Distributed systems are generally heterogeneous, since it is virtually impossible to have a large system with identical components at each level of infrastructure (e. g., disks, switches, routers, network cards, firmware, operating systems, software, etc.). These differences are hard to reproduce as they change often and both hardware and software evolve with time. Heterogeneity may cause failures (e. g., incompatible software versions) or unexpected behavior (e. g., network congestion due to asymmetric links) as well.
Reporting the minute details of platform and software are necessary to reproduce a heterogeneous platform. More importantly, however, heterogeneity should be avoided if possible so that the results are more general and do not require special hardware, for example.

Hardware faults. Hardware cannot be considered infallible, especially when working with large-scale systems where the number of components increases the probability of failure substantially. In fact, faults in large systems are more like a rule than an exception - studies have shown that the yearly failure rate of hardware disks can be as high as 6 % [149]. The faults may go unnoticed, which may lead in practice to incorrect conclusions from an experiment. Graphically, an experimenter may be unable to distinguish between the state X and any of the states A and B in the experimental cycle in Figure 3.
Monitoring of a platform can be used to discover failures and a strange behavior of a platform. Similarly, checkpointing may help to progress despite the presence of failures.

Software bugs. Similarly, problems present in the software, such as *bugs*, may imperceptibly influence the behavior of a system or reveal themselves unexpectedly only in particular conditions. .
There are no obvious ways to protect oneself from bugs in existing, large software stacks that researchers depend on. Following proper software engineering practices may at least eliminate bugs in the researcher's code and scripts.

Inability to identify relevant factors. Some distributed systems are so complex that it is virtually impossible to control, let alone to identify all relevant factors that may influence an experiment. In fact, even the most innocent parameters may have a quite substantial effect on the system behavior [133]. If these factors cannot be identified, they cannot be communicated as well, and others cannot prepare their reproduction study accordingly.

One has to be very careful to dismiss factors, even the most seemingly innocuous. Moreover, as mentioned in Section 2.1.5, some statistical techniques may be used to formally eliminate unimportant factors.

Errors in experimental process. Since experimenting is often regarded as a one-time activity, they are often prepared in a hasty manner and without much care, so that they "just work". Consequently, errors creep in due to hurry, repetition and low code reuse. These problems with experimentation process may not be observed in a timely fashion or even simply ignored by a researcher in a hurry.

Some of these problems can be addressed by using existing and well tested tools, methodologies and protocols.

Complexity of experimental process. Due to complexity of computing systems, experiments tend to be complex as well. Description of most experiments is therefore difficult to comprehend and important details may be unavailable to a researcher trying to reproduce an experiment.

Moreover, the common way of preparing experiments is a *bottom-up* approach where independent elements (e. g., tools, scripts, services) are linked together in an unstructured way. An alternative is a *top-down* approach where an experiment is designed as an abstract description whose particular implementation is supplied later. We will address this issue in much greater extent later in Chapter 4.

The similar issues of complexity in software programming have been addressed by many researchers in the past. A methodology that essentially introduced a top-down approach to programming, *structured programming*, was quite radical at some point [65]. Methodologies to progressively refine software projects have been proposed [205], but managing complexity of software projects remains a difficult and unsolved problem.

Little or no provenance. The imperfect description of experiments may not be enough to reproduce it, therefore it is useful to store the minute details of experiment execution, so that they can be analyzed later. These additional data is called *provenance* and is essential to understand the context of an experiment and its lack may be insurmountable barrier for reproducibility.

Provenance is mostly understood as a documentation of how experimental results were captured, transformed and summarized, but as we will see in Chapter 5, it can be generalized to a widely understood *context* of an experiment.

Provenance can be collected by many supporting tools, such as discussed in Section 2.2.3 and more general provenance collection techniques described in Section 5.2.

Configuration of the platform. The installation of software required to run experiments is a complex process itself. In fact, the deployment of software is a well-known problem that faces similar challenges as other distributed systems.

The difficulties of deploying a complex system consisting of many interacting nodes and services is well-known and many solutions exist to tackle it (see Section 2.2.2).

Time Another difficulty is posed by passing of physical time, which in the long run will make all experimental research unreproducible. The reasons are simple: technology to run experiments will not exist anymore, nor will software run on newer processor architectures [45].
For this reason, it seems, each attempt at reproducible research must define its target of long-term reproducibility. Some research projects may be satisfied with 2 years of prospective reproducibility, whereas others may target longer periods of time. Virtualization (Section 2.3.6) may be used to extend the lifespan of research work, but comes with its own reproducibility problems.

2.1.4.3 *Non-technical threats to reproducibility and countermeasures*

The state of reproducibility from a less technical point of view has been analyzed in computational sciences [70] and parallel computing [103]. There are many social and methodological reasons as why reproducible research is not common.

Time & effort. Making research reproducible takes time and effort that could be used for other purposes. Current methodologies for ensuring reproducibility are hence too bothersome and tedious when followed.
It can be argued however, that technical means, tools and methodologies can be used to ensure a good enough reproducibility without making the whole process longer.

Scientific community & publishing. Current research practices do not encourage reproducibility. There is no real incentive to make and publish reproducible research. The publishers may have no interest in verifying reproducibility of submitted research either.

Intellectual property & capital. Details of research may not be published simply because they are a valuable asset of a researcher or a company. Moreover, researchers may be afraid that their research will be plagiarized.

Reputation concerns. Researchers may withhold publication of details of their work (e. g., source code), because of not being particularly proud of its quality and hence undermining their reputation.

Some of these threats would arguably require changes to the existing publish model. There are already some modest efforts to not only encourage the distribution of source code with publications, but even mandating it. At the time of writing, *Science* requires its authors to submit complete source code, whereas *Nature* considers it best practice (similar rules apply to accompanying data).

Similarly, electronic journals have been started in various domains which require the authors to publish the source code demonstrating their findings

in demand. For instance, IPOL[1] is such a concerted effort in the domain of image processing and image analysis. No such journal exists, as far as the author can tell, for research in experimental distributed systems, the less in large-scale ones.

2.1.5 *Experimental design*

As every domain of science, computer science requires a design of experiments to prepare valid and efficient experiments. Experimental research in distributed systems presents some additional challenges. Extensive work on experimental design for performance evaluation has been done [107].

First, the aforementioned plenitude of influencing and interrelated factors, poses a non-trivial difficulty. The exhaustive exploration of parameter space is impossible for even simple systems which rules out *full factorial designs* of experiments. There exist techniques, however, that may be used to narrow down the analysis to the factors that have the principal effect on behavior of a system (*fractional designs*).

Second, most of performance evaluation questions are often not clearly defined and form multi-criteria optimization problems without any proper definition of the best solution. This makes quantifying properties of computer systems difficult and complicates fair comparisons between them.

Moreover, experiments in computer science are rarely statistically robust, for example experimental runs are rarely repeated to eliminate random variations (less than 31 % of papers [43]). This casts a doubt on the repeatability of results even within a single experiment. Moreover, even such an innocuous detail such as the choice of proper mean to summarize results may lead to skewed or even wrong results [82, 107].

2.2 TOOLS TO FACILITATE GENERAL RESEARCH

This section summarizes the most relevant general-purpose tools that help researchers in computer science with no particular focus on a specific domain. The main purpose of such tools is to improve the productivity of researchers that work with computers. They may improve it directly, by speeding up the overall process, or prospectively, for example by making it easier to carry on research by future researchers. Note that our list is minimalistic and by no means complete, since we focus on tools relevant to our work. These tools are important to our research, since we too aim at facilitating research in distributed systems. The approaches presented in this section are especially relevant to the description of experiments, which will be discussed in Chapter 4.

The contents of this section partially overlap with the domain referred to as *research programming* which is the form of programming activity that aims to write programs that obtain insights from data. This domain has been

1 http://www.ipol.im

the subject of an extensive analysis recently [99]. The same analysis distinguishes four different phases involved in the process: preparation, analysis, reflection and dissemination.

First, *literate programming* is presented in Section 2.2.1, followed by Section 2.2.2 which discusses tools used to configure and manage experimental environment. Finally, we discuss tools to capture context of experiments in Section 2.2.3.

2.2.1 *Documentation and notetaking*

A well-known approach, not only to research, but to programming in general, is known as *literate programming* [115]. It consists in writing the program with extensive documentation which is an *integral part* of the program. In fact, the Knuth's original approach intermingles code and comments in one single flow of text.

This approach has been generalized to research activities such as statistical analysis [118], *literate experimentation* [174] and planning and time management [166].

2.2.2 *Configuration management*

Configuration of a computing platform is clearly a difficult problem to researchers, but even more so to people whose job is to run and manage such systems on a daily basis. Indeed, the difficulty of performing the very first step, software and service deployment, is shared by both researchers and system administrators.

A common way to manage relationships between software is to explicitly track their dependencies. This is an approach that is a base for all popular Linux distributions such as Debian, Ubuntu (DEB packages), Red Hat, Fedora (RPM packages) and others. By having explicit dependency information, a system may install a software package with all its transitive dependencies automatically.

Since the dependencies can be satisfied in multiple ways, system configuration necessary to run the package at a particular version is not generally unique and hence unreproducible. Moreover, and it is a corollary of the previous observation, the build environment of a package is not reproducible as well, therefore there is no guarantee that the package will be identical when built even two times. There are however some promising efforts to address this problem[2].

Configuration management tools try to solve the problem by providing a high-level framework to describe system configuration. Puppet[3] and Chef[4], Ansible[5] are commonly used in automating administrative tasks such as software

2 https://wiki.debian.org/ReproducibleBuilds
3 https://puppetlabs.com/
4 http://www.opscode.com/chef/
5 http://www.ansible.com

provisioning and the configuration of operating systems. They simplify complex deployments by providing a clear, declarative description of a desired system state and then carrying out necessary steps to reach it. Moreover, they abstract differences between different operating systems, so that the same configuration scripts can be used everywhere. Such tools are commonly used by researchers to address the reproducibility of experimental platform [114].

Operating at even higher level are orchestration management tools, like Juju[6], which are designed to coordinate complex systems in flexible and reactive ways, usually in the cloud computing context. These tools can coordinate changes on multiple machines in tandem so that required invariants are preserved.

Traditional configuration management systems are oblivious to system state that is not specified explicitly. For example, if a user account information is not specified in Puppet recipes, the final system configuration may or may not have such an account. Consequently, it means that, at least with respect to unspecified facts, the final state need not be reproducible.

This is mitigated by the novel approach of the Nix package manager [68], which is based on the *functional* description of software and its dependencies. More precisely, every software package is a deterministic result of its configuration and its dependencies. The approach has many useful features, in particular it allows installing multiple versions of the same software (something usually impossible with traditional package managers) or have fully atomic system upgrades. Nix has been used to built a fully functional Linux distribution called NixOS [69].

2.2.3 *Capturing context*

Given the complexity of experiments, it is crucial for reproducibility to capture experimental context, that is, to store information about an environment where the given experiment has been executed and all related details, such as input files, environment variables, current state of codebase, etc. Experimenters can take advantage of version control systems (e. g., Git, Subversion) or more sophisticated tools like Sumatra [57], which aim at recording the scientific context in which the given experiment was performed. Such tools store historical information about experiment runs and their parameters and results. This approach documents research with a low burden and enables meaningful comparing of different experiment runs.

Creation of ontologies describing the experimental platform has been proposed as well [164] as a method for helping reproducibility in computational sciences.

A related idea is that of *reproducible papers*, that is, scientific publications that contain all relevant details on how the final publication was created. This includes raw data, scientific protocols, configuration details, software versions possibly in interactive and easy-to-use way [128].

6 https://juju.ubuntu.com/

2.3 TOOLS TO FACILITATE RESEARCH IN DISTRIBUTED SYSTEMS

This section completes the previous one by focusing on tools that are particularly useful for research in distributed systems and less so for general computer science. Presented techniques will be especially relevant to low-level features (command execution, checkpointing, fault tolerance, testbed types, etc.) of our experiment control engine which will be presented in Chapter 4.

We start with an overview of experimental testbeds in Section 2.3.1. This is followed by the discussion of monitoring in Section 2.3.2. Then, workload generation is introduced in Section 2.3.3 and tools helping with parametric studies in Section 2.3.4. Fault tolerance and methods to achieve it are the subject of Section 2.3.5. Finally, we discuss both virtualization and containerization in Section 2.3.6.

2.3.1 *Testbeds*

To meet the needs of *in-situ* experimenting, researchers designed and convinced their funding bodies to build large systems dedicated to research in networking, protocols as well as distributed and parallel computing. Many *testbeds* have been built for this precise reason and even design guidelines on how such systems should be built have been identified [172]. The increasing interest in testbeds led recently to special journal issues dedicated fully to them [179].

Testbeds differ in many aspects, in particular they *focus* on different domains of experimental research, but also use different *approaches*. It is impossible to fully separate them, yet some general trends exist.

The scientific focus of testbeds is represented by domains such as: networking, distributed and parallel computing (including cloud computing), services and applications, or wireless communication. This is unsurprising, considering that precisely research in these domains requires extensive empirical verification.

In terms of approach, there can be three main, but partially overlapping approaches distinguished. Testbeds such as Emulab and ORBIT rely on network emulation (*emulation* testbeds) to produce repeatable conditions. Such testbeds tend to be a single-site installations since emulation of the network is coordinated centrally. Testbeds giving a direct and unrestricted access to bare-metal machines (*in-situ* testbeds) tend to be more distributed, usually consisting of a few sites connected with reliable, dedicated, but generally heterogeneous network. The classical example of such a testbed is Grid'5000. The last group of testbeds (*virtualized* testbeds) builds on virtualization technologies applied to the computing power and network resources. This is the most recent approach that tries to exploit the advantages of virtualization, that is, the flexibility and virtually unlimited resources. Examples of such testbeds include Open Cirrus, and more recent CloudLab and Chameleon.

The approach of a testbed does not necessarily imply its focus. For example, there is nothing stopping the use of a testbed such as Grid'5000 for cloud

computing. Moreover, it is also the case that the virtualized testbeds offer bare-metal provisioning of physical machines (e. g., Chameleon).

Another approach, or rather a meta-approach, that gains a lot of traction recently, consists in federating testbeds in such a way that resources of different testbeds can be used simultaneously and transparently. The enabling technology for such federation is Software Defined Networking. An example of such a testbed federation is GENI.

Below is a short summary of the most prominent testbeds ordered alphabetically, followed by *meta-testbeds* that aggregate and federate other testbeds. Considering the rapid creation and evolution of testbeds, this list is arguably not exhaustive nor up-to-date.

DAS (Distributed ASCI Supercomputer[7]) is a Dutch wide-area distributed system designed by the Advanced School for Computing and Imaging (ASCI). It is the fifth generation of this research project which distinguishably employs various HPC accelerators (e. g., GPUs) and novel wide-area interconnect based on light paths. Its current iteration is called DAS-5 (2015), but was preceded by other systems going back to 1997. DAS does not offer a dedicated tool to control experiments, however it provides a number of tools to help with deployment, discovering problems and scheduling.

EMULAB [202] is a network testbed that allows one to specify an arbitrary network topology and characteristics such as latency and bandwidth of network links. It is achieved thanks to dedicated network nodes running the Dummynet [41] emulator. These nodes do not explicitly participate in the experiment, but are located between the nodes and shape the network traffic according to the configuration. This feature ensures high-performance emulation of a predictable and repeatable environment for experiments, at least in terms of network configuration. Users have access to the "root" account on testbed nodes and hence can easily customize their software and configuration. Emulab comes with a dedicated tool to control experiments (see Section 3.4.3). The initial and main Emulab site is located at the University of Utah.

More recent work tries to improve Emulab's environment repeatability understood as an ability to recreate a state of the platform [34, 158]. APT (Adaptable Profile-driven Testbed) uses advanced network emulation of Emulab and snapshotting of disk images to persistently store experimental environment for later use and easy referencing. There are also recent and promising efforts to leverage Emulab for cloud computing research[8].

Emulab is an open-source solution and has been deployed besides its main site. In particular, it has been deployed as an isolated testbed for potentially dangerous research [129].

7 http://www.cs.vu.nl/das5/

8 https://www.cloudlab.us

G-LAB [169, 170] (German-Lab) is an experimental facility of universities in six German cities: Würzburg, Kaiserslautern, Berlin, München, Karlsruhe, and Darmstadt. G-Lab consists of approximately 200 general-purpose nodes connected with high-quality links between sites. The experimenters can use bare-metal machines with custom boot images, but the default way of interacting is to use virtualization (in particular via a local PlanetLab installation). Similarly to Emulab, G-Lab can create arbitrary topologies which may also span multiple sites.

GRID'5000 [39] is an experimental testbed dedicated to the study of large-scale parallel and distributed systems. It is a highly configurable experimental platform with some unique features. For example, a customized operating system (e. g., with a custom kernel) can be installed and full "root" rights are available. The platform offers a REST API to control reservations, but does not provide dedicated tools to control experiments. However, the nodes can be monitored during the experiment using a simple API. The multiple sites of the testbed are connected with fast, reliable and dedicated network. Grid'5000 provided a model used by other testbeds, e. g., FutureGrid [84]. Recently, FutureGrid has evolved into a more cloud-oriented testbed, Chameleon[9].

The research profile of Grid'5000 is not specified, but due to the low access to resources, it focuses mainly on system research, protocols and networking. Interestingly, the computing power of Grid'5000 has been also used to factor a 768-bit long RSA key, the current factorization record to date [113].

IOT-LAB, CORTEXLAB AND NITOS-LAB form a group of related testbeds used to experiment with wireless sensors (IOT-LAB[10]), cognitive radio (CorteXlab[11]) and other wireless technologies such as WiFi, Bluetooth or ZigBee (NITOS-Lab).

The testbeds are managed by the OneLab consortium[12] and financed from the same sources. Some software used by these testbeds originated from Grid'5000 project.

JGN-X, STARBED3 are testbeds focusing on research in next-generation networks and protocols. The JGN[13] (Japan Gigabit Network) testbed enables experiments on multiple networking layers: layer 3 (e. g., IPv4 and IPv6), layer 2 (non-IP Ethernet) and also layer 1 (wide-area optical network). The related VNode project provides network virtualization with a novel isolation technique inspired by PlanetLab slices.

StartBED (Hokuriki Research Center), on the other hand, uses a scripting language to model the platform and VMWare virtualization technology to deploy it on general-purpose computers.

9 http://www.chameleoncloud.org/
10 https://www.iot-lab.info/
11 http://www.cortexlab.fr/
12 https://www.onelab.eu/
13 http://www.jgn.nict.go.jp/english/index.html

OPEN CIRRUS [11] is a cloud computing testbed and the result of collaboration of many entities including such companies as Yahoo, Intel, HP, as well as universities. It is a testbed distributed around the globe at multiple sites. Each site has different research focus, but they are all united by a single authentication system, monitoring resource tracking and other services. The motivation behind Open Cirrus is to provide low-level hardware access, heterogeneity of environment and large data-storage capabilities. It is therefore a platform not only for "application studies", but also for more basic "system research".

ORBIT [143, 157] is a radio grid testbed for scalable and reproducible evaluation of next-generation wireless network protocols. It offers a novel approach involving a large grid of 400 radio nodes which can be dynamically interconnected into arbitrary topologies with reproducible wireless channel models. Due to inherent variability of the wireless communication, the designers of ORBIT made a special effort to address possible reproducibility issues. A dedicated tool to run experiments with ORBIT platform is OMF (see Section 3.4.6).

PLANETLAB [11, 145] is a globally distributed platform for developing, deploying and accessing planetary-scale network services. It consists of geographically distributed nodes running a light, virtualized environment (i. e., OpenVZ Linux which is a form of containerization). Network between PlanetLab's nodes is exposed at layer-3 (network level) and since nodes are connected over the standard Internet, links between them are unreliable. PlanetLab offers Plush (see Section 3.4.4) for experiment control.

The testbed offers network emulation via use of Dummynet [40], and measurement infrastructure to obtain various metrics, observe topology of network and get notifications about important events, such as changes of link properties [24].

There is the concern of low reproducibility of research done in PlanetLab, mainly due to its best-effort capabilities, however some concerns are unfounded [177].

PlanetLab is a successful model of a testbed and there exist many different variants and installations of it. Recently some of them try to reach layers below the network layer: VINI [18] (Virtual Network Infrastructure), VICCI [146] and GpENI [127] (Great Plains Environment for Network Innovation).

The above presentation about testbeds is summarized in Table 2 (features and properties).

To increase the size and the list of features offered by network testbeds, researchers started to federate them in unified systems. Such federations offer unprecedented features, heterogeneity and configurability. The most influential testbed federation is GENI [21] (Global Environment for Networking Innovation), a distributed virtual laboratory sponsored by the U.S. National

Table 2: Summary of the most important testbeds. The dates in parentheses approximate the dates when testbeds became available (based on the analysis of online resources and publications).

	Characteristics			
	Focus	Approach	Participants	Distribution
DAS (first in 1997, last in 2015)	Services & algorithms	Clusters in a wide-area network	NOSR[1] and other bodies	Six clusters (around 200 nodes)
EMULAB (2000)	Networking & systems	Bare-metal nodes with network emulation	University of Utah	One site (more than 300 nodes)
G-LAB (2009)	New technologies and applications	Multiple sites managed together	German universities with many partners	6 sites
GRID'5000 (2006)	Parallel and distributed computing	Multiple sites of bare-metal nodes	GIS[2] and other bodies	11 sites (1041 nodes)
IOT-LAB, CORTEXLAB, NITOS-LAB (2014)	Wireless communication	Large scale infrastructure to test for testing wireless devices	OneLab consortium	6 sites (2728 wireless nodes)
JGN-X, STARBED3 (2011)	Next-generation networks	Federation of testbeds	NICT[3]	1398 nodes in various groups
OPEN CIRRUS (2010)	Systems & services	Federation of heterogeneous data centers	Many companies, institutes & universities	10 sites
ORBIT (2005)	Wireless communication	Reproducible wireless network emulation	NSF[4] and other bodies	A single site (400 wireless nodes)
PLANETLAB (2002)	Systems & services	Nodes hosted by research institutions	Various universities and organizations	More than 700 nodes worldwide
GENI (2009)	Networking & systems	Large federation of testbeds & native sites	Various organizations & testbeds	Hard to estimate

[1] Netherlands Organization for Scientific Research
[2] Scientific Interest Group
[3] National Institute of Information and Communications Technology
[4] National Science Foundation

Science Foundation for development, deployment and validation of transformative, at-scale concepts in network science, services and security. GENI relies on the concept of virtualization of computing power and advanced,

flexible network configuration and emulation. GENI builds on previous advances in testbed design, for example it embraces the PlanetLab's concept of slices attributed to experiments.

The GENI federation not only uses dedicated racks for virtualized computation and storage resources, but it is also known for successfully integrating resources of other testbeds such as PlanetLab, Emulab, ORBIT, G-Lab, etc. They are all available via unified API that abstracts details of each site. By using technologies related to software-defined networking (SDN) such as OpenFlow switches or virtualized cellular wireless communication (WiMaX), GENI is currently able to create transparent layer-2 networks from geographically and administratively distributed sites. Assigned network resources may offer Quality Of Service (QoS) guarantees thanks to network slicing [171].

GENI has been used for realistic experimentation and research in areas such as novel routing strategies, future Internet protocols, software-defined networking, among others. GENI does not have a unified platform to control experiments due to its extreme heterogeneity. Its users have to use an API and other tools to control the experiments instead. However, there are multiple tools used for resource discovery and reservation.

Fed4Fire is another research testbed federation funded under European Union's Seventh Framework Programme (FP7) addressing Future Internet Research and Experimentation (FIRE). It includes over 15 different facilities from around the world. Like GENI, Fed4Fire is dedicated to experimentally driven research with wired, wireless, OpenFlow and cloud-computing testbeds. Thanks to the federated control and services, the experiments can involve hardware from multiple testbeds simultaneously. Fed4Fire is also interoperable with the GENI federation [198].

Different testbeds usually use a dedicated system that manages testbed resources in face of multiple users interacting at the same time, but there is some overlapping (e. g., see Grid'5000 and IOT-Lab above). There exist, especially in Grid context, management middleware systems that manage access to resources, such as Globus [83].

2.3.2 *Monitoring*

A useful element of each experiment is its monitoring. It can be used to collect the overview of a system while it was executing an experiment. Obtained data can be a principal result of the experiment, but also all additional data that can lead to deeper insights and as a way to detect that the system did not run in unexpected manner.

There exist a lot of software used to monitor production systems, collect historical information and raise alerts in presence of unexpected events. One has to be wary, however, since too extensive and intrusive monitoring may have a significant effect on the behavior of the system. This in turn may lead to situations where a monitored system shows a behavior that is not present without monitoring.

Some of the testbeds presented above include monitoring as a standard feature. PlanetLab's nodes are monitored and services exist to find nodes that are available [24]. Monitoring (via instrumentation) is also an integral part of ORBIT (via OMF) [156].

The collection of large amount of monitoring data from multiple sources is a difficult problem. Special infrastructures, languages and methods have been devised to collect and correlate performance measurements [33].

2.3.3 *Workload generation*

Another useful aspect in some research studies is to be able to produce representative workload during an experiment. For example, while testing load balancing techniques, a researcher may want to test it under a realistic load.

Tools to generate load (or traffic) are readily available for many situations and exist in many incarnations. As a bare minimum, such tools offer a way to generate a constant workload (e. g., a constant number of HTTP requests per second), sometimes with options to change in with time. Such tools are essentially benchmarks and may be considered unrealistic.

Another method consists in using real, historical information that was collected on a real system and replaying it [63]. Such workload traces are sometimes archived and made available publicly [14] and even standardized for the narrowed and precise types of loads.

2.3.4 *Tools supporting parameter studies*

By a *parameter study*, we understand an experiment that focuses on discovering the influence of parameters (or factors) on the behavior of the system.

Tools like ZENTURIO [150] and Nimrod [1] help experimenters to manage the execution of parameter studies on cluster and Grid infrastructures. Both tools cover activities like the set up of the infrastructure to use, collection and analysis of results. ZENTURIO offers a more generic parametrization, making it suitable for studying parallel applications under different scenarios where different parameters can be changed (e. g., application input, number of nodes, type of network interconnection, etc.). Even though Nimrod parametrization is restricted to application input files, a relevant feature is the automation of the design of fractional factorial experiments. NXE [98] scripts the execution of several steps of the experimental workflow from the reservation of resources in a specific platform to the analysis of collected logs. The whole experiment scenario is described using XML which is composed of three parts: topology, configuration and scenario. All the interactions with resources and applications are wrapped using bash scripts. NXE is mainly dedicated to the evaluation of network protocols.

14 http://www.cs.huji.ac.il/labs/parallel/workload/

2.3.5 *Checkpointing and fault tolerance*

Checkpointing is a general technique that provides fault tolerance in presence of failures in a distributed system. Various methods and protocols has been proposed by researchers that provide fast and scalable recovery [78]. One of the main difficulties is to ensure that checkpoints represent a consistent view of a system as a whole and that restarting from it will not affect the final outcome. In particular, in-transit network packets must be accounted for.

Checkpointing is used by researchers working with *in-situ* distributed systems as well [34]. The method consists in using a lightweight hypervisor (Xen in this case) that can stop the system globally, and a network emulation layer (based on Dummynet) that takes care of network packets in transit.

2.3.6 *Virtualization and containerization*

With the emergence of robust, efficient and cheap virtualization, scientists turn to cloud computing infrastructures as a viable experimentation platform [74, 102]. There are many technologies and vendors who offer virtualization services. Linux Xen [15], for example, was employed by Amazon in their successful Amazon Web Services platform. Similarly, Linux KVM is another open source solution and VMWare[15] is a well-known commercial vendor.

Among the well-known commercial cloud providers one can find Amazon EC2[16], Windows Azure[17], Google Cloud Platform[18], and many others. There are non-commercial, open-source solutions available as well, such as OpenStack[19].

The services offered by cloud solutions can be split into separate groups with increasingly deeper access to the underlying infrastructure: *Software as a Service* (SaaS, i. e., only an application is provided), *Platform as a Service* (PaaS, i. e., a ready-made system like a database is provided), *Infrastructure as a Service* (IaaS, i. e., a whole computing platform is provided) and, finally, *Network as a Service* (NaaS, i. e., additionally network can be virtualized) [51]. Arguably the researchers interested in networking or system research will only be interested in the last two types of services.

Even though the development of cloud computing solutions was not inspired by the need of a research platform, scalability and elasticity offered by those make it an attractive solution for science. A framework oriented toward reproducible research on such infrastructures has been proposed [114].

Although virtualization is mostly associated with computing power, more and more focus is given to network virtualization [148]. There are even efforts to virtualize high-performance networking elements, such as InfiniBand interconnect [161].

15 http://www.vmware.com/
16 http://aws.amazon.com/ec2/
17 http://www.windowsazure.com/
18 https://cloud.google.com/
19 http://www.openstack.org/

There are many advantages that cloud platforms can offer to science:

- cheap and instant access to virtually unlimited resources,
- no maintenance and associated costs,
- progressively cheaper costs of cloud services,
- easy and cheap storage of datasets (cf. Amazon Public Data Sets[20]),
- ease of collaboration.

There are also the dangers and difficulties of using cloud computing for research and the use of outsourced infrastructure in general:

- questionable reproducibility of research (e. g., the cloud resources are arbitrarily shared or even over-reserved without users even knowing),
- missing experimental functionality such as network virtualization or low-level access due to a traditional focus on supporting Web services,
- performance concerns about virtualized environment,
- low applicability to some domains of computer science such as high-performance computing, since most cloud offerings rely on standard architectures and components,
- vendor lock-in due to low standardization of cloud platforms,
- security concerns about data stored in remote locations.

Some of these issues can be resolved with advanced cryptography [92], private clouds or research testbeds such as those presented in Section 2.3.1.

Containerization is an alternative and lightweight approach to virtualization. Whereas virtualization focuses and separating computation and networking fully from the host system, containerization uses software separation, usually through mechanisms implemented in an operating system.

Basic containerization techniques have been present for a long time in research community to improve management of software. Modules [88] have been used to separate and activate scientific software with dependency management.

The performance of containers was found to be on par with native execution [81]. This includes both start-up time of containers (which can be measured in milliseconds) and overall performance. There is however overhead associated with copy-on-write (COW) techniques used for storage by containerization methods. Containerization has been used to scale evaluation of peer-to-peer (P2P) systems [13].

It has been observed that containerization is traditionally tied to *Platform as a Service* (PaaS), whereas virtualization more often to *Infrastructure as a Service* (IaaS) solutions [81]. The same authors argue that due to hard separation of host and guest systems in virtualization, containerization will always offer better performance (e. g., due to inability of virtualization to account for NUMA memory hierarchy in the host).

20 https://aws.amazon.com/datasets/

Containerization sees a lot of interest in system administration and software deployment. For example Docker[21] attained much interest from *DevOps* community as a lightweight and powerful system for software and service deployment. The features of Docker-like containerization have been also appreciated by research community that seeks reproducibility [23].

2.3.7 *Efficient and scalable command execution*

Efficient command execution on the large number of nodes is a very practical problem, which is not entirely solved. As it is going to be an important building block in our approach, we dedicate a few paragraphs to this subject.

The goal of command execution is to run a command on multiple nodes efficiently and correctly. Various technical problems arise as the number of nodes increases, mostly due to the limited scalability of network technologies and protocols. All presented approaches use SSH to access nodes and execute commands on them, and therefore rely on TCP and the supporting protocol stack. Standard command execution uses relatively low amount of effective bandwidth (both input and output of commands is rather small), but large amount of control traffic required to coordinate the whole process.

A naive way consists in executing a command via SSH on each node in sequence. It suffers from linear time with respect to the number of nodes, but is otherwise robust, as it does not stress the network at all. A simple improvement is obtained by having a limited number of parallel command executions (a *window*). The optimal size of the window depends on the network details and on the CPU resources of the machine that initiated the execution. Until the use of resource is not saturated, the theoretical speedup is proportional to the window size. This method is used notably in *clush*[22].

TakTuk [48] is a tool for large-scale remote execution and file distribution using an efficient tree-like topology constructed over the set of nodes (the arity of the tree is configurable). If the network remains unsaturated and protocols scale properly, the theoretical time of the execution is logarithmic with respect to the number of nodes.

Parallel command execution is sometimes a feature of configuration management tools (presented in Section 2.2.2).

The efficient distribution of data is a related problem. It consists in distributing a single piece of data on all involved nodes. Contrary to the command execution, the process may involve large volumes of data transferred from the initiating node. The methods of command execution can be used for this purpose, but usually dedicated methods are more efficient.

Kascade [123] builds a topology-aware chain of TCP connections between nodes, transfers data in a pipeline and offers nearly optimal performance in the best case (no faults). It is however susceptible to slow links that limit the bandwidth of the method.

21 https://www.docker.com/
22 https://cea-hpc.github.io/clustershell/

UDPCast[23] is a file transfer tool that can send data simultaneously to many destinations on a LAN using IP multicast. It features various methods and enables a fine-grained tuning of parameters. It is however unfair to other kind of traffic (due to no congestion avoidance in UDP multicast) and may disrupt unrelated communication.

2.4 SCIENTIFIC WORKFLOWS

This section summarizes the current knowledge about scientific workflows, a set of methods and models that have seen a wide adoption and success in computational sciences. Scientific workflows and scientific workflow systems are of great relevance to our approach since they share similar purpose, structure and goals. There are nevertheless important differences and hence the scientific workflows will be used as a comparison, not as a direct inspiration of our approach. They will be also very important in our work on provenance in Chapter 5.

First, we turn to the discussion on the purpose and general use of scientific workflows in Section 2.4.1. The most popular scientific workflow systems are presented in Section 2.4.2. Then, in Section 2.4.3, the group of approaches similar, but distinctively different from scientific workflows, is presented. In Section 2.4.4, the expressive power of scientific workflows is analyzed. Section 2.4.5 discusses the advantages and difficulties of analyzing scientific workflows. Finally, Section 2.4.6 explores how, or if at all, scientific workflow systems are used by computer scientists working with distributed systems.

2.4.1 *Purpose of scientific workflows*

The aim of scientific workflow systems is the automation of scientific processes that a scientist may go through to get publishable results from raw data. The main objective is to communicate analytic procedures repeatedly with minimal effort, enabling the collaboration on conducting large, data-processing, scientific experiments [180]. Domains that routinely use scientific workflows include biology, astronomy, medicine and other natural sciences. This is due to useful features that such systems provide [61].

More concretely, the following desiderata for existence of scientific workflows have been identified [120]:

- Seamless access to resources and services
- Service composition & reuse and workflow design
- Scalability
- Detached execution
- Reliability and fault-tolerance
- User-interaction
- Smart re-runs

23 https://www.udpcast.linux.lu/

- Smart (semantic) links
- Data provenance

More detailed requirements have been proposed too [126].

As one can see, scientific workflows are designed specifically to compose and execute a series of computational or data manipulation steps. Normally, those systems are provided with GUIs that enable non-expert users to relatively easily construct their applications as a visual graph. Workflows consist of computational steps connected with links that conceptually pass output of one step to another one (see Figure 4 for example).

Most scientific workflows do not permit cycles in their dependency graphs and therefore such workflows are directed acyclic graphs (DAGs). The simple and constrained model remains expressive enough to support even sophisticated computational experiments.

Some authors raise concerns about practical reproducibility of scientific workflows [20]. It is reported that nearly 80 % of workflows cannot be reproduced. This "decay" of ability to reproduce workflows is mostly due to the lack of resources required for workflow execution.

Despite much success in computational sciences, the scientific workflows see virtually no use in the computer science community. The reasons behind that situation will be analyzed in Section 2.4.4 and Section 2.4.6.

2.4.2 *Existing scientific workflow systems*

In this section, we will present the most popular scientific workflow systems used by researchers. Each system will be presented and its distinguishable features mentioned.

Scientific workflow systems have been compared and classified in a few studies [16, 180, 208].

ASKALON [80] has as its goal a simplified development of Grid applications. It offers a novel environment based on a set of tools and methodologies that make development of Grid applications easier. In particular, ASKALON makes the optimization of real applications a routine task and everyday practice. The primary use of ASKALON is to run and control grid applications that consume and transform data. It has been used to manage applications from domains such as materials science, hydrology, meteorology and astrophysics.

The workflows in ASKALON are dataflows expressed in Abstract Grid Workflow Language (AGWL [79]) which mixes data-flow and control-flow constructs (e. g., parallel-for). The workflows are instantiated by a workflow enaction engine which deploys the workflow on a grid. The workflow execution is then monitored.

GALAXY [95] is an open-source platform for data intensive biomedical research, such as genomic research. It has constructs for filtering, grouping, sorting and extracting data items from available datasets or from

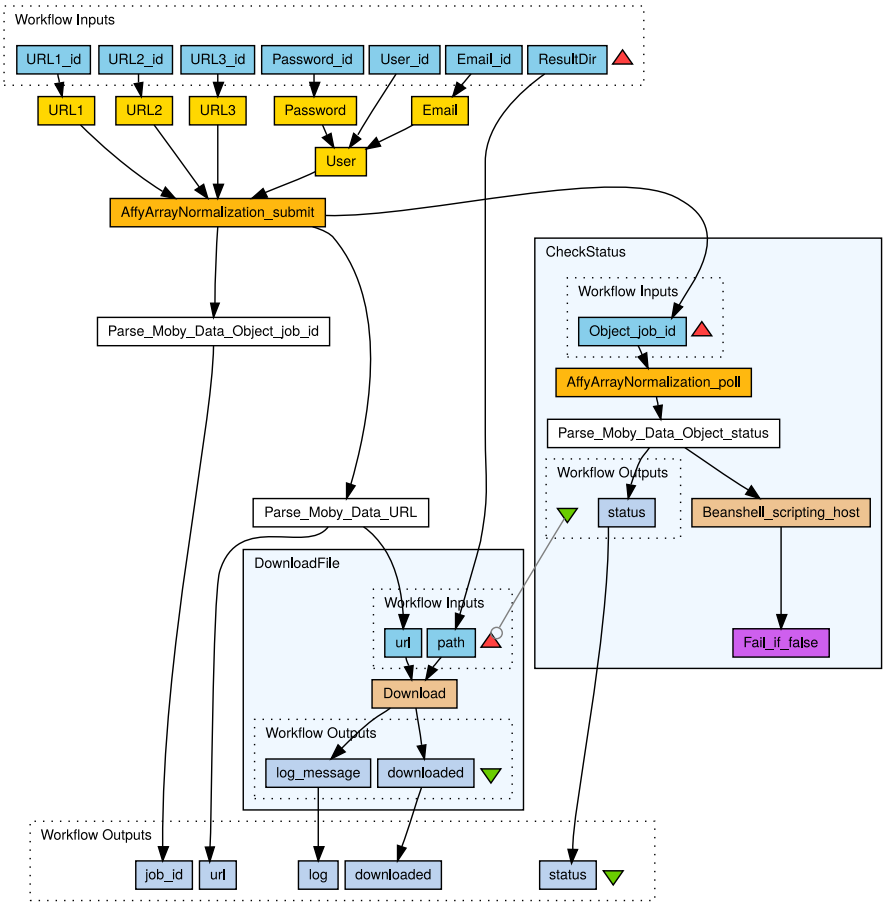


Figure 4: Example of a scientific workflow from the Taverna scientific workflow system. The workflow accesses an asynchronous web service that normalizes raw data from a DNA microarray.

user-uploaded ones. Available tools can be chained into more complex workflows that are executed on Grid systems.

The end users of Galaxy need not know the details of used tools as they are encapsulated as reusable abstractions. Moreover, the provenance of data in workflow is saved, so that the same analysis can be in theory reproduced later.

KEPLER [120] is another scientific workflow system that uses *actor-oriented modeling* that governs interactions between components. A scientific workflow is the composition of independent *actors* communicating via interfaces called *ports*. Basically, the incoming data that comes through *input ports* are processed by the actor and sent further on via *output ports*. There is also a global *director* object that defines detailed semantics how actors are orchestrated, in particular on how they are parallelized.

Kepler can model general workflow patterns including some control-flow ones. Kepler has been used to implement the COMAD model [126]

which is a formal framework with useful properties for users, in particular for providing provenance information for data objects.

PEGASUS [62] is a framework for mapping complex scientific workflows onto distributed systems. It is able to dynamically remap resources in case of failures and continue despite their presence. Moreover, it can checkpoint workflow execution at some stage to restart it later.

Among other useful features of Pegasus are: multi-level provenance collection and support for cloud platforms as runtime distributed systems [12]. Moreover, Pegasus explores intelligent and automatic design of computational experiments [94].

TAVERNA [138, 206] is a tool for the composition and enactment of scientific workflows. Although its primary focus is on bioinformatics research, it is nevertheless a general tool. A standard Taverna workflow consists of steps interacting with Web Services, requesting and downloading data, and processing them in structured pipelines.

With Taverna, scientists have access to different tools and resources that are freely available from a large range of life science institutions. Once constructed, the workflows are reusable bioinformatics protocols that can be shared, reused and repurposed. The repository of public workflows is available[24].

VISTRAILS [19, 35, 87] is a framework for enabling interactive multiple-view visualizations of scientific data. It introduces a *vistrail* which is an abstract pipeline description that can be instantiated and executed to obtain visualizations of data. Intermediate results can be cached, a necessity for rapid researcher-driven exploration.

Distinguishably, VisTrails focuses on visualization of data as opposed to pure data processing. As a result, it also provides built-in and easy-to-use functionality for the exploration of parameter space. Moreover, VisTrails can trace modifications to the workflow itself, enabling the researchers to also explore the space of possible workflow-based visualizations.

This short overview of scientific workflow systems shows distinctive features offered by them. First, the main purpose of all systems is to aid the execution of data processing steps, usually in the grid context. Second, the models used by the systems are mostly data-centric and scientific workflows tend to be acyclic. Finally, the scientific workflow systems have a good support for provenance collection.

2.4.3 *Workflow-like approaches*

There are some approaches to data processing that bear similarities to scientific workflows.

24 http://www.myexperiment.org

At the very basic level, *Makefiles* offer a similar type of model consisting of *targets* that generate new files taking into account data dependency between them. Results of a target may be cached so that future executions can use them instead of repeating their creation again. Moreover, Makefile execution can be parallelized, just as scientific workflows, although only within a single machine. This model has been adopted to produce reproducible results and figures from data [168].

There are examples of parallel programming languages for parallel computing, such as Swift [204]. A program written in Swift is executed in parallel and groups of steps executed in order are inferred from data dependencies. Similarly to scientific workflows, Swift programs can be executed on a grid system, and its engine is responsible for data movement and job submission. Coincidentally, our implementation presented in Chapter 4, bears some similarities with parallel programming languages.

Other data-processing models restrict expressiveness even more than scientific workflows, for example MapReduce [59], where only two stages are present (called, unsurprisingly, *map* and *reduce* phases). It has the effect of making it easier to program such a computational process, improving its performance, and ensuring high fault-tolerance. On the other hand, the model cannot be used efficiently in the general case. These drawbacks have been addressed in later research and a general dataflow, MapReduce-like computation is possible [207].

Finally, Airavata [122] represents another idea similar to scientific workflows. It is a framework used to build *science gateways*, i. e., well-defined access methods to otherwise incompatible scientific applications and systems. Airavata can be used to compose, manage, execute and monitor them in orchestrated way. As such, it also bears resemblance to business workflow systems which are discussed later on.

2.4.4 *Expressiveness of scientific workflows*

As has been shown, scientific workflows differ in their functionality, but fundamentally they are all based on the same restricted model that limits their expressiveness. The expressive power of scientific workflows has been studied and is well understood [52, 61].

First of all, scientific workflow systems are generally restricted to acyclic dataflows. Moreover, scientific workflows are usually *static* in the sense that their structure is not subject to dynamic changes during execution (and if it is, then that fact is hidden from the abstract workflow representation).

2.4.5 *Analysis of scientific workflows*

The rigid structure of scientific workflows enables researchers to perform "meta-analyses", that is, analyses of a form that real-life scientific workflows take. It is useful in many aspects: workflows can be checked for correctness, simplified or optimized. The basic verification of correctness is present in

every scientific workflow system and usually consists of a type system that verifies if the inputs and outputs of each workflow step are of the correct type. The design of a general workflow analysis tool has been proposed [53].

Another interesting application is studying similarities and differences between scientific workflows from the whole domain [90]. Such analyses led to interesting generalizations such as the confirmation of a hypothesis that scientific workflows are data-centric processes with occasional domain-specific focus on data visualization (e. g., social network analysis). It has been also observed that *data preparation* (i. e., initial transformation of input data to useful representation) accounts for more than 50 % of workflow structure.

2.4.6 *Scientific workflows in computer science*

It may come as a surprise that scientific workflows are not used to control computer science experiments. Despite advantages brought by them, they are not, at least in the current form, applicable in experimental computer science.

First of all, the execution of scientific workflows is by no means a data collection process, contrary to experimentation with a computer system. In fact, the execution of a scientific workflow presupposes existence of input data which have been collected beforehand and is essentially a data processing step. In standard scientific workflows this may be a dataset of astrophysical images, genetic markers or MRI scans, all of which have been collected in a separate process. For this reason, scientific workflows do not concern themselves with data collection *per se*.

Second, the acyclic, data-centric model [90] used by scientific workflows does not match the reality of experimental computer science where steps are often repeated many and an undefined *a priori* number of times (e. g., the number of iterations may depend on the number of physical machines available). These systems lack complex control patterns that are quite common in experimentation and may be therefore detrimental and limiting.

Next, by noting that scientific workflows are a tool to perform computational steps on input data, results of scientific workflow execution should not depend on an underlying platform. Whether a workflow is executed with 10 machines or 1000 machines should not matter for conclusions drawn from results, however time required to obtain them may be significantly different in both cases. It is in fact the very goal of scientific workflows to execute a computational workflow without its user knowing the gory details of his computing system. How this computation is performed physically is a detail that scientific workflows try to hide (knowledge about such details can be used to find problems and perform optimizations, however). This is in a strong contrast with experimental computer science where details on how a given system computes is *the very subject of the study*.

Finally, although it is a consequence of previous observations, scientific workflows do not help with necessary steps such as deployment of complex software and distributed systems. Moreover, a low-level access to machines

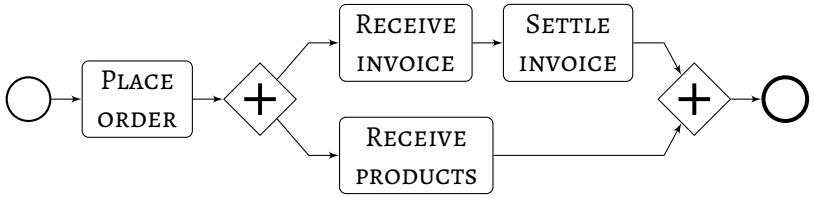


Figure 5: Example of a formalized business process. The model uses workflows to represent the real-life process.

and network may be restricted or just impossible due to abstractions offered by scientific workflows.

On the other hand, scientific workflows could be used in principle to process data collected during an experiment with a distributed system. Whether this application of scientific workflows brings any benefits, remains to be seen.

2.5 BUSINESS PROCESS MANAGEMENT

This section is dedicated to the state of knowledge about *business processes* and *workflow models of them* as they are known in the domain of Business Process Management. Workflow patterns identified in the domain are the principal structure of proposed model for representing experiments, as will be shown in Chapter 4. Moreover, the structure of workflows will have great bearing on the model of provenance presented in Chapter 5.

In Section 2.5.1 we introduce the basic notions and definitions. Then, the purpose of workflows in BPM is explained in Section 2.5.2. In Section 2.5.3, a few examples of real BPM systems are presented. Process mining and other techniques of analyzing business processes are presented in Section 2.5.4. We finish with Section 2.5.5 that explains why BPM systems do not see much use in research on distributed systems.

2.5.1 *Definitions*

Business processes consist of a set of activities that are performed in coordination in organizational and technical environment to achieve a business goal [200]. The classic example is the process of buying a product (the goal being the order and shipment of the product), as shown in Figure 5. Real-life business processes may possess a formalized description, but often are conducted informally. It is the goal of Business Process Management and associated techniques, to help formalize and hence understand business processes.

The most established modeling language for business processes is based on *control-flows*. For the purpose of this work, we define control-flows as workflows consisting of a set of activities that are performed under causal, temporal and spatial constraints to achieve a specific goal. In this work this goal will be understood as collection of experimental results. This definition is closely

related to the one of the workflows in the domain of Business Process Modeling [116] and in this work both terms will be interchangeably. Differences between control-flows and data-flows are studied quite extensively [16, 120], including expressiveness of both formalisms [52]. Both types of workflows cannot be strictly separated, that is, some scientific workflows, which are generally based on data-flows, offer some control-flow patterns, and vice-versa.

One should not confuse *Business Process Modeling*, *Business Process Management* and *Workflow Management* [116]. The first relates to a process of taking a real-world process (e. g., a process of evaluating a loan application) and turning it into a formal workflow in unambiguous form, presumably graphical. Business Process Management is a more holistic discipline that includes modeling, but also all aspects of how workflows should be executed, analyzed and improved.

Neither modeling nor management of business processes are necessarily related to computer science. In fact, one can argue that the *ad hoc* use of business processes predates computers and modern computer science. The use of computing systems to manage business processes is tackled by the supportive science of *Workflow Management*. This is reflected in the business process lifecycle as presented in Figure 6.

There are important differences between data-flows (represented mainly by scientific workflows) and control-flows (represented by workflow models of business processes in BPM). First, semantics of workflow elements is normally different. In control-flows, workflow nodes represent *activities* (actions performed at a given step), whereas directed edges represent *causal relations* between these activities. Common *workflow patterns* used to express advanced functionality have been identified successfully in the literature [200]. This clearly translates to larger expressiveness, but also complicates interpretation of workflows.

2.5.2 *Purpose of workflows in Business Process Management*

The main purpose of workflows in Business Process Management is to model complex processes. The two main communities interested in them are represented by business administration (to model interactions between people and administrative processes) and computer science community (to model interactions in complex and distributed computer systems).

In the former case, one is interested in turning an often informal process to a model that can be managed in a strict and formal way. In particular, such a process can be managed with a help of computers with all advantages they provide. Such an approach has many features: easy access to data, integration with other computerized systems, ability to store and query historic data, notifications, formal verification of process consistency, to name just a few. Having a formal and approachable representation can also lead to useful insights, such as identification of process bottlenecks, deficiencies and optimization opportunities.

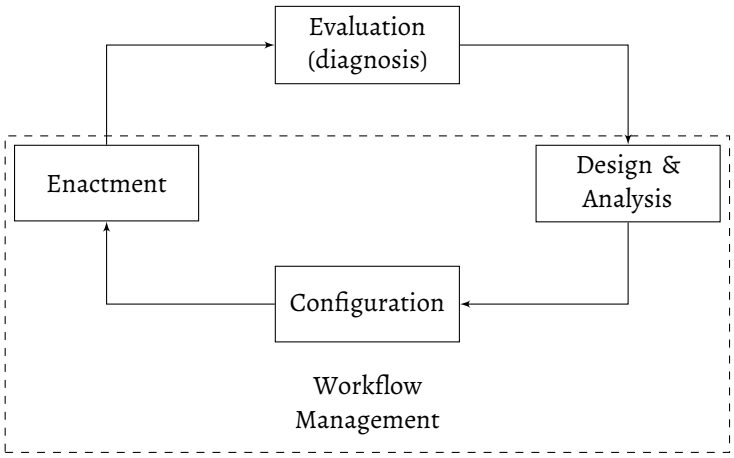


Figure 6: Business process management lifecycle. Four principal steps are present in Business Process Management, of which only 3 are encompassed by Workflow Management (Design, Configuration and Enactment).

In computer science, on the other hand, the main use of business processes and their workflow models is to orchestrate interaction of distributed services, usually *Web services*. The purpose is similar to the traditional use of workflows, but with focus moved from human interactions to more technical aspects such as compatibility of interfaces, transactional behavior, handling failures in communication without losing consistency, etc. The *de facto* standard for workflow description and orchestration in this domain is the Web Services Business Process Execution Language (WS-BPEL, less formally BPEL) [142]. It has a broad support in many software systems and from companies such as IBM or Microsoft.

Business processes have been also used to implement a fault-tolerant approach to management of distributed applications [135, 187].

Perpetual development and evolution of business processes can be summarized as *business process lifecycle* [190, 200]. Although different authors use different names for the same steps, one can define a cycle consisting of 4 steps: *Design & Analysis*, *Configuration*, *Enactment* and *Evaluation* (sometimes *Diagnosis*). The first step consists of turning a real-world process into a workflow model and validating its abstract correctness. Configuration consists of implementation and tests, whereas enactment leads to a working, monitored and curated system. Finally, there is evaluation which consists in drawing conclusions from system's runtime which may in turn lead to another iteration of the cycle.

Surveys show that the principal use of workflows is for process automation (33 %), simulation (27 %), document analysis (22 %) and process improvement (18 %) [2].

2.5.3 *Existing BPM systems*

A BPM system consists primarily of an engine that executes workflow models of business processes, however most platforms include also a set of tools to help with other aspects of a BPM lifecycle.

Most solutions include a graphical interface to build workflows from basic patterns. Workflows can be validated to make sure that they respect basic semantics of workflow models. A finished workflow can be executed with an execution engine which controls all important aspects.

Most tools offer an interface (usually Web-based) to view details of the process and possibility to interact with it. For example, the current state of the process can be monitored, its execution suspended or canceled. Most tools offer ways to generate user interfaces that implement workflow steps that require human interaction. Similarly, non-technical business users may have access to a global view of all existing processes via reports and monitoring.

The portability of workflows is provided by Business Process Model and Notation (BPMN) standard [140]. The standards defines a portable and understandable notation for all business users, from non-technical people modeling processes to technical developers implementing these processes. Nevertheless, the current version of BPMN (2.0) describes also execution semantics of workflows.

Examples of BPM software include Activiti[25], Bonita[26], jBPM[27] and ProcessMaker[28]. Most of them are based on Java, arguably due to existing community, mature technologies and tools, and its widespread use in implementation of Web Services and transactional systems. Underlying representation of data objects is often serialized as XML (*eXtensible Markup Language*) data, a standard which is traditionally used in communication of Web Services. Moreover, BPMN is also based on XML.

2.5.4 *Analysis of business processes*

The analysis of business processes modeled as workflows may lead to general characterizations of them (e. g., to identify common properties and patterns), but also serve as a help to steer their development [10]. Historical data and previous versions of the same workflows may be used to give informed, intelligent proposals as to how modify a workflow, who should be assigned to newly added tasks, etc.

Researchers have analyzed existing business processes and observed that there is a set of *workflow patterns* that frequently appear in informal real-life processes [189]. Among 20 defined patterns are such basics as *sequence of intermediate actions*, but also advanced patterns such as *arbitrary cycles* are listed. This seminal work on workflow patterns became a *de facto* standard on how to

25 http://activiti.org
26 http://www.bonitasoft.com
27 http://www.jbpm.org/
28 http://www.processmaker.com

evaluate expressiveness and completeness of a workflow engine. Workflow patterns can be used to build workflows of arbitrary complexity. The detailed presentation of these patterns is postponed to Section 4.3.1.3.

Process mining [3] is another technique that consists in reconstructing control-flow from unstructured, linear logs. It has the potential of turning an informal set of activities into structured models, presumably based on workflows. It is especially interesting in our case of scientific experiments where *ad hoc* approaches are prevalent and the incentive of using structured approaches is not always justified.

2.5.5 *Business processes in computer science*

Models of business processes are used to orchestrate complex interactions between distributed services, usually in the context of Service-Oriented Architecture (SOA) and similar paradigms. The purpose of business processes is to interact multiple, loosely-coupled and administratively separated services to obtain a common goal. Despite this successful application, business processes and workflow technologies have not been used yet for the purpose of running and controlling experiments in computer science.

First, their use is traditionally focused on modeling complex, real-life processes involving documents, products, institutions and people interacting with each other. The existing methodologies and software solutions focus on human-process interactions, which is not useful in our case.

Second, business workflows are general tools and lack domain-specific tools and workflow patterns. Although their expressiveness allows modeling of computer processes with additional work and the wrapping of services, it is potentially cumbersome, and requires effort and training. Put differently, business process management systems do not easily interoperate with the ecosystem of experimental computer science: tools, platforms, usage patterns, methodologies and culture. In particular, they are not *a priori* suited for performance measurement and monitoring, timely execution of commands on multiple nodes, or data distribution and collection.

What is more, business process management systems are often large software packages that require complex installation, maintenance and the range of supporting services (e. g., databases, schedulers, etc.). This puts unnecessary burden on the experimenters (or even forbids them from using them) and threatens reproducibility. Additionally, many such systems are proprietary and hence threaten reproducibility even more.

In the light of low to non-existing use of workflows in experimental computer science, our work is pioneering by trying to use them to control the execution of complex experiments. Many of the mentioned shortcomings will be addressed in Chapter 4.

If you can not measure it, you can not improve it.

— William Thomson, 1st Baron Kelvin

3 SURVEY OF EXPERIMENT MANAGEMENT TOOLS

3.1 INTRODUCTION

The previous chapter made it clear that the field of large-scale distributed systems is a difficult area of experimental research for many reasons. The studied systems are complex, often non-deterministic and unreliable, software is plagued with bugs, whereas the experiment workflows are unclear and hard to reproduce. Scientists are confronted with low-level tasks that they are not familiar with, making the validation of current and next generation of distributed systems a complex task.

In order to lower the burden of running an experiment and to empower scientists with new capabilities, *experimental testbeds* have been designed and built, along with experiment management tools that manage the execution of experiments on such testbeds. The last decade has seen more interest in the latter, mainly influenced by the needs of particular testbeds and other problems found in the process of experimentation such as reproducibility, replicability (see Section 2.1.4), automation, ease of execution and scalability. Despite much research in the domain of distributed systems experiment management, the current fragmentation of efforts asks for a general analysis. In this chapter, we therefore propose to build a framework to uncover the common and missing functionality of these tools, to enable meaningful comparisons between them and to give informed recommendations for future improvements and research.

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

The rest of this chapter is structured as follows. In Section 3.2, the motivation for developing experiment management tools is presented. Then, in Section 3.3, a set of features offered by existing experimentation tools is constructed and each element is carefully and precisely explained. In Section 3.4, we present the list of experiment management tools helping with research in distributed systems, where each tool is shortly presented and its features explained. The final analysis is done in Section 3.5, followed by a summary in Section 3.6.

3.2 MOTIVATIONS FOR EXPERIMENTATION TOOLS

In this section, we introduce a few definitions, among them the definition of *experiment management tool*, and explain motivations behind their existence. This serves primarily as a way to define more precisely what we intuitively consider to be an experiment management tool.

For our purposes, an *experiment* is a set of actions carried out to test (confirm, falsify) a particular hypothesis. An *experiment management tool* (for research in distributed systems) is a piece of software that helps with the following main steps during the process of experimenting:

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

Note that in Section 2.1.3.1, a model was given that resembles the four elements mentioned above. The two differences are: the design of experiment is not the part of experiment execution (and hence is not mentioned in Section 2.1.3.1), and we do not consider the cleanup phase here.

In this study we narrow the object of study even more by considering only *general-purpose* experiment management tools (i. e., tools that can express arbitrary experimental processes) and only ones that experiment with real applications (i. e., *in-situ* and *emulation* methodologies). The former restriction excludes many tools with predefined experimental workflows whereas the latter excludes, among others, simulators (see Section 2.1.3).

Finally, the experiment management tools may have a main motivation, which often is one of the following:

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

The following sections cover these principal motivations.

3.2.1 *Ease of experimenting*

The first motivation, and the main one, for creating experimentation tools is helping with the scientific process of experimenting and making the experimenter more productive.

By providing well designed tools that abstract and outsource tedious yet already solved tasks, the development cycle can be shortened, while becoming more rigorous and targeted. Moreover, it may become more productive as the scientist may obtain additional insights and feedback that would not be available otherwise.

The ease of experimenting can indirectly help to solve the problem of research of questionable quality in the following sense. As the scientific community puts pressure on scientists to publish more and more, they are often forced to publish results of dubious quality. If they can forget about time-consuming, low-level details of an experiment and focus on the scientific question to answer, hopefully they could spend more time testing and improving their results.

3.2.2 *Exploring the parameter space*

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

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

- number of nodes,
- network topology,
- hardware configuration (processors, network bandwidth, disks, etc.),
- workload during the experiment.

Of course there are countless other factors that influence experiments and that we cannot easily account for. Being able to ignore irrelevant ones is one of the things that make the *in-situ* experiments so difficult.

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

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

3.2.3 *Scalability*

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

The most important properties concerning scalability are:

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

These metrics depend on the experiment parameters (see Section 3.2.2) and implementation details. Among important factors that limit scalability understood as the metrics above are:

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

The scalability of experiments is only one of the principal motivations behind the development of simulators, as has been presented in Section 2.1.3.3.

3.2.4 *Replicability and reproducibility*

Replicability which is also known as replayability deals with the act of repeating a given experiment under the very same conditions. In our context it means: same software, same external factors (e. g., workload, faults, etc.), same configuration, etc. If done correctly, it will lead to the same results as obtained before, allowing others to build on previous results and to carry out fair comparisons.

There are several factors that hamper this goal: the size of the experiment, heterogeneity and faulty behavior of testbeds, complexity of the software stack, numerous details of configuration, unrepeatable conditions, etc. It is one of the goals of experimentation tools to try to control the experiment in such a way as to be able to produce the same results under the same conditions.

As it was explained in Section 2.1.4, replicability is hard to achieve, but cannot be mistaken for reproducibility, an even more demanding property. Achieving reproducibility is much harder than replicability because we have to deal with the measurement bias that can appear even with the slightest

change in the environment. Therefore, in order to enhance the reproducibility of an experiment, the following features are required:

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

3.3 FEATURES OF EXPERIMENT MANAGEMENT TOOLS

In this section, we present the structured set of properties available in experiment management tools for distributed systems. This classification has been obtained after a literature review using the following sources:

- tools used and published by the most influential, large-scale testbeds (see Section 2.3.1),
- papers referenced by these tools and papers that cite them,
- IEEE and ACM digital libraries search with the following keywords in the abstract or title: *experiments*, *experiment*, *distributed systems*, *experimentation*, *reproducible*.

We identified 8 relevant tools for managing experiments that met our criteria of the *experiment management tool*. An extensive analysis of the papers dedicated to those tools was performed, and an extensive list of features offered by them emerged. The majority of the features give binary information: either the feature is present, or it is absent. A few features have a larger domain of values than just binary choice, *Representation* for example.

The binary score attributed to most features is a simplification, since all tools having a given feature may provide it to a different extent, possibly in a fundamentally different and incomparable way, leading to less objective results. Moreover, the classification assumes that each feature is of the same importance and therefore has the same "weight" in the final score. A more detailed evaluation of each feature could be done, but it would require an even more fine-grained classification than the existing one and somewhat arbitrary assignments of weights. Despite this, the current classification will be enough to obtain significant observations without unnecessary burden on the researchers.

These features have been split into 9 groups: *Type of Experiments*, *Description Language*, *Interoperability*, *Reproducibility*, *Fault Tolerance*, *Debugging*, *Monitoring*, *Data Management* and *Architecture*. The following sections present the classification in to groups, and descriptions of the features. The grouping is concisely summarized in Figure 7.

3.3.1 *Type of experiments*

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

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

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

3.3.2 *Description language*

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

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

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

Expressiveness (Yes / No) of the description language makes it effective in conveying thoughts and ideas, in short and succinct form. Expressiveness provides a more maintainable, clearer description. Various elements

can improve expressiveness: well-chosen abstractions and constructions, high-level structure, among others.

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

3.3.3 *Interoperability*

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

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

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

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

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

3.3.4 *Reproducibility*

This group of features concerns all methods used to help with reproducibility and repeatability as was described in Section 2.1.4.

Provenance tracking (Yes / No) is defined as a way of tracing and storing information of how scientific results have been obtained. An experimentation tool supports data provenance if it can describe the history of a given result for a particular experiment. An experimentation tool can

provide data provenance through the tracking of details at different layers of the experiment. At the low-level layer, the tool must be able to track details such as: command-line parameters, process arguments, environment variables, version of binaries, libraries and kernel modules in use, hardware used, and executed file system operations. At the high-level layer, it must track details such as: the number of nodes used, details of used machines, the timestamps of events, and the state of the platform.

The detailed presentation of methods for collection provenance is postponed to Chapter 5.

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

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

3.3.5 *Fault tolerance*

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

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

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

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

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

Verification of configuration (Yes / No) consists in having an automatic way to verify the state of an experimentation platform. Usually such a step

is performed before the main experiment to ensure that properties of the platform agree with a specification. We distinguish verification of:

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

3.3.6 *Debugging*

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

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

Logging (Yes / No) consists in collecting, storing and presenting auxiliary information emitted during experiments, such as messages about the progress. The messages are normally stored sequentially along with their timestamps making the log a one-dimensional dataset. The log can be used to debug an experiment and document its execution.

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

3.3.7 *Monitoring*

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

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

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

experiment, to detect problems with the execution or as a way to get additional insights about the experiment.

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

3.3.8 *Data management*

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

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

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

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

3.3.9 *Architecture*

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

Control structure (Centralized / Decentralized) is the type of organization that the tool uses to control the execution of experiments. The architecture of a tool is *centralized* if the control of an experiment is centralized and there exists one node that performs all principal work. Otherwise, if there are multiple nodes involved in the experiment control, then the architecture is *decentralized*.

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

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

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

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

3.4 SURVEYED SYSTEMS

The aim of this section is to present the state of the art of the existing tools for experimentation with distributed systems. We focus our attention on the tools that fulfill the criteria for being considered as an experimentation tool (for a list of tools that are not included in the analysis, see Section 3.4.10). Moreover, we include as a reference *Naive approach*, which is defined as a standard, *ad hoc* way of running experiments that relies on the standard Unix environment.

3.4.1 *Naive approach*

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

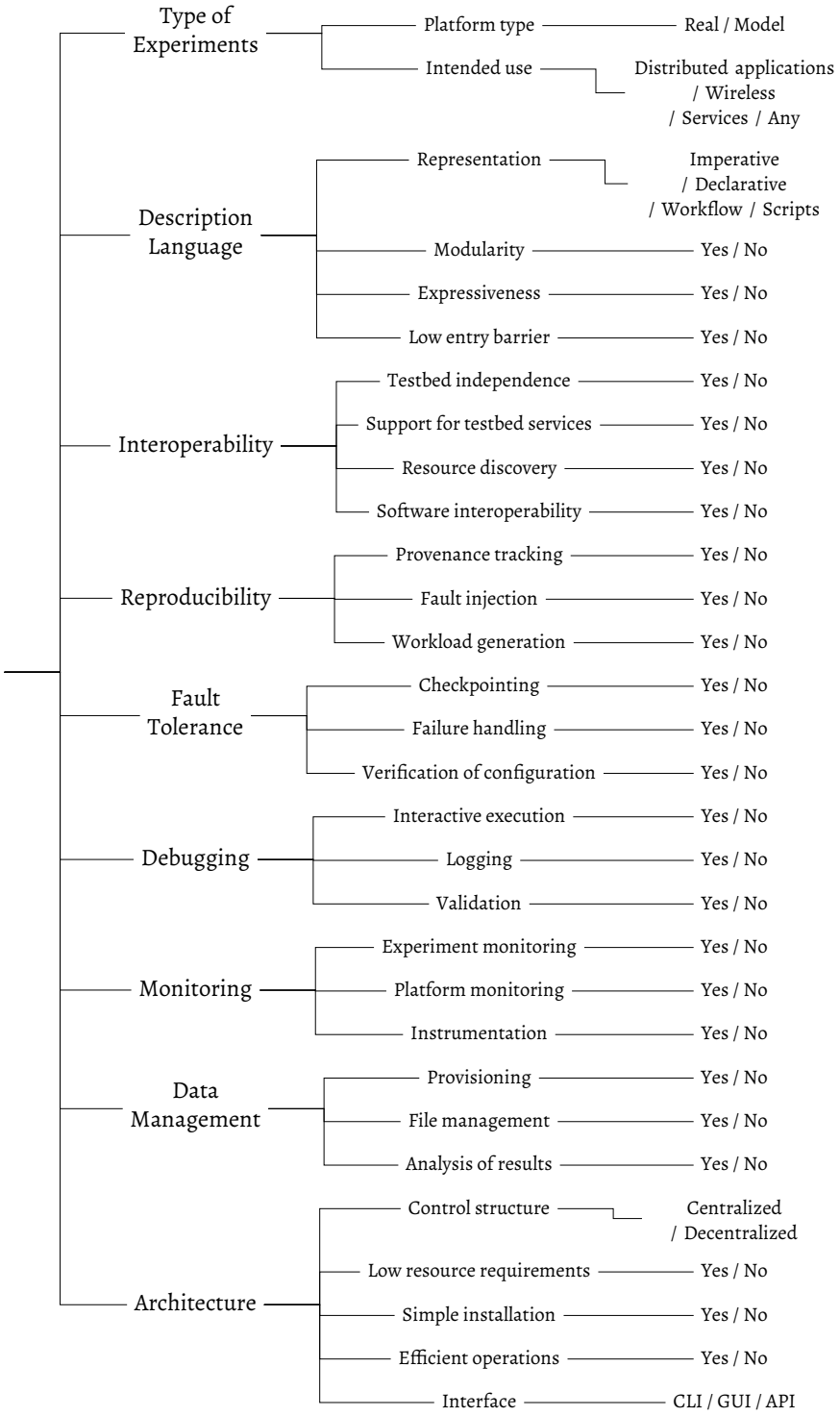


Figure 7: Tree of features identified in experiment management tools. All evaluated properties and features are presented with their respective domains of values. The properties are grouped into 9 groups that cover different aspects of experiment management.

Table 3a: Summary of experiment management tools for distributed systems research. Each feature is presented along with a number of tools that provide it. Similarly, for each group a percentage of implemented features from this group is shown. Features that are due to the integration with a testbed are marked with ⋆ (continued in Table 3b).

		Naive approach	**Weevil**	**Workbench**	**Plush/Gush**	**Expo**	**OMF**	**NEPI**	**XPFlow**	**Execo**
Type of Experiments	Platform type	Real	Real	Model	Real	Real	Real	Real, Model	Real	Real
	Intended use	Any	Services	Any	Any	Any	Wireless [1]	Any	Any	Any
Description Language (18/27 ≈ 67 %)	Representation	Scripts	Declarative [2]	Imperative [3]	Declarative [4]	Imperative [5]	Imperative [6]	Imperative [7]	Workflows [8]	Imperative [9]
	Modularity (4/9)	No	Yes	No	No	No	No	Yes	Yes	Yes
	Expressiveness (7/9)	No	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
	Low entry barrier (7/9)	Yes	No	Yes	Yes [10]	Yes	Yes	Yes	No	Yes
Interoperability (22/36 ≈ 61 %)	Testbed independence (8/9)	Yes	Yes	No	Yes [11]	Yes	Yes	Yes	Yes	Yes
	Support for testbed services (7/9)	No	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
	Resource discovery (5/9)	No	No	Yes ⋆	Yes	Yes ⋆	Yes	Yes	No	No
	Software interoperability (2/9)	No	No	No	Yes	No	Yes	No	No	No
Reproducibility (5/27 ≈ 19 %)	Provenance tracking (1/9)	No	No	Yes	No	No	No	No	No / Yes [12]	No
	Fault injection (3/9)	No	Yes	Yes ⋆	No	No	Yes ⋆	No	No	No
	Workload generation (1/9)	No	Yes	No	No	No	No	No	No	No
Fault Tolerance (13/27 ≈ 48 %)	Checkpointing (5/9)	No	Yes	Yes ⋆	No	No	No	Yes	Yes	Yes
	Failure handling (6/9)	No	Yes	No	Yes	No	Yes	Yes	Yes	Yes
	Verification of configuration (2/9)	No	No	Yes ⋆	No	No	Yes	No	No	No

[1] Supports wired resources as well
[2] GNU m4
[3] Event-based (Tcl & ns)
[4] XML
[5] Ruby
[6] Event-based (Ruby)
[7] Modular API based on Python
[8] BPM-based DSL
[9] Modular API based on Python
[10] Using GUI
[11] PlanetLab oriented
[12] See Chapter 5
⋆ Provided by testbed

Table 3b: Summary of experiment management tools for distributed systems research. Each feature is presented along with a number of tools that provide it. Similarly, for each group a percentage of implemented features from this group is shown. Features that are due to the integration with a testbed are marked with ⋆ (continued from Table 3a).

		Naive approach	**Weevil**	**Workbench**	**Plush/Gush**	**Expo**	**OMF**	**NEPI**	**XPFlow**	**Execo**
Debugging (17/27 ≈ 63 %)	Interactive execution (7/9)	Yes	No	Yes	Yes	Yes	Yes	Yes	No	Yes
	Logging (6/9)	No	No	Yes ⋆	No	Yes	Yes	Yes	Yes	Yes
	Validation (4/9)	No	Yes	Yes	No	No	No	Yes	Yes	No
Monitoring (10/27 ≈ 37 %)	Experiment monitoring (4/9)	No	No	Yes	No	No	Yes	Yes	Yes	No
	Platform monitoring (4/9)	No	No	Yes ⋆	Yes	No	Yes	Yes	No	No
	Instrumentation (2/9)	No	No	No	Yes	No	Yes	No	No	No
Data Management (13/27 ≈ 48 %)	Provisioning (5/9)	No	Yes	Yes ⋆	Yes	No	Yes	Yes	No	No
	File management (5/9)	No	Yes	Yes	Yes	No	Yes	No	No / Yes[1]	Yes
	Analysis of results (3/9)	No	No	Yes	No	No	Yes	No	Yes	No
Architecture (19/27 ≈ 70 %)	Control structure	Centralized	Centralized	Centralized	Centralized	Centralized	Decentralized	Decentralized	Centralized / Decentralized[2]	Centralized
	Low resource requirements (6/9)	Yes	Yes	No	No	Yes	No	Yes	Yes	Yes
	Simple installation (7/9)	Yes	Yes	No	Yes	Yes	No	Yes	Yes	Yes
	Efficient operations (6/9)	No	Yes	No	Yes	Yes	Yes	No	Yes	Yes
	Interface	CLI	CLI	GUI, CLI, API	CLI, GUI, API	CLI	CLI, GUI	CLI, GUI	CLI	CLI

⋆ Provided by testbed
[1] See Section 5.5.3.3
[2] See Section 5.5.1.2

Figure 8: Timeline and publications about experiment management tools. The publication that attracted most of citations (the main publication) is underlined.

and repeat. Since the experiment is run in partially manual fashion, the user can react against some unexpected behaviors seen during the experiment.

3.4.2 *Weevil*

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

3.4.3 *Emulab workbench*

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

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

3.4.4 *Plush/Gush*

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

and GENI control frameworks: PlanetLab, ORCA[1] and ProtoGENI[2]. One of the most important features of *Plush* is its capacity to manage failures. The server receives a constant stream of information from all the client machines involved in the experiment and performs corrective actions when a failure occurs.

3.4.5 *Expo*

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

3.4.6 *OMF*

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

3.4.7 *NEPI*

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

1 http://groups.geni.net/geni/wiki/ORCABEN
2 http://www.protogeni.net

3.4.8 *XPFlow*

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

The design and evaluation of XPFLOW is presented in Chapter 4 and then extended and improved in Chapter 5. The features *Provenance tracking*, *File management* and decentralized *Control structure* are not yet integrated in the mainline XPFLOW, but have a working prototype which is presented in Chapter 5. For this reason they are not taken into account in this survey.

3.4.9 *Execo*

Execo is a generic toolkit for scripting, conducting and controlling execution of actions, including experiments, in any computing platform. Execo provides different abstractions for managing local and remote processes as well as files. The engine provides functionality to track the experiment execution and offers features such as *parameter sweep* over a defined set of values. The partial results of the parameter sweep can be saved to persistent storage, therefore avoiding unnecessary reruns in case of a failure.

3.4.10 *Tools not covered in the study*

In this section, we briefly discuss other tools that could be mistaken for experiment management tools. In particular, we do not include:

- non general-purpose experiment management tools,
- scientific workflow systems,
- simulators and abstract frameworks,
- configuration and orchestration management software,
- tools capturing experimental context.

They either contradict the definition (cf. Section 3.4.10.1) or support only subset of all activities required by it (cf. Section 3.4.10.4). Nevertheless, these tools are sometimes used by experiment management tools to implement features presented in Section 3.3.

3.4.10.1 *Non general-purpose experiment management tools*

Tools like ZENTURIO [150] and Nimrod [1] help experimenters to manage the execution of parameter studies on cluster and Grid infrastructures. Both

tools cover activities like the set up of the infrastructure to use, collection and analysis of results. They are, however, restricted to parameter studies.

NXE [98] is another tool automates several steps of the experimental workflow, including reservation of resources and analysis of collected data. On the other hand, it targets network protocol evaluation.

All these tools are not general enough to be included in this study.

3.4.10.2 *Scientific workflow systems*

The scientific workflow systems were thoroughly analyzed in Section 2.4. The main objective of scientific workflow systems is to communicate analytical procedures repeatedly with minimal effort, enabling the collaboration on conducting large, data-processing, scientific experiments. Goals such as the collection of data provenance and experiment repeatability are both shared by scientific workflows and experimentation tools, however they are not experiment management tools.

There are two main reasons why scientific workflows are not covered in our study. First, scientific workflows are data-centric which restricts types of processes that can be modeled. Second, the declarative representation of many scientific workflows as acyclic graphs is generally limited in its expressiveness, therefore they do not meet the criteria of *general-purpose* experimentation tools according to our definition.

3.4.10.3 *Simulators and abstract frameworks*

Simulators have been analyzed in Section 2.1.3.3. Even though they provide many features required by the definition of the experimentation tool, they are not included in our study. First, they do not help with experiments on real platforms as they provide an abstract and modeled platform instead. Second, the goals of simulators are often very specific to a particular research subdomain and hence are not general-purpose tools.

Other tools such as Splay [119] and ProtoPeer [89] go one step further by making easy the transition between simulation and real deployment. Both tools provide a framework to write distributed applications based on the model of the target platform. They are equipped with measurement infrastructures and event injection for reproducing the dynamics of a live system.

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

3.4.10.4 *Configuration and orchestration management software*

Configuration and orchestration software has been already presented in Section 2.2.2. They simplify complex deployments by providing unambiguous, declarative description of a desired system state and then carrying out necessary steps to reach it. Operating at even higher level are orchestration man-

Table 4: Number of publications citing papers dedicated to each experimentation tool (as verified on 27 September 2015).

Tool	First publication	Citations
Weevil	2005	79
Workbench	2006	83
Plush/Gush	2006	200
Expo	2007	18
OMF	2009	203
NEPI	2010	66
XPFlow	2012	10
Execo	2013	1

agement tools, like Juju[3], which are designed to coordinate complex systems in flexible and reactive ways, usually in the cloud computing context.

All these tools do not fulfill the definition of the experiment management tool. First, they are not *general-purpose* since no precise control over the execution is available (which is actually the goal of these tools). Second, the collection of results is not present. Nevertheless, they are often used as a building block by experiment management tools.

3.4.10.5 *Tools capturing experimental context*

Tools of this kind have been presented in Section 2.2.3. Experimenters can take advantage of version control systems (e. g., Git, Subversion) or more sophisticated frameworks which aim at recording and tracking the scientific context in which a given experiment was performed.

These tools are not experiment management tools according to our definition, but they may be used to document the history of any software project, including the experiment description and results. Some tools use them as a building block to store experimental context (e. g., Emulab Workbench, Section 3.4.3).

3.5 ANALYSIS

Existing tools for experiment control were analyzed and evaluated using our set of features defined in Section 3.3 and the final results are presented in Table 3a and Table 3b. For each position in the table (i. e., each property/tool pair) we sought for an evidence to support possible values of a given property in a given tool from a perspective of a prospective user. To this end, the publications, documentation, tutorials and other on-line resources related to the

3 https://juju.ubuntu.com/

given approach were consulted. If presence of the property (or lack thereof) could be clearly shown from these observations, the final value in the table reflects this fact. However, if we could not find any mention of the feature, then the final value claims that the feature does not exist in the tool, as for all practical purposes the prospective user would not be aware of this feature, even if it existed. In ambiguous cases additional comments were provided. Much more detailed analysis that led to this concise summary is available on-line[4]. Using information collected in the table, one can easily draw a few conclusions.

There is no agreement whether a declarative description is more beneficial than an imperative one. Declarative descriptions seem to be associated with higher modularity and expressiveness, but at the price of a higher entry barrier. Moreover, the tools tend to be independent of a particular testbed, but those with tight integration offer a more complete set of features or features not present in other solutions. For example, Emulab Workbench is one of the most feature-complete tools, but many of its features are provided by the Emulab testbed.

The majority of addressed features come from *Architecture* (70 %), *Description Language* (67 %), *Debugging* (63 %) and *Interoperability* (61 %) groups. On the other hand, support for *Monitoring* is quite low (37 %), whereas support for *Reproducibility* is almost nonexistent (only 19 %). The features available in the majority of the analyzed tools are: *Testbed independence* (8/9), *Expressiveness* (7/9), *Low entry barrier* (7/9), *Support for testbed services* (7/9), *Interactive execution* (7/9), *Failure handling* (6/9), *Logging* (6/9), *Resource discovery* (5/9), *Checkpointing* (5/9), *File management* (5/9) and *Provisioning* (5/9). Moreover, the tools have nearly universally *Simple installation* (7/9), *Low resource requirements* (6/9) and offer methods to perform *Efficient operations* (6/9). The most unimplemented features are *Provenance tracking* (1/9) and *Workload generation* (1/9), both crucial for reproducibility of experiments. We will turn to the tracking of provenance in Chapter 5.

Additionally, some tools offer features that are unique to them: *Software interoperability* (Plush and OMF), *Provenance tracking* (Workbench), *Workload generation* (Weevil), *Verification of configuration* (Workbench and OMF) and *Instrumentation* (Plush and OMF).

Finally, we did a simple "impact analysis" of described tools by summing all unique scientific citations to papers about each tool using Google Scholar (see Table 4). Such a measure clearly favors tools that were created earlier, but is nevertheless an interesting indicator. The most cited tool is OMF which very recently overtook Plush (27 September 2015). As interesting as these data may be, we abstain from drawing any more conclusions from them. The summary of this analysis is available on-line[5].

4 http://xpflow.gforge.inria.fr/thesis/survey/survey.yaml
5 http://xpflow.gforge.inria.fr/thesis/survey/impact.yaml

3.6 SUMMARY

In this chapter, we identified and presented the list of properties offered by general-purpose experiment management tools for distributed systems on real platforms. The diversity of the research domain of distributed systems motivated the development of different techniques and tools to control experiments, and explains the multitude of approaches and features. With the construction of the feature list, we tried to establish a common vocabulary in order to understand and compare the existing experiment management tools. Comparison of the most important tools revealed interesting patterns.

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

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

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

The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.

— Edsger Dijkstra, *Notes on Structured Programming*

4 BPM-BASED EXPERIMENT MANAGEMENT

4.1 INTRODUCTION

In the previous chapter a comprehensive set of features offered by experiment management tools has been identified. Moreover, we observed that popular approaches have many overlapping features and that no single solution has all of them. More importantly, we found that some important features, such as *reproducibility* are universally underrepresented.

In this chapter, we introduce a novel approach to control 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 model. We illustrate our findings with XPFLOW, a workflow engine tailored to the requirements of distributed systems research. Despite its young age, this approach shows promising results, unique features and clear research directions to improve it further.

The rest of the chapter is structured as follows. In Section 4.2, we revisit the features identified in the previous chapter (see Figure 7 on page 54) and show that business process workflow models are a promising approach to implement them. Then, in Section 4.3, 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, domain-specific workflow patterns and checkpointing of workflow execution.

We also describe the building blocks of workflows (patterns) and their execution semantics. Then we present a series of three case studies that evaluate the properties of our approach and describe its real-life applications. Therefore, in Section 4.4, we evaluate our approach with a large-scale experiment. In Section 4.5, a challenging experiment with nearly 40 000 nodes is presented. The last case study is discussed in Section 4.6, which focuses

on a thorough performance evaluation of a data-distribution technique conducted with our approach. Finally, Section 4.7 concludes the chapter and outlines future research directions.

4.2 ADEQUACY OF BUSINESS PROCESS MANAGEMENT FOR EXPERIMENT MANAGEMENT

In this section, we use the framework established in Chapter 3 to make an initial evaluation of our approach. To this end, the use of business process workflow models is evaluated in the light of features presented there, and shown to have useful properties that are not present in the majority of existing approaches. As the approach relies and borrows from scientific workflows (Section 2.4) and Business Process Management (Section 2.5), it is expected to profit from features offered by both of them. This analysis highlights only the *potential* of the approach – the real-world implementation may possess only a subset of the promised features, as it is in our case.

See Table 5 for the summary.

Type of Experiments. It has been already stated before that our goal is to target real platforms (*Platform type* is *Real*) and so is the focus of our approach. The use of workflows for experiments running on the model of a platform is less justified due to higher control of the experiment execution.

Moreover, as has been shown, workflows have been used to manage various processes, ranging from human resources management, production, emergency situation management, to tasks such as orchestration of distributed services. Their successful application and use in such diverse situations is a strong indication that they are a general tool and will apply to many methods and ways of conducting experiments. Hence, there is no particular intended use for the approach (*Intended use* is *Any*).

Description Language. Our approach relies on workflow representation of experiments (*Representation* is *Workflow*). More precisely, it uses business workflow patterns as building blocks.

Workflows, being built from interchangeable patterns, are inherently modular (*Modularity*), but also offer great expressiveness due to their top-down representation and information hiding (*Expressiveness*). The primary goal and high-level tasks of a workflow can be grasped from its top-level, graphical representation.

For the same reasons, the models of business processes (as well as scientific workflows) can be used by non-specialists to perform tasks that would otherwise require technical or domain-specific knowledge. It has therefore the potential of giving the experimenters a gentle way to start using the approach (*Low entry barrier*).

Interoperability. Business processes have been used as a model to orchestrate heterogeneous, distributed services, and hence are not only capable of

interoperability, but are also the solution to provide interoperability itself between real systems (*Features in Interoperability*). In particular, the discovery of resources is available to the extent that the platform offers a service to perform such a task.

It will be shown later on that the approach indeed offers great interoperability in terms of testbed independence and software interoperability.

Reproducibility. The rigid structure of workflows has another useful property: it is quite straightforward to trace provenance of objects produced or involved in the workflow (*Provenance tracking*). Scientific workflows are the prime example, as they offer advanced and deep tracking of data provenance. However, the collection of provenance in control-flows poses some additional challenges as will be shown in Chapter 5.

Fault Tolerance. It is one of the principal uses of workflows (both in Business Process Management and scientific workflow systems) to account for exceptional situations and mitigate their effects on the proper execution (*Failure handling*).

Similarly, the rigid and well-defined structure of workflows allows efficient and, more importantly, correct checkpointing of intermediate state (*Checkpointing*). This found a successful application in scientific workflows where in some cases the previously obtained results of the workflow can be reused later hence avoiding costly recomputation.

Debugging. Workflow systems have advanced logging capabilities which automatically collect rich and structured information, as opposed to the naive logging consisting of a linear sequence of events (*Logging*).

Moreover, workflow systems often offer graphical user interfaces that enable interaction with on-going processes and real-time monitoring of progress (*Interactive execution*).

Finally, the correctness of workflows can be often performed statically due to a well-defined model (*Validation*). For example, some scientific workflow systems use meta-information about the types of data exchanged between computational steps to establish basic correctness of the construction.

Monitoring. Workflow systems have been traditionally well suited to monitor the progress of executed workflows and processes (*Experiment monitoring*). It is indeed the very purpose of workflows to present the detailed view of modeled processes. Some applications of workflows in Business Process Management rely precisely on this information to make intelligent decisions based on historical data (see Section 2.5.4).

Data Management. Workflow systems rarely address the problem of provisioning the platform with required software. On the other hand, scientific workflows are very capable of managing and transforming data, be it in the form of files or coming from external sources (*File management*, *Analysis of results*).

Architecture. Although workflow systems are predominantly centralized systems (i. e., their execution is managed by a single process), they often interact with multiple external services and hence form a distributed system (*Control structure* is *Decentralized*).
Scientific workflows are often used with large computer installations consisting of many nodes and hence require efficient and scalable implementation of collective operations such as distribution of data, discovering faults and monitoring (*Efficient operations*).
Workflow systems offer rich graphical interfaces to observe and interact with the process (*Interface* is *GUI*). This can be done thanks to the natural graphical representation of workflows.

As can be seen, the workflow-based approach covers most of the required features. In particular it offers convenient representation of experiments, high interoperability, checkpointing and provenance tracking. The last two features are particularly interesting, since they have a rather low support in the existing tools.

4.3 XPFLOW - BPM-BASED EXPERIMENT MANAGEMENT TOOL

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 afterward and stored as an experiment log.

Our implementation of this approach, XPFlow, is written in Ruby which was chosen because of its popularity in configuration management tools, such as Puppet or Chef, 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 set of different software and low-level tools are used: SSH (to control nodes), TakTuk [48] (for a scalable command execution, see Section 2.3.7) and some standard Unix programs.

In the remaining part of this section, a high-level overview of XPFlow is presented. First, in Section 4.3.1, we introduce the workflow-based repre-

Table 5: Advantages of workflow-based approach to experimentation (the summary of analysis in Section 4.2). The first group (*Type of Experiments*) is omitted.

Group of features/properties	Features offered by workflow-based approach	Remaining features
Description Language	Representation (workflow), Modularity, Expressiveness, Low entry barrier	–
Interoperability	Testbed independence, Support for testbed services, Resource discovery, Software interoperability	–
Reproducibility	Provenance tracking	Fault injection, Workload generation
Fault Tolerance	Checkpointing, Failure handling	Verification of configuration
Debugging	Interactive execution, Logging, Validation	–
Monitoring	Experiment monitoring	Platform monitoring, Instrumentation
Data Management	File management, Analysis of results	Provisioning
Architecture	Control structure (decentralized), Efficient operations, Interface (GUI)	Low resource requirements, Simple installation

sentation of experiments that is based on workflow models of business processes, but introduces some important improvements. We also make a brief introduction to the implemented control-flow and experimental patterns. In Section 4.3.2, we discuss the modular and extensible architecture that builds on the workflow representation. Section 4.3.3 discusses the handling of failures through dedicated patterns. Finally, in Section 4.3.4 we discuss the generality of our approach and the limits of its applicability.

4.3.1 *Workflow-based description of experiments*

One of main problems with published research is the lack of experimental verification of the presented ideas – substantial amount of research based on "engineering epistemology" does not devote much space to evaluation of presented ideas (only 36 % of such publications dedicate more than 20 % of space to evaluation [193]). Even if experiments are presented in a publication, they are often poorly documented using informal and ambiguous language. The lack of a formal description of experiments prevents any formal analysis or verification of correctness. Moreover, the existing practices (the use of low-level scripts and the 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 representation of workflows which maintains one-to-one relation with their original description.

In this section we discuss the domain-specific language that is used to describe workflows (Section 4.3.1.1) and how this representation manifests itself as a high-level workflow (Section 4.3.1.2). Then we present common basic workflow patterns (Section 4.3.1.3) and domain-specific, experimental patterns (Section 4.3.1.4).

4.3.1.1 *Domain-specific language representation*

In our approach, workflows are described using a 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 Figure 9 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.

```
process :setup_experiment do |clients, server|
  parallel do
    forall clients do |node|
      run :setup_client, node
    end
    sequence do
      run :setup_server, server
      run :start_server, server
      log "Server started"
    end
  end
end
```

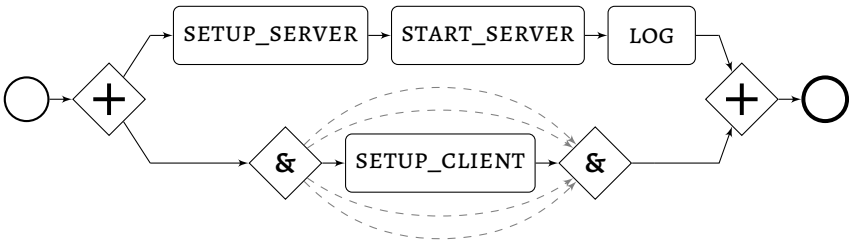


Figure 9: Example of a mapping between the DSL and a workflow. Different language-level constructs (*parallel*, *forall*, *sequence*) result in respective workflow patterns (parallel execution of subworkflows, a parallel loop and sequential execution of activities, cf. Figure 12).

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. Finally, plain text files can be compared using standard tools and meaningfully versioned with version control systems (this aspect will be explored further in Chapter 5).

The domain-specific language is primarily a tool to express the workflow structure of experiments, but it has additional functionality. In particular it has variables that are *single assignment promises*, behavior similar to parallel programming languages (e. g., Swift [204]), and functional programming languages (e. g., Haskell). They are used to store results of workflow executions and a way to pass parameters to workflows. They define data dependencies between tasks ordered using control-flow patterns, resembling the dependencies between tasks in scientific workflows.

To avoid the perils of concurrent access to shared variables, the following semantics are enforced:

- the workflow variables are *promises* (also known as *futures*) and every access to them blocks until their value is available; it is the behavior present in some parallel programming languages,

```
process :example do
  parallel do
    a = activity1()
    b = activity2(a)
    c = activity3()
  end
end
```

Figure 10: Semantics of workflow variables in XPFlow. The data dependency forces the runtime order of otherwise parallel activities. The execution of `activity2` depends on the result of `activity1`, hence it will be executed after it. On the other hand, the last activity (`activity3`) will be executed in parallel to the both remaining activities.

- the variables can be assigned only once, and later reassignments are not allowed; this behavior, often called *single assignment*, is common to the functional programming languages,
- only a single thread of execution can set a given variable.

Such a behavior completely avoids the problem of concurrent access. Moreover, the variables can be used as a synchronization primitive. An example illustrating this behavior is presented in Figure 10.

An alternative way to communicate state between workflow steps is to use *workitems*. Workitem is a global state that is the context of execution of currently executing workflow. We decided against this solution since it suffers from the aforementioned problems with access synchronization. We nevertheless consider this feature an implementation detail, and hence it will not be discusses further.

4.3.1.2 *Workflow representation*

An 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 [189]. Workflows can be built from user-defined activities or from activities provided by the core library that includes node management, data collection and statistics, logging, etc.

A workflow has the structure of a graph that is easy to visualize or transform. 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, to ensure that all referenced activities exist, for example. 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

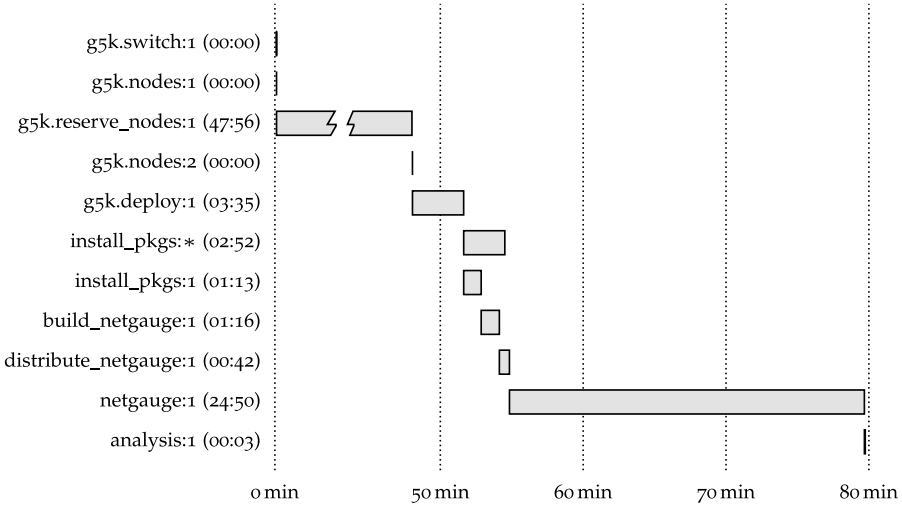


Figure 11: Gantt chart of an experiment (edited for readability). Various activities are executed, some of them in parallel: `install_pkgs:*` represents multiple parallel activities `install_pkgs` that execute on the nodes of the experiment (by means of `forall` experimental pattern).

experiment. The timing information about activities can be presented in a graphical form as well, for example as a Gantt chart (see Figure 11).

4.3.1.3 *Control-flow patterns*

In this section we present workflow patterns used to model experiments in our approach. This includes most of the standard patterns identified in the literature on control-flow patterns. We also discuss which patterns are not available and the reasons behind it.

Traditionally, the traditional workflow patterns can be grouped into: *basic control-flow patterns*, *advanced branching and synchronization patterns*, *structural patterns*, *patterns involving multiple instances*, *state-based patterns* and *cancellation patterns* [189]. Not all traditional patterns are implemented nor available in our approach.

More generally, the following assumptions have been made:

- The structure of workflows is *hierarchical* – the patterns are nested inside each other, but arbitrary cycles are forbidden. This implies that workflows can be represented as a tree and can be directly and arbitrarily reused within other patterns. Moreover, with some minor exceptions, the nesting of workflows respects causal ordering of executions – the nested workflows start *after* the parent workflow is started, and finish *before* the parent workflow are terminated.
- Some patterns are difficult to implement correctly due to their complicated semantics, hence they are not supported. In particular, we disallow most of the cancellation patterns (i. e., patterns that stop execu-

tion of other patterns) since they pose non-trivial synchronization and consistency difficulties.

It must be noted, however, that these assumptions do not limit the expressiveness of workflows modeled in such a way.

In what follows, we present the list of patterns that are essential in the approach. Each pattern is briefly presented and its semantics under normal and anomalous execution explained. The graphical representations of the patterns are given in Figure 12.

The following basic patterns are supported:

Activity is a pattern that declares a named action that can be executed during workflow execution. It is often a low-level action such as an execution of a command on a remote node. In our implementation, the language used to implement activities defaults to Ruby. The XPFLOW runtime has a number of predefined activities that can be used in any workflow. Activities execute successfully when the underlying Ruby code finishes without a problem.

Activities are declared using the `activity` keyword.

Sequence executes a list of other activities in a strict, sequential order. Each activity in the sequence is activated only if the previous one has successfully finished.

The sequence executes with a success if, and only if, all its activities executed successfully.

Sequence is represented with `sequence` in the DSL, but can be also implied is some other patterns, in particular the *Process* pattern described below.

Process is a pattern that executes according to the semantics of the *sequence* pattern. Contrary to a sequence, however, it has a name associated with it, which is used to reference it in workflows and execute it, possibly multiple times.

Processes execute under the semantics of the *sequence* pattern.

Processes are declared using the `process` keyword.

Parallel split (also known as *fork and join*) executes a list of activities in parallel, without predefined order between them, and waits for the termination of all of them.

The execution of parallel split is successful if, and only if, all parallel activities executed successfully. If one of the parallel activities failed, the parallel split awaits for all executions nevertheless, and then fails.

Parallel split is available as `parallel` in the DSL.

Exclusive choice executes one of possible activities according to the result of a comparison. It can be considered a workflow counterpart of the classical branching in programming languages.

The execution of exclusive choice succeeds if, and only if, the chosen activity executed successfully.

The DSL offers two patterns that implement *exclusive choice*: `on` for simple, binary branching, and `switch` for handling multiple choices.

Among branching and synchronization patterns, the following are implemented:

Multi-choice executes possibly multiple activities activated when associated conditions are met. It is a parallel version of *exclusive choice* pattern presented above. Multiple branches are executed with semantics equivalent to the *parallel* pattern.
Similarly to the *parallel* pattern, the execution is successful when all parallel branches are executed without a failure.
The DSL implements this pattern as `multi`.

Discriminator executes multiple parallel activities and waits till the predefined number (say N) of them finishes successfully. When this number is reached, the execution of discriminator finishes and the control is handed over to the next activity. Some of the remaining parallel activities may still execute.
The discriminator will execute successfully if, and only if, at least N of its activities executed successfully. Note that it is not necessary for *all* activities to succeed.
The DSL pattern implementing the *discriminator* pattern is `any`.

The following structural pattern is available too:

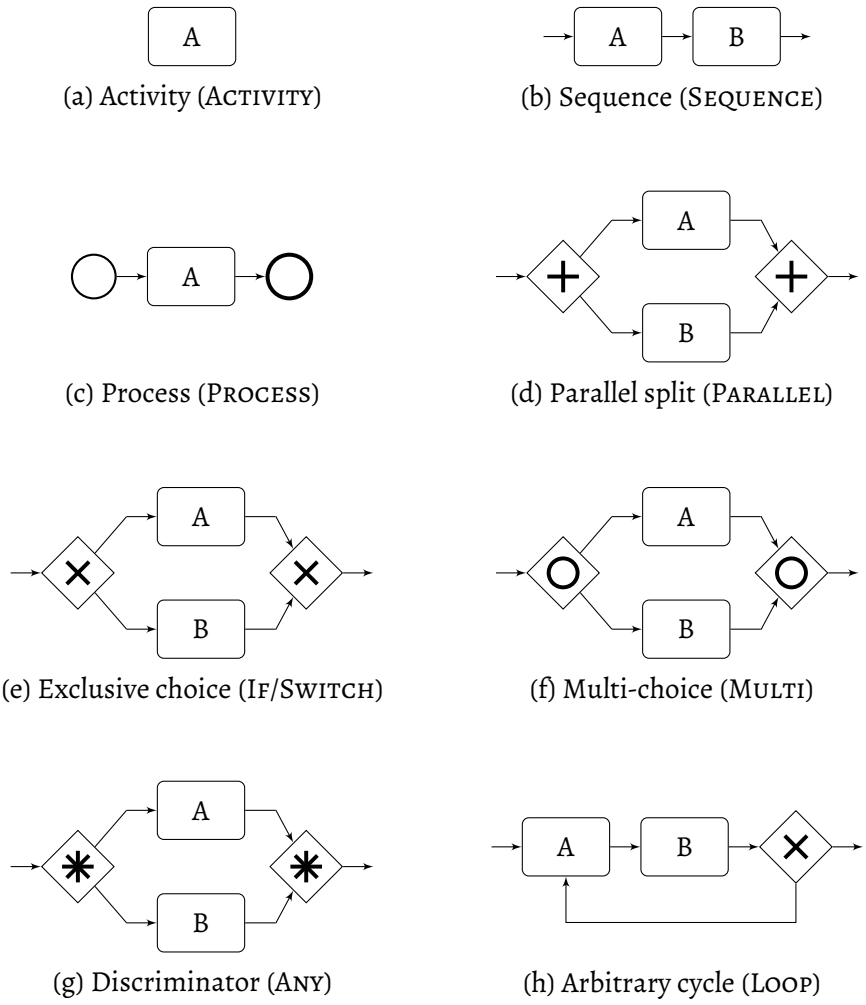
Loop executes a sequence of activities repeatedly and sequentially until a condition is met (similarly to the *while* loop from common programming languages). At each iteration, the sequence of activities is executed according to the semantics of the *sequence* pattern.
This pattern is executed successfully if, and only if, all its iterations are executed successfully. The execution fails immediately when one of the iterations fail.
The DSL construct implementing the *loop* pattern is `loop`.

4.3.1.4 *Experimental patterns*

In this section, we present the remaining patterns implementing operations that are particularly useful when modeling experiments. The presentation follows the same structure as the previous section: the patterns are briefly presented, along with their execution semantics. Figure 13 presents the visual representation of these patterns.

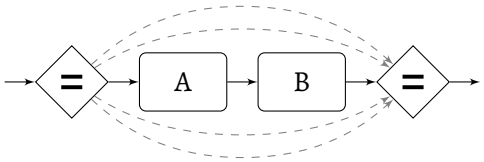
The following patterns are available:

Sequential loop of multiple instances executes multiple instances of a single activity. Each iteration is executed separately and with different parameters. It is usually used to iterate over a set of values.
The loop executes successfully if, and only if, all iterations finished successfully. It fails immediately when the currently executing iteration fails.
The DSL counterpart of the pattern is called `foreach`.

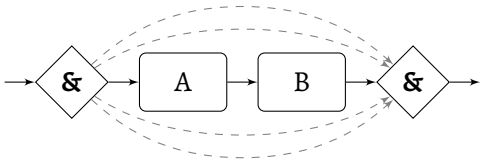


(a) Activity (ACTIVITY)
(b) Sequence (SEQUENCE)
(c) Process (PROCESS)
(d) Parallel split (PARALLEL)
(e) Exclusive choice (IF/SWITCH)
(f) Multi-choice (MULTI)
(g) Discriminator (ANY)
(h) Arbitrary cycle (LOOP)

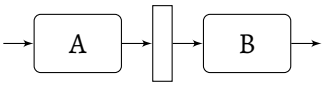
Figure 12: Overview of the most important control-flow patterns. The notation is based on, but not entirely identical to, the Business Process Model and Notation.



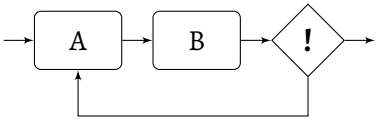
(a) Sequential loop of multiple instances (FOREACH)



(b) Parallel loop of multiple instances (FORALL)



(c) Checkpoint execution (CHECKPOINT)



(d) Retry workflow (TRY)

Figure 13: Overview of experiment-oriented patterns in XPFLOW.

Parallel loop of multiple instances executes multiple instances of a single activity in parallel. There is no order between parallel instances, and each iteration is executed with different parameters.
The loop executes with the semantics similar to the *parallel* pattern. In particular, its execution is successful if, and only if, all parallel iterations finish successfully. The failure is propagated only when all iterations are finished.
This pattern is available as `forall` in the DSL.

Checkpoint stores the state of workflow execution so that the workflow execution can be restarted from this point later. Checkpoints are only meaningful within the *sequence* pattern, and change the way the sequence is executed. When such a sequence is executed, the most recent checkpoint is restored and the execution continues as if the checkpoint pattern just finished. If there are no checkpoints available, the sequence executes normally. The proper use of checkpoints assumes certain properties of the workflow where the checkpoint pattern is used, as will be discussed in Section 4.3.3.3.
There are two complementary cases when the *checkpoint* pattern is executed: when the checkpoint is taken, and when it is restored. In both cases, the execution is successful, when the respective operations succeeded.
The DSL operation implementing the pattern is available as `checkpoint`.

Try executes an activity and re-executes it if fails (hence the executions are sequential). It finishes the execution if either: the predefined number of retries has been reached, or on of the executions finished successfully. The proper use of this pattern requires the nested activity to have certain properties as discussed in Section 4.3.3.
The execution is successful if, and only if, there is at least a single successful execution of the inner activity. Also, the execution finishes as soon as the inner activity finishes with a success.
The DSL implements this pattern via `try` keyword.

4.3.2 *Modular and extensible architecture*

The common shortcoming of existing methods and tools is the difficulty of extending their functionality or reuse them in a different context. Most of the tools support a single testbed and require a dedicated software stack to function, for example. Similarly, many approaches suffer from limited 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 that can be grouped together to form more com-

plicated workflows. Moreover, the workflows can act as parameters for the execution of other workflows giving the 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.

Two features of our approach approach are presented in this section. First the interoperability with external environments and tools is discussed (Section 4.3.2.1), and then composable and reusable experiments (Section 4.3.2.2).

4.3.2.1 *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.

The primary abstraction is that of a node which at the level of a workflow is treated just as any other variable. To interact with the node, one uses the set of activities that provide a portable and scalable implementation of common operations. These operations include remote command execution, 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 to an alternative set of activities that 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.

The scalable execution of commands on multiple nodes, as well as distribution of files, is implemented on top of TakTuk [48].

4.3.2.2 *Composable experiments*

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. An example of such a nested experiment is presented in Figure 14.

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.

```
experiment :scalability do |exp, opts, *args|
  items = items_of opts
  n = length_of items
  start = start_of opts
  step = step_of opts
  sizes = range(start, n, step)
  foreach sizes do |size|
    subset = item_range(items, size)
    result = run(exp, subset, *args)
    log "Result for #{size} is #{result}"
    value([ size, result ])
  end
end
```

Size	Performance
1	2.4
2	1.7
3	1.5

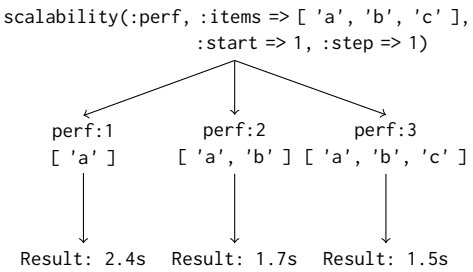


Figure 14: Example of a composable experiment that measures performance with multiple nodes (DSL representation on the left, an example how scalability experiment executes `:perf` experiment with a list of 3 items, and final results of the example on the right). A sub-experiment passed as an argument 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.

4.3.3 *Workflow execution and failure handling*

The ultimate goal of experiment execution is to finish it with success, even in the presence of intermittent 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 may fail for known reasons (such as timeouts) but also in 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 such failures.

Error handling in our approach consists of special workflow patterns to handle failures and checkpointing. However, before discussing them, a discussion about failures during experiment execution (Section 4.3.3.1) is necessary. Then we discuss the handling of failures via dedicated patterns, and properties that workflows must have to be used with them (Section 4.3.3.2). At the end, we turn to checkpointing as another fault tolerance technique (Section 4.3.3.3).

4.3.3.1 *Failure types and workflow properties*

We distinguish the two types of failures:

- *Platform failures* are caused by external technical problems, software or hardware bugs, etc., and in an idealized environment such failures would not happen. Examples of platform failures include a dropped HTTP connection due to an overloaded server or a node crashing due to software or hardware error.
- *Workflow failures* are the result of wrong structure of the experiment or wrong assumptions which, contrary to platform failures, happen even in an idealized environment. An example of a workflow failure 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 (see the execution semantics of control-flow patterns in Section 4.3.1.3). Platform failures can be further divided into *intermittent* and *fatal* ones. In practice, these two types of failures cannot be easily distinguished and the standard approach consists in using timeouts and retries to decide whether the failure is intermittent or fatal.

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.

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 prerequisites as the result of the execution. An example of a workflow that is not restartable would be one that moves a file - the file will not be present the second time the workflow is run. Such a workflow modifies its own prerequisites that are necessary for its proper execution.

A workflow is *idempotent* if a sequence of its multiple successful executions is equivalent to a single successful execution. Observe that making the 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 contents of the files a 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 if in the presence of intermittent failures it will nevertheless eventually succeed in a state that is equivalent to a single

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

Figure 15: Error handling during parallel installation of software. 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. Without the `try` block *any* failure what propagate to parent activities and, if not handled by them, would stop the experiment.

successful execution, then we will say that it is *eventually successful*. If it is also idempotent, then we will say it is *eventually idempotent*.

The aforementioned properties are crucial in a discussion explaining failure handling patterns and checkpointing. We show that by handling intermittent failures adequately, the experiment as a whole can still be executed properly.

4.3.3.2 *Failure handling patterns*

Users do not have to explicitly define what constitutes a failure as any anomalous behavior is treated as such. On the other hand, they can force an error (e. g., as a result of monitoring activity), signaling the presence of a failure. Failures can be subsequently delegated to failure handling patterns.

Any workflow can be embedded inside a failure handling pattern that will retry its execution the predefined number of times (the `try` pattern). 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. The 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.

Therefore, if the experiment is built from constituents that are eventually successful, 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 Figure 15.

4.3.3.3 *Checkpointing*

The state of the experiment execution can be saved in strategic, predefined places and restored later. The common use of that feature is to restart the execution from already reached checkpoints in the case of a failure, to save time and other resources. 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 finished experiment fails unexpectedly, the cause of the problem can be found, fixed and the experiment restarted.

It is important to note that the checkpoint contains the high-level state of the experiment execution, not the full checkpoint of the platform. The content of the checkpoint is essentially the serialized state of the workflow engine and external information that is necessary to reconnect to the testbed and nodes. This is usually enough, since full checkpointing would be prohibitively costly or require close, and often unavailable, cooperation with the testbed. The structured form of experiments represented as workflows allows taking checkpoints without much cost, and without much restrictions in practice.

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

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

4.3.4 *Applicability and limitations*

Our approach was developed as a method to run *in-situ* experiments in distributed systems. This makes it suited for this kind of experiments, but less practical in other situations.

It is arguably well suited when running experiments with grid, cluster and cloud infrastructures as will be shown in the remaining sections of this chapter. There are a few reasons behind this. First, these are the platforms that are traditionally associated with *in-situ* experiments. Second, the evolution of our approach and its implementation was driven by the author's own experience in this domain. In particular the patterns available in XPFlow are precisely those that are common while experimenting with such platforms.

It is not clear whether the approach can profit research with testbeds running emulated resources, such as wireless testbeds (such as ORBIT). Such platforms are often used by means of dedicated tools and interfaces, and consequently the use of workflow-based approach may not interoperate well with existing infrastructures.

Although most operations managing infrastructure are distributed in the current implementation, the workflow execution is not. This led and will lead to scalability problems in some extreme scenarios. Moreover, the workflow execution is synchronous and a direct reaction to asynchronous events is not possible (such a feature would greatly complicate checkpointing). As a consequence, failures are handled when their devastating effects are encountered, making the experiments that study failures in distributed systems cumbersome.

Although XPFLOW is efficient in *distributing* data, such as files, and running commands efficiently, it is not yet able to *collect* large volumes of data afterward and index it in useful way. Moreover, only limited interaction with external data sources, such as monitoring infrastructures, was implemented and tested. A general framework to collect data and provenance is one of the most direct research directions and will be discussed in Chapter 5.

Another impeding difficulty is the entry barrier associated with the approach. The researchers may refrain from changing the old habits, due to the initial complexity and not immediately obvious benefits of learning the new approach. Moreover, the current implementation lacks in terms of usability and features, which is another reason that may discourage the potential uses.

A possible way to encourage the use of our approach would be to give an automatic or nearly automatic way to rewrite existing experiments to the workflow-based approach. This is an interesting research question by itself, that is, how to find control-flow structure in *ad hoc* and manual actions of somebody experimenting with a testbed. *Process mining* solves the equivalent problem in Business Process Management, but whether the similar methods apply in our case is not directly clear.

Finally, nodes that are involved in an experiment must provide a Unix-like environment. These shortcomings are not inherent to our approach and can be addressed by hardening and extending our implementation.

4.4 CASE STUDY I - TYPICAL EXPERIMENTAL SCENARIO

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. In particular, we test such features as the *workflow representation* of experiments, the *modularity and extensibility* of our approach, and *failure handling* via dedicated workflow patterns.

This evaluation has been published as part of a conference article [30].

4.4.1 *Introduction and technical details*

Our experiment is a basic performance analysis of a single server instance of nginx[1] HTTP server (version 1.2.1) with 4 worker threads, while serving a varying number of clients. As a benchmark, we used ApacheBench[2]. A single run of the experiment consists in measuring the throughput of a single HTTP server while serving many clients simultaneously during 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 the filesystem cache) and therefore we mostly measure the scalability and reliability of the request loop implemented in the server. The operating system used to run all experiments is Debian 64-bit (wheezy) running a Linux kernel (version 3.2).

The following subsections follow the structure of Section 4.2 in the sense that each subsection evaluates one of the features announced previously.

4.4.2 *Workflow representation*

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

Moreover, the workflow can be visualized by exporting its graphical form to a file. This may in some cases provide an advantage over a plain, textual representation. Moreover, the workflows may be exported in a form that may be tuned for the inclusion in scientific articles.

The overview of the experiment is presented as a workflow presented in Figure 16. That part is independent from the underlying testbed and normally is preceded by testbed-specific part that reserves physical nodes and prepares a testbed.

4.4.3 *Modularity and extensibility*

To evaluate the modularity and extensibility of our implementation, we show the following results:

- the experiments can be built from modular and reusable components (Section 4.4.3.1),
- the experimental testbeds can be replaced while running the same, unmodified experiment on each of them (Section 4.4.3.2).

1 http://nginx.org/
2 http://httpd.apache.org/docs/2.4/programs/ab.html

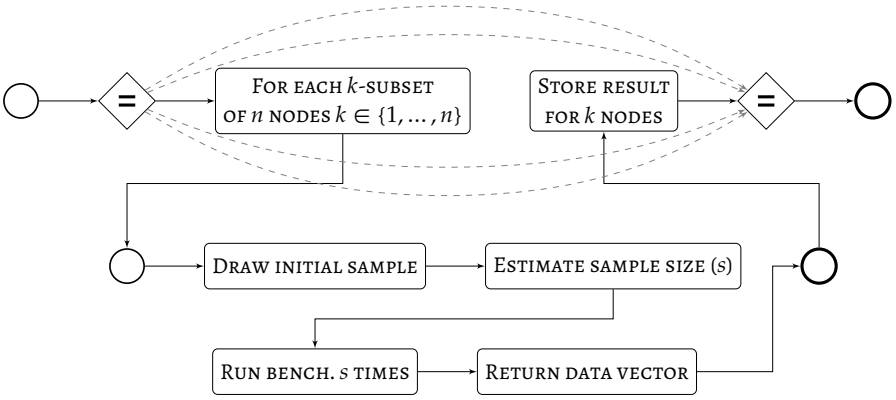


Figure 16: Principal workflow of the experiment consisting of 2 composable experiments and the main experiment (i. e., the HTTP server benchmark). The input of the experiment is a set of nodes provided by a testbed-specific workflow. One node is designated as a server, the remaining act as clients. The *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.

4.4.3.1 *Modular structure of the experiment*

Our experiment is built from reusable, modular components. One can distinguish three sub-experiments with the increasing depth of nesting:

- the scalability experiment – runs a given experiment with an increasing number of objects (in our case these objects are clients),
- the minimal-sample experiment – given an experiment 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 Figure 16.

The *minimal-sample experiment* uses the fact that results are represented as variables passed between activities. In particular, a workflow may have a looping behavior that depends on a result of partially collected results (see Figure 17 for illustration). The workflow assumes that the results returned by the nested experiment follow a normal distribution. This implies that the average of n results follows Student's t-distribution with $v = n - 1$ degrees of freedom. After an initial sample is drawn, the nested experiment is re-executed until the confidence interval derived from Student's t-distribution is smaller than a desired precision.

4.4.3.2 *Interchangeable testbeds*

The modular structure of the experiment allows replacing parts of it with ease. In particular, we show that a single experiment can be run unmodified

```
process :minimal_sample do |precision|
  first_samples = foreach(range 3) do
    run :experiment
  end
  data = data_vector(first_samples)
  loop do
    conf_prec = confidence_ratio_of data
    return_on (conf_prec <= precision)
    r = run :experiment
    data_push(data, r)
  end
  value(data)
end
```

Figure 17: Implementation of the *minimal-sample experiment* (simplified). First, 3 measures are taken and then the experiment is repeated until a desired precision is reached.

on 3 different testbeds: a container-based testbed, a real testbed consisting of physical nodes, and a virtualized one resembling the OpenStack infrastructure.

LIGHTWEIGHT CONTAINER-BASED PLATFORM

This platform was used to rapidly develop the experiment without the 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 problems early.

At the technical level, this feature is implemented using Linux Containers[3]. All instances share a common base image using the Copy-On-Write features of a Btrfs filesystem, whereas their virtual network interfaces are bridged with the host network card. The process of launching containers is implemented as a separate XPFLOW workflow and is not included in the main XPFLOW runtime.

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

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 the Intel Xeon X3440 processors, 16 GB RAM, a magnetic hard disk and Gigabit Ethernet network cards. The second one consists of machines that only significantly differ in that they have the Intel Xeon L5420 processors. We used the total number of 199 nodes during the experiment.

3 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 the 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.

CLOUD-LIKE PLATFORM

This variation of the experiment uses the 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. This experiment is the most challenging, as well as the most time-consuming due to a fairly complicated installation process. The complete listing of workflows used to run the deployment is presented in Figure 18.

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 Figure 20).

The physical infrastructure that hosts the cloud-like setup 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. The 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[4] 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 the installation of Debian Linux with necessary software. The image can be reused in different experimental scenario.

4.4.4 *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.

4 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,   #! text="Copy files"
    "files/master.sh", "/tmp/"
  execute_many master, #! text="Configure the node"
    "bash /tmp/master.sh"
end

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

activity :get_instances do |master, slaves|
  results = run :execute_many, 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,        #! text="Stop all VMs"
    "bash /tmp/flush.sh"
  execute server,        #! text="Start server VM"
    "bash /tmp/server.sh"
end

process :launch_instances do |nodes|
  execute_many nodes,     #! text="Stop all VMs"
    "bash /tmp/flush.sh"
  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 = \   #! text="Deploy Grid'5000 nodes"
    deploy_grid5000_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 =  #! text="Configure VMs"
    retrieve_instances(master, server, hosts)
  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
```

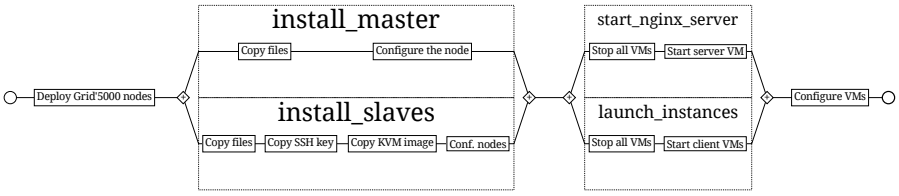


Figure 18: Full listing of workflows used to deploy a cloud-like testbed and the autogenerated graphical representation of the `deploy_cloud` workflow. Blocks defined with the `process` keyword describe workflows, whereas blocks defined with `activity` contain a low-level Ruby code. Various workflow patterns are presented: failure handling, checkpoints, parallel execution and composable experiments. See also Figure 19 which lists the most important built-in activities.

Activity	Description
`log`	store a message in the experiment log
`bootstrap_*`	initialize functionality on nodes
`execute`	run a command on a single node
`execute_many`	run a command on many nodes efficiently
`distribute`	send a file to many nodes efficiently
`test_connectivity`	check if nodes are reachable
`save_yaml`	store results in a file
`g5k_*`	interact with the Grid'5000 testbed

Figure 19: List of built-in activities in XPFlow. They are used, among others, to execute commands on nodes, distribute files and interact with the testbed (cf. Figure 18).

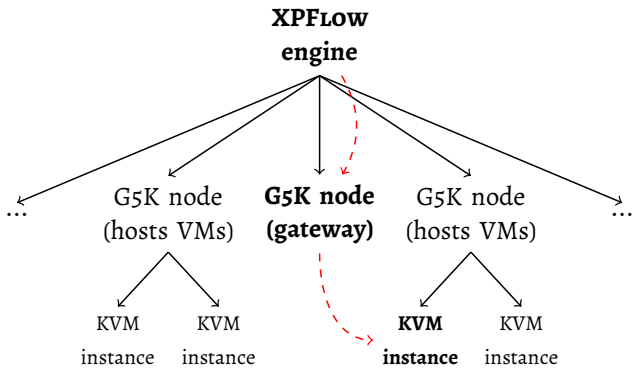


Figure 20: Hierarchical structure of nodes used in the cloud-like testbed. Three nested levels can be distinguished (XPFlow engine, Grid'5000 nodes, KVM instances). The red, dashed path describes a connection used to access and control virtual machines.

Our approach to error handling allowed us to cope with some intermittent failures that happened during the experiments. In particular, the occasional problems with running the benchmark were handled by restarting the part of workflow that failed and problems have been mitigated (see the use of the `try` workflow pattern in the `:run_nginx` workflow in Figure 18).

The experiment was structured according to the workflow properties described before. For example, the checkpoints were used in proper places to save time and to ease debugging. During the development of the experiment, they were occasionally used to fix bugs in the experiment that caused it to fail and then to restore the execution state before the failure.

4.4.5 *Conclusions*

The obtained results are presented in Figure 21. 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 physical testbed offers better performance than the virtualized environment. The peak performance reached more than 30 000 requests per second for 180 clients and would likely improve if more nodes were used.

The results obtained in the virtualized environment are not as good. The performance reached around 10 000 requests per second in our experimental scenario. The lower performance is due to the overhead of CPU and network interface virtualization. Also, we noticed that for more than 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 the cloud-like infrastructure managed with XPFLOW, we were not limited by the size of the physical infrastructure and were able to run our experiment with more than 2000 virtualized nodes.

We have illustrated a few aspects of conducting experiments represented as control-flows. First, the experiment was expressed with a domain-specific language based on common workflow patterns extended with some special patterns present in experiments in distributed systems research. The main workflow is not monolithic and instead consists of independent workflows, some of them potentially useful in a different context. Such a top-down approach promotes logical workflow structure, progressive development of an experiment, as well as reusability which is nearly non-existing in current practices.

Second, the top-down description of experiments leads to an experiment that can be executed on different platforms. We illustrated this fact by replacing the standard physical platform with a cloud-like one, or with a one hosted on a single node. This has been achieved with measurably low amount (i. e., lines of code) of changes done to the primary workflow. If developed and adopted, this can also lead to more efficient use of physical resources, as the experimenters will be able to test their experiments locally and then smoothly deploy them on a final platform.

Third, we have shown the usefulness of such features as failure handling via special workflow patterns, and checkpointing. The former assured that our experiments could continue despite occasional problems, and the latter saved significant, although unmeasured, amount of time that would be otherwise lost to manual experiment control. Ensuring that workflows used in the experiment met the requirement criteria presented in Section 4.3.3.2 and in Section 4.3.3.3 proved to be easy and straightforward in practice.

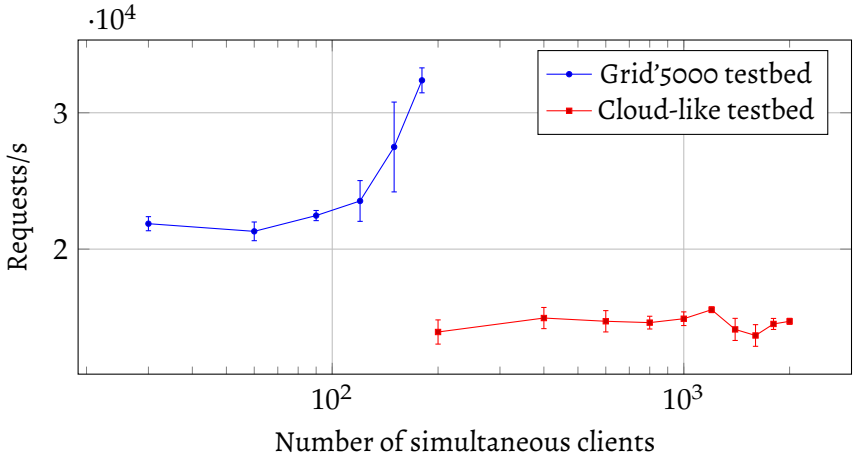


Figure 21: Performance (measured in requests per second) of the nginx HTTP server while serving a varying number of clients (X axis has a logarithmic scale). Results for two different testbeds are shown: one built using two physical clusters and one cloud-like with virtualized nodes. Each data point is presented with its 95 % confidence interval derived from Student's t-distribution.

The raw results, XPFlow software and the experimental workflows are available[5].

4.5 CASE STUDY II - EXPERIMENT WITH 40000 NODES

In this section we present a case study using our approach to conduct a large-scale experiment. This experiment consists in evaluation of a new feature implemented in the Distem system emulator. To this end, the experimental environment is set up and the evaluation of three different algorithms for the scalable execution of commands is conducted. We will briefly introduce the topic, describe the experimental setup and mention the advantages of using our approach in this case.

This section covers some details of these experiments only superficially as they are not required in our context. For the complete presentation, please see the original publication [26].

4.5.1 *Introduction*

Distem [165] is a distributed system emulator that leverages advanced Linux features like LXC, CPU frequency scaling and traffic control, to emulate a heterogeneous platform on top of a homogeneous cluster. Heterogeneity can be obtained by (1) specifying a virtual network topology where latency and bandwidth of the links can be defined, (2) emulating a degraded CPU capability, (3) executing several virtual nodes on a single physical node. Furthermore,

5 http://xpflow.gforge.inria.fr/thesis/experiments/xpflow2013.tar.xz

Distem is able to inject failures in the virtual platform in order to perform the experiment in realistic conditions.

Distem is a valuable tool for distributed system evaluation since it allows one to perform experiments in various conditions with only one physical testbed. Furthermore, Distem improves reproducibility since the same environment setup can be reproduced on a different physical infrastructure. Distem can be applied to domains such as: the study of a peer-to-peer protocol, the study of a load-balancing algorithm, the study of a scheduler, etc.

Distem scales easily to several thousands of virtual nodes (*vnodes*) running on clusters with 100 physical nodes (*pnodes*). Going beyond these numbers in terms of scale meets with various network issues. The primary reason is that the Distem *vnodes* are in the same L2 network. When running more than 5000 of them using the current architecture, several ARP-related problems appear. The ARP protocol is used to provide mapping between a network hardware address (i. e., MAC) and an IP address, a process required in Ethernet networks to enable communication between any two nodes. The encountered problems are:

- the ARP protocol does not scale very well, in particular when a lot of requests are performed at the same time, a phenomenon known as *ARP flooding*;
- the ARP tables, in particular those in the network equipment, have a hard-wired size limit (usually 8096 or 16 192 entries) or are conservatively small (in the case of default Linux settings); consequently, the addresses of *vnodes* are constantly evoked according to the LRU policy, which causes many ARP requests and excess traffic; this problem can be worked around by partitioning the address into smaller networks and setting up routing between those smaller networks;
- by default, the ARP entries are purged after a quite short time, leading to unnecessary and intrusive ARP traffic during an experiment.

Some of those issues has been fixed at the system level. Indeed, Distem performs tuning of various operating system parameters on the *pnodes* to avoid useless ARP requests. In particular, on each *pnode* Distem increases the ARP table size and the aging time of the ARP entries. Furthermore, Distem can be used to statically set a complete ARP table on all the *vnodes*, completely avoiding the ARP requests between them. These adjustments prove to be very effective, making it possible to perform experiments with 15 000 *vnodes* in the same L2 network. However, the problem of ARP tables still affects the network equipment, since in most of the cases it is not possible to modify their settings.

To go beyond this limitation, we leveraged the capabilities of overlay networks, and in particular the VXLAN [121] protocol. VXLAN encapsulates L2 network traffic in UDP datagrams, relieving the network switches since they do not have to deal anymore with *vnode* traffic directly: their ARP tables only need to store information about *pnodes*.

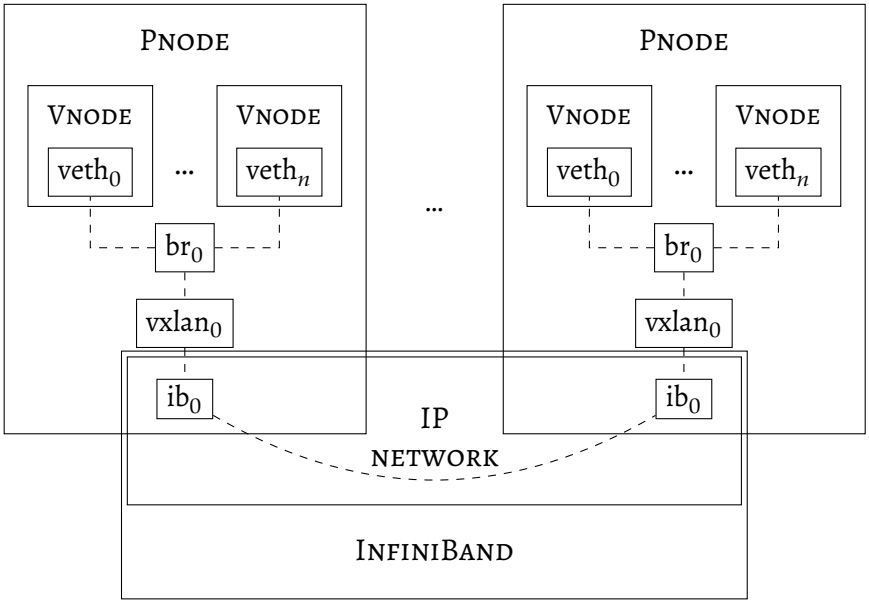


Figure 22: Network stack of a virtual node when using VXLAN encapsulation. Each virtual node has a virtual interface (veth$_i$) that is bridged with VXLAN interface (vxlan$_0$) on each *pnode*. This interface is associated with physical IP interface (ib$_0$). In our study IPoIB is used, but more traditional IP over Ethernet can be used as well.

Furthermore, to enable high bandwidth and low latency between *pnodes*, we used the InfiniBand network. This allowed us to lower the inter-*vnode* latency and to run large-scale experiments since the involved network equipment is more efficient when dealing with high-throughput traffic.

4.5.2 *Technical details*

Using VXLAN to encapsulate the inter-*pnode* traffic slightly modifies the original Distem design. To create such an overlay we leverage in-kernel Linux implementation of VXLAN supported since version 3.7.

Figure 22 presents the network stack of a *vnode* when using the VXLAN encapsulation in Distem. First, a VXLAN interface is set on top of the *pnode*'s physical interface. Because the VXLAN overlay carries aggregated traffic of all *vnodes*, a high-performance network interface offering IP interface, like 10G-Ethernet interface or IPoIB, is needed. Then, each *pnode* has a bridge that contains the VXLAN interface and the interfaces of all *vnodes* on the given *pnode*. In this case, the bridge does not contain directly the physical network interface and all the *vnodes*' traffic is encapsulated before reaching the physical network interface. There is no need to specify routes between *pnodes* since VXLAN uses multicast-based discovery to learn the routes between nodes automatically.

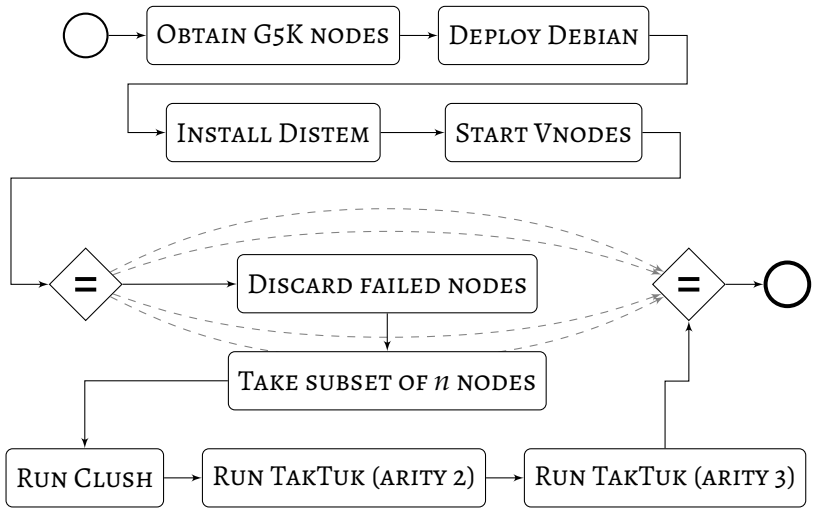


Figure 23: Workflow of the experiment in Section 4.5. After the infrastructure is prepared (the first two lines of the workflow), a varying number of virtual nodes (n) is used to run Clush and TakTuk methods in a simple loop (denoted with "="). At each iteration, the infrastructure is analyzed and failed virtual nodes are discarded if necessary.

4.5.3 *Experimental validation*

In those experiments, we want to study two parallel commands tools: TakTuk [48] and ClusterShell [183] (also known as *Clush*). Both tools aim at the efficient execution of commands on a large set of nodes, which is a critical concern since executing administrative tasks or executing complex applications on large-scale clusters may rely on such tools. As we will see, executing a simple command on tens of thousands of nodes can take a lot of time if not performed in a proper way.

ClusterShell and TakTuk differ in their way to achieve high performance. ClusterShell uses a sliding window: the root node establishes SSH connections in parallel to several nodes, bounding the number of concurrent connections. TakTuk, instead, uses a tree-based algorithm, using the already connected nodes to connect to additional nodes. The experiment presented below will compare these two strategies.

4.5.3.1 *Physical setup*

The experiment has been executed on the Grid'5000 testbed. More precisely, we used the *Graphene* and *Griffon* clusters from the Nancy site. *Graphene* is composed of 144 nodes (1 CPU Intel X3440 @2.53 GHz, 4 cores/CPU, 16 GB RAM on each node) and *Griffon* is composed of 92 nodes (2 CPU Intel L5420 @2.50GHz, 4 cores/CPU, 16 GB RAM on each node). Both clusters are interconnected with 20 Gbit InfiniBand network links and run Debian Jessie with Linux kernel at version 3.12.

On top of those clusters, we emulated a virtual platform with Distem and 162 physical nodes (*pnodes*). Each *pnode* hosted 246 *vnodes*. All inter-*pnode* traffic was encapsulated inside a VXLAN overlay. Each VXLAN interface was created on top of the physical IPoIB interface to leverage the performance of the InfiniBand network.

4.5.3.2 *Methodological setup*

The experiments where carried out with XPFlow. Among its features are robust failure handling and useful workflow patterns that model common experimental activities. In particular, XPFlow allowed us to transparently manage the failure of one physical node during our experiments, as it is explained in the following section. Figure 23 presents the experimental workflow.

Although our primary goal is to verify and show the scalability of Distem, we wanted to show it by conducting a study of various methods for command execution in large computer installations. To this end, we measured the time necessary to successfully execute the command `true` on a varying number of virtual nodes. Each measure is repeated 3 times and the results are presented with 95 % confidence intervals according to the Student's t-distribution.

The raw results, the experimental workflow and the associated files are available[6].

4.5.3.3 *Results*

The results obtained with the small number of virtual nodes are presented in Figure 24. This range was chosen to show the size of the infrastructure when the sliding-window algorithms for command execution become inferior to these based on the tree topology. It happens around 1400 virtual nodes (for TakTuk with arity 3) and around 1800 virtual nodes (for TakTuk with arity 2).

Figure 25 shows the same type of experiments, but with a much larger infrastructure. Distem, thanks to its use of the VXLAN encapsulation, shows a great scalability. Our initial goal of 40 000 virtual nodes was almost reached, as we were able to successfully run our measurements with up to 39 852 virtual nodes. The reason for that is that one physical node failed during our measurements and had to be discarded. The presence of this failure was detected and addressed by our experimental framework. Apart from that detail, we do not see why the scalability of Distem could not be pushed even more.

It can be expected that the total execution time consists of a constant factor, a linear factor (due to a constant time required by each node) and a logarithmic factor (due to a tree topology used by TakTuk). Therefore, we modeled the execution time using the model function

$$T(n) = A \cdot n + B \cdot \log(n) + C \tag{1}$$

6 http://xpflow.gforge.inria.fr/thesis/experiments/scale2014.tar.xz

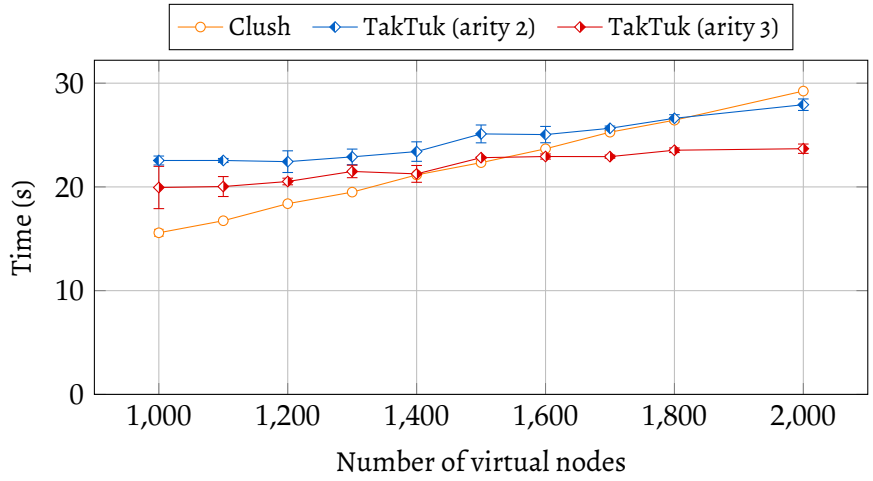


Figure 24: Time required to execute a command using various methods (small number of virtual nodes). Clush performs better than TakTuk-based methods for around 1400 virtual nodes, but is outperformed by them for 2000 virtual nodes and more.

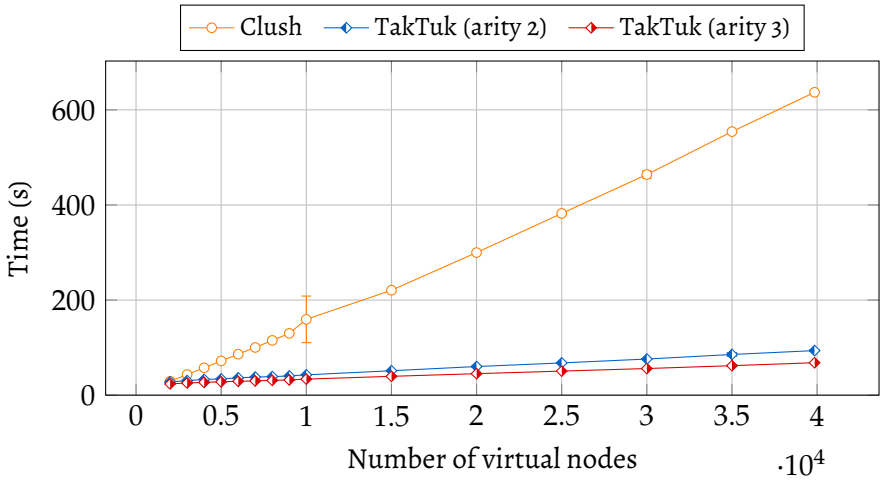


Figure 25: Time required to execute a command using various methods (large number of virtual nodes). Clush is increasingly outperformed by both variations of TakTuk which use a tree overlay to execute commands.

where T - execution time, n - number of nodes, and A, B, C - parameters of the model. The least squares fitting of the data to this model gives:

$$T_C(n) = 0.01649 \cdot n - 7.55 \cdot \log(n) + 52, \tag{2}$$

$$T_2(n) = 0.00146 \cdot n + 3.55 \cdot \log(n) - 4, \tag{3}$$

$$T_3(n) = 0.00099 \cdot n + 2.27 \cdot \log(n) + 4. \tag{4}$$

where T_C, T_2, T_3 are results for Clush, TakTuk with arity 2, and TakTuk with arity 3, respectively.

We clearly see that the linear factor of Clush has the most impact on its total execution time, whereas in the case of the TakTuk methods the linear

factor is the order of magnitude smaller and constitutes smaller percentage of execution time.

4.5.4 *Conclusions*

We showed the Distem's ability to scale to large emulated infrastructures. We were able to successfully run our experiments with 39 852 virtual nodes hosted on 162 physical nodes interconnected with InfiniBand (although one physical node failed unexpectedly), showing the scalability of VXLAN and Distem. Finally, we used this infrastructure to perform the analysis of different methods for command execution.

XPFlow proved to be a useful tool to conduct these experiments. The use of patterns for the fault-tolerant execution allowed us to continue with the experiment execution despite the problems. Notably, we were able to detect the physical node that crashed during the experiment. Although, we did not established why the node failed, our experimental workflow accounted for the problem and the experiment continued with a smaller number of nodes.

The experiment can be run unattended, with the exception of node reservation which is manual. Moreover, restarting the experiment does not require the previously obtained results and reuses the same platform if it is still available. This is achieved via the checkpointing feature of XPFlow, as well as via a data saving pattern that executes a workflow only if it has not been finished previously.

Thanks to the abstraction of the platform, the experiment control engine has the capability of running on a remote machine while communicating with the platform. This proved to be very useful, as the results were instantly available on our machine and ready to be investigated. Moreover, we could rapidly fix bugs or implement missing features locally without constantly switching between the local machine and the platform.

One of the problems found during these experiments was the number of managed nodes. XPFlow represents each node as a directory in the filesystem with around a dozen of associated files. This turned to be problematic, since with nearly 40 000 nodes, the space in temporary directory was soon exhausted. It was addressed by changing the representation of nodes in this experiment, but it calls for a more efficient representation of node states in our engine.

Another observation is that the work on the experiment proceeded in distinctively different way than we were accustomed with. The focus shifted from the low-level details such as running shell scripts by hand, copying files between machines, etc., to more high-level problems, such as how to take care of execution failures and whether they will render the results invalid. In essence, we were able to solve many problems at more generic and formal level than the common way of inspecting the low-level details of the platform or *ad hoc* scripts.

4.6 CASE STUDY III - EVALUATION OF DATA DISTRIBUTION ALGORITHMS

The last case study concerns a series of experiments that were carried out to evaluate a novel, efficient and fault-tolerant file distribution method targeting high-performance clusters. As in the previous section, we once again introduce the topic to the reader, explain the experimental setup with the focus on the workflow-inspired control of the experiments.

Our focus is the control of experiments and for that reason some details of the original study have been omitted (in particular, the evaluation of fault-tolerant features and low-level details of fault-tolerant protocol are missing). For more information, please see the original publication [123].

4.6.1 *Introduction*

Over the recent years, many areas of scientific research and industry shifted to a data-driven model, which paves the way to many changes in today's society, science and engineering. However, this important paradigm evolution changes our vision of our computing infrastructure: due to the exponential growth of amounts of information, the storage and the management of data emerged as the new bottleneck for many applications.

Several very different ways to organize data have been designed and used over the years. Traditional storage servers with large RAID array of disks continue to be used, but are often aggregated into storage clusters as part of a distributed file system such as Lustre, or more recently GlusterFS or Ceph. Those file systems provide a POSIX interface, or at least an interface that is very similar to POSIX. Switching from POSIX-compliant semantics to other logical organizations of data enables higher performance, scalability and fault-tolerance, as demonstrated by NoSQL databases such as Apache Cassandra and MongoDB, or the MapReduce programming model.

The broadcast of data from one storage system to a large number of nodes is typically used as the first operation of distributed data analysis. The same operation is also required in other contexts, such as the efficient broadcast of system images in Clouds or HPC clusters. Broadcasting has been the subject of a lot of attention both from a theoretical and from a more practical point of view.

There are algorithms that have optimal or near-optimal performance on contention-free networks. Unfortunately, most real-life networks have bottlenecks that make them behave poorly. This can be addressed to some extent by network fabrics with high performance and efficient hardware data broadcast (e. g., InfiniBand). Protocols such as UDP multicast or BitTorrent have been tried as well, but due to their verbosity, overhead and unfair behavior with respect to other network flows, they may not acceptable or efficient.

In this case study we briefly present and evaluate *Kascade*, a solution for the broadcast of data to a large number of nodes. Kascade organizes nodes

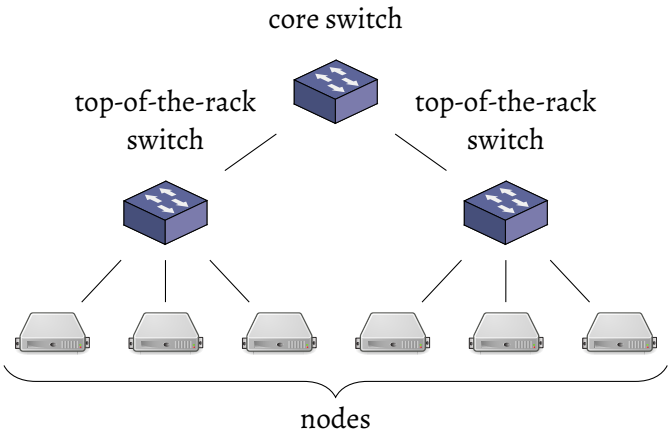


Figure 26: Fat tree network: links between the core switch and the top-of-the-rack switches are *fatter* (higher available bandwidth) than the ones between the top-of-the-rack switches and the nodes.

in a pipeline to achieve high scalability, and includes fault-tolerance mechanisms to handle the failure of nodes during the transfer.

4.6.2 *Technical details*

In this section, we present Kascade, a pipelined and fault tolerant file broadcasting tool. Kascade leverages standard network technologies (TCP/IP), is written in Ruby, and provides a friendly command-line interface.

There are three main challenges that Kascade had to address:

Local storage performance. The local storage on nodes is a performance bottleneck: mechanical drives can be written to at around 100 MB/s, whereas the fastest SSD drives provide from 500 MB/s to 600 MB/s, in both cases much lower than the 10 Gbit/s Ethernet bandwidth. Hence, it is important to start writing data as soon as possible, rather than waiting for the full data to be received. Moreover, it is advantageous to write the raw partition directly to the disk, instead of decompressing all files and paying the price of file system overhead.

Efficient use of fat tree networks. Most clusters have hierarchical, fat tree networks such as the one in Figure 26. Top-of-the-rack switches are connected to between 30 and 35 nodes using 1 Gbit Ethernet links, but the inter-switch links are only 10 Gbit. It is therefore important to organize the communication in such a way as to avoid bottlenecks.

Fault-tolerance. Failures are an important problem in the context of large-scale infrastructures. With many nodes cooperating, the chance that a single one fails is relatively high (that chance is aggravated by the fact that mechanical disks are involved). The transmission method should cope with such situations and continue despite failing nodes.

These difficulties are addressed with:

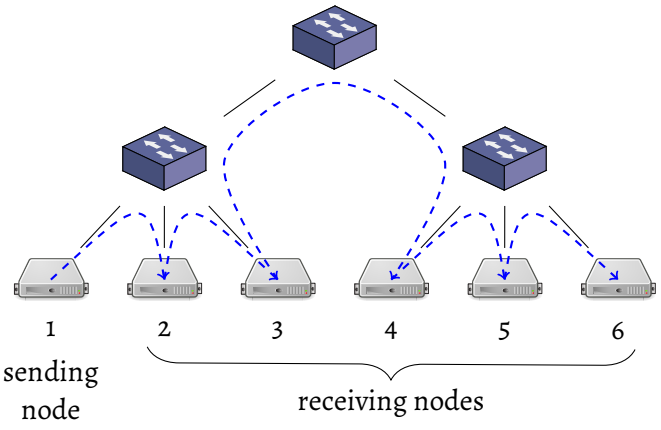


Figure 27: Topology-aware pipeline built by Kascade: node 1 (sending node) connects to node 2, which connects to node 3, etc. Node 10 connects back to node 1, which is used to forward the final report.

Topology-aware pipeline. Kascade builds a pipeline where each internal node receives data from its predecessor and passes it to the next one (and simultaneously saves it to the local storage). Moreover, the order of nodes in the pipeline reflects underlying physical topology. As a result, each participating link is used only once in each direction, avoiding congestion (see Figure 27).

Efficient start. To initiate the transfer, Kascade first copies itself to all nodes and then starts itself on each of them. The goal is to build the pipeline as soon as possible, so that the transfer may start. This step is performed using TakTuk [48] (default) or ClusterShell [183], both very efficient for the parallel distribution of small files. Although, TakTuk uses an efficient, tree-like hierarchy for distribution, we do not use it as it is susceptible to failures. Instead a windowed mode is used, which has better properties in this respect.

Fault tolerance. The Kascade protocol uses a few control messages to signal the end of transfer, aborted transfer, failures, etc. Most importantly, Kascade is able to detect and react to nodes failing in the transmission chain. If this happens, the chain is reestablished without the problematic node and the transfer can continue.

4.6.3 *Experimental validation*

In the following section we present the evaluation of the Kascade approach to data distribution and its comparison with other existing methods. In particular, we seek to answer the following questions:

- How does Kascade perform and scale up to large number of nodes?
- How does Kascade perform on high-performance networks?
- What is the impact of network topology on performance?

- What it the impact of I/O performance on performance?
- How does Kascade perform on smaller files?
- How does Kascade perform with large-latency links?

To answer these questions, we evaluated different methods for data distribution in computer networks:

Kascade is the approach described in Section 4.6.2. We evaluate the topology-aware version (*Kascade*) and a variant with a random order of nodes (*Kascade/rand*). The version used in our experiments is 0.1.5. Kascade can be found in the `addons/kascade` directory of the Kadeploy3 source distribution[7].

TakTuk is a tool for large-scale remote execution and file distribution using an efficient tree-like topology. We evaluate two different TakTuk overlays: a tree of arity 1 (i. e., a tree that degenerates into a chain, shown as *TakTuk/chain*) and of arity 2 (*TakTuk/tree*). The version used is 3.7.4.

UDPCast is a file transfer tool that can send data simultaneously to many destinations on a LAN using IP multicast. We use the default mode that uses a feedback channel. The version used is 20120424.

MPI Broadcast is an implementation of data distribution method using MPI primitives (e. g., `MPI_Bcast`). The algorithm uses a 1 MB size buffer to send consecutive fragments of a file to participating nodes. We consider two ways of execution: with Ethernet (*MPI/Eth*) and with InfiniBand (*MPI/IB*). The MPI runtime is Open MPI (version 1.4.5).

The operating system is Debian/Linux 7 with kernel version 3.2.0. The experiments were run on different clusters of the Grid'5000 infrastructure [39].

All figures show average, normalized bandwidth as a function of the number of nodes (the node initiating the transfer is not included in that number). The bandwidth is computed as the size of a file divided by the time required to transmit it to all nodes. The results are presented with 95 % confidence intervals according to Student's t-distribution.

The raw results, XPFlow distribution and the experimental workflows are available[8].

4.6.3.1 *Results*

The listing of the most important parts of the experiment is presented in Figure 28. It can be seen that the experiments in this section bear great resemblance to the evaluation done in Section 4.4. In fact, the high-level workflow is nearly identical to the one presented in Figure 16.

The results are of the experiments are presented in Figure 29. The following experiments have been conducted:

7 http://kadeploy3.gforge.inria.fr/
8 http://xpflow.gforge.inria.fr/thesis/experiments/kascade2014.tar.xz

```
use :g5k

import :common, "common.rb"

process :entry do
  job = g5k_auto_raw :site => var(:site)
  nodes = g5k_nodes(job)
  nodes = g5k_kadeploy(job, "wheezy-x64-nfs")
  checkpoint :kadeployed
  log "Nodes: #{nodes}"
  execute_many nodes, "true"
  run :prepare_nodes, nodes
  checkpoint :nodes_prepared
  run :testsuite, nodes
end

process :testsuite do |nodes|
  execute_many nodes, "rm -f /tmp/*.sh"
  distribute "files/*", nodes, "/tmp/"
  root = first_of nodes
  chain = tail_of nodes
  generate_file root, var(:size)
  log "Head is: #{root}"
  log "Chain is: #{chain}"
  results = run :"common.scalability",
                :"/single_measure", {
    :items => chain,
    :step => var(:step, :int),
    :start => var(:step, :int) }, root
  save_yaml "./results/results-#{var(:type)} \
             -#{var(:size)}.yaml", results
end
```

```
process :single_measure do |chain, root|
  generate_machinefiles(root, chain)
  run :"common.minimal_sample",
      :"/run_transfer", {
        :precision => 0.1, :initial => 3
      }, root, chain
end

process :run_transfer do |root, chain|
  log "Timeout is #{TIMEOUT}"
  log "Head: #{root}"
  log "Chain: #{chain}"
  i = try :retry => 5, :timeout => TIMEOUT do
    parallel do
      kascade_clean(root)
      kascade_clean(chain)
    end
    info do
      r = execute root,
            "/tmp/#{var(:type)}-run.sh"
    end
  end
  value(time_of i)
end
```

Figure 28: Simplified listing of workflows used by the experiments in Section 4.6. Many parts of the experiment reuse the elements introduced in Section 4.4, in particular the `:minimal_sample` and `:scalability` workflows. Moreover, a platform-specific workflow `:entry` can be seen that features integration with the Grid'5000 testbed. The workflow responsible for running the data distribution method is `:run_transfer`.

Performance and scalability. The experiment consists in sending a 2 GB file to a varying number of nodes via 1 Gbit/s Ethernet network (Figure 29a). The file is not saved to the disk.

Kascade and MPI Broadcast achieve almost the maximum utilization of the 1 Gbit/s Ethernet network and scale very well with number of participating nodes. The performance of UDPCast drops rapidly if the number of nodes exceeds 120, but outperforms the TakTuk methods which show only the one third of the theoretical bandwidth utilization.

Performance (10 Gbit/s Ethernet). The second experiment explores the performance on a cluster of 14 nodes interconnected with 10 Gbit/s Ethernet network (Figure 29b). The transmitted file has 5 GB.

No method was able to saturate the high-speed links, due to the saturation of memory bandwidth. Among them, the best one is MPI Broadcast that peaked at approximately 5 Gbit/s, but usually stays around 3 Gbit/s. It is followed by UDPCast which is able to reach slightly more than 3 Gbit/s, but usually rests just above 2 Gbit/s. Kascade shows a more stable behavior with transfer bandwidth slightly above 2 Gbit/s (likely to improve with tuned implementation). In contrast, the methods based on TakTuk show particularly low performance.

Performance (20 Gbit/s InfiniBand). In an experiment very similar to the previous one, Ethernet is replaced by IP-over-InfiniBand (with 2 switches) and the performance measured (Figure 29c).

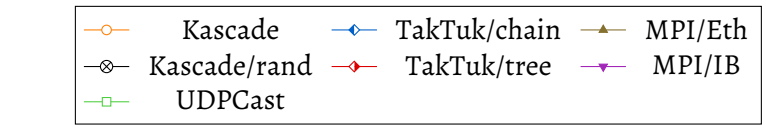
MPI Broadcast is very efficient for the small number of nodes, but does not scale very well and with 160 nodes shows a very low performance similar to TakTuk, due to saturation of the link between switches. Kascade, although having more modest performance for small number of nodes, is fairly scalable and shows a behavior similar to the experiments with 10 Gbit/s Ethernet network.

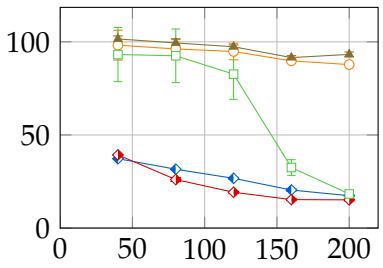
Performance (random node order). In an experiment similar to the first one, we test the impact of the network topology and force a random order of nodes during Kascade transfer (Figure 29d).
Unsurprisingly, the performance of Kascade deteriorates significantly. This phenomenon, observed also for MPI Broadcast distribution, is caused by saturation of links between switches. This does not pose a problem in our case, since the topology of the network is known to us.

Performance (writing to disk). In all previous experiments the hard disk was neither read from nor written to. In this experiment, a 2 GB file is distributed just like it was described in the first experiment, but this time the data is written to a hard disk instead of being discarded. The fact of data reaching the disk is not concerned - its presence in the file system cache is enough (Figure 29e).
The results show a much lower performance when the file is written to a disk. Among the methods, Kascade has the best performance by being able to write around 45 MB/s.

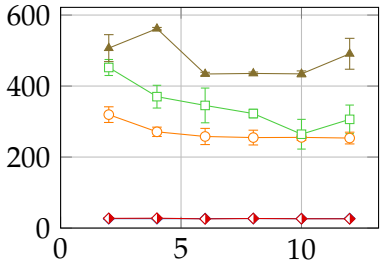
Performance (small file). To measure the overhead of protocols used by each method when transferring small files, we have run a transmission of a small file (50 MB) as in the first experiment (Figure 29f).
The transmission of small files gives a different picture than the one presented in Figure 29a. The setup time takes relatively more time and methods that have efficient start-up (i. e., MPI Broadcast and UDPCast) are clearly better. The results for Kascade or not surprising as it actually uses TakTuk to start itself on all involved nodes.

Performance (large latency). Finally, the last experiment measures the performance while using a routed, heterogeneous, long-distance network (Figure 29g). We chose 6 sites in the Grid'5000 testbed (Lille, Grenoble, Luxembourg, Lyon, Rennes and Sophia, in this order) and reserved one node on each of them. Moreover, we reserved one more node on the first site so that the first point in each plot represents intra-site distribution (see Figure 29h). Each consecutive point, therefore, represents an inter-site 10 Gbit/s link with higher latency of around 16 ms between sites (compared to 0.2 ms within a single site). The methods that cannot work with routed traffic are excluded.
All methods tend to lose performance when using high-latency and heterogeneous links. Nevertheless, Kascade offers the best overall performance. MPI Broadcast suffers from network and node heterogeneity and is even outperformed by TakTuk.

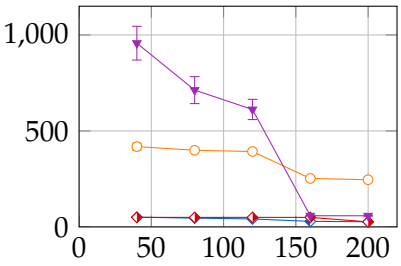




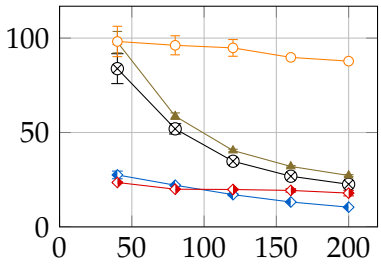
(a) Performance and scalability



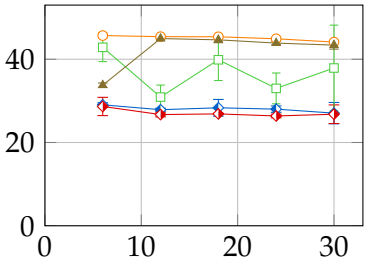
(b) Performance (10 Gbit/s Ethernet)



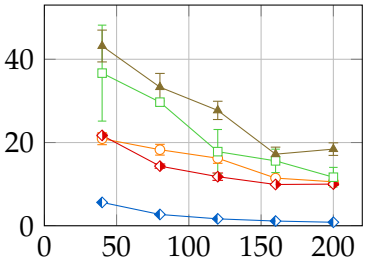
(c) Performance (20 Gbit/s InfiniBand)



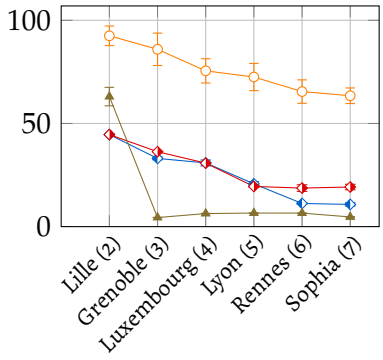
(d) Performance (random node order)



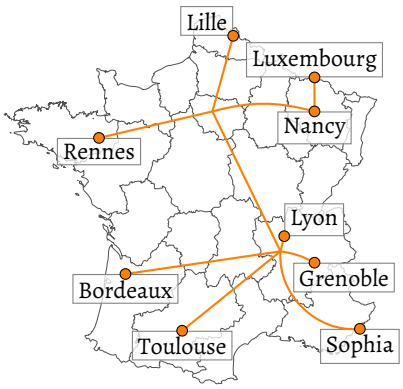
(e) Performance (writing to disk)



(f) Performance (small file)



(g) Performance (large latency)



(h) Grid'5000 topology

Figure 29: Evaluation of Kascade (see summary in Section 4.6.3.1). The vertical axis represents the obtained throughput in MB/s, whereas the vertical is the number of nodes participating in the data distribution. The last figure is the topology used in the scenario (g).

4.6.4 *Conclusions*

We presented Kascade, a solution for the large scale broadcast of data. Kascade achieves high performance and scales very well – saturating 1 Gbit/s networks even at large scale. It also provides acceptable performance on high performance networks (10 Gbit/s Ethernet, 20 Gbit/s InfiniBand) and networks with large latency. Among the evaluated solutions, it is the only one that performs adequately in all situations.

The experimental process was streamlined thanks to XPFlow. In particular, composability of experiments allowed us to replace the experimental environment, while keeping the same main experiment that consisted of running data distribution methods with a varying number of participating nodes. In fact, each experiment presented in Figure 29 shares the same structure and the same common workflow that runs the data distribution method (`:run_transfer` in Figure 28). The adaptive calculation of a sample size allowed us to minimize the time required to run the experiments, and checkpointing accelerated the development and collection of results.

We reused many workflows developed previously, in particular those that were used in the evaluation in Section 4.4 and the workflow presented in Figure 16. This again shows the modularity and reusability of components in our approach.

Some new patterns were introduced in this case study, for example `info` that collects basic information about subworkflow execution. Although very useful in practice, their inclusion in XPFlow has not been finalized.

4.7 SUMMARY

In this chapter we have presented our novel method of modeling and conducting experiments. It is based on two mature and tested approaches: Business Process Management and scientific workflows. We have shown with three case studies that this novel methodology is a promising approach to the management of experiments. It offers a range of properties that are not available in the current methods. It has also shown applicability in the context of large-scale experiments.

In particular, the modular construction of workflows makes the experiments reusable. Moreover, the experimental patterns improve the robustness of the experiments, ensuring that they finish successfully despite intermittent failures. Many features are the result of direct application of Business Process Management: implicit monitoring, the graphical representation of the process, etc.

There are many ways to improve the current approach. First, the execution of experiments is centralized on the master node (often the machine of the experimenter) which schedules tasks on remote nodes. It would profitable to be able to schedule workflows on remote nodes to achieve greater scalability.

Second, other experimental patterns could be identified and implemented in XPFlow. More generally, the capability of declaring custom patterns and

reusing them in a modular way would be a promising improvement. This would enhance the functionality and modularity of the current approach.

Third, a methodology for porting already existing experiments to the representation based on workflows could be developed. It can be done manually, of course, but the interesting challenge would be to do it automatically. One of the promising approaches coming from the Business Process Management domain, *process mining*, strives to find workflow structure from unstructured process logs. Whether this could be applied in our context, remains to be seen.

Last, with respect to reproducibility and documentation, it is useful to capture provenance of the experiment. This poses some interesting challenges for control-flows, and is addressed in the following chapter.

In solving a problem of this sort, the grand thing is to be able to reason backwards. That is a very useful accomplishment and a very easy one, but people do not practice it much.

— Sherlock Holmes, *Study in Scarlet* (by Sir Arthur Conan Doyle)

5 PROVENANCE TRACKING IN CONTROL-FLOWS

5.1 INTRODUCTION

Computers are becoming faster and more powerful, but our understanding is not advancing accordingly. On the contrary, since the systems become more and more complex one can argue that we know less and less about them. This discrepancy is disconcerting and calls for an action in almost all domains of computer science.

Experimental research in distributed systems is especially exposed to this problem. The systems under study are complex, built from similarly complex software and hardware which interact in unexpected ways. Often the scale of experiments is large and even their execution is challenging, as is the understanding of a particular system or drawing scientifically valid conclusions. A platform may suffer from intermittent or fatal failures which should not go unnoticed. The number of relevant factors and the level of complexity has long surpassed our capacity to reason about such systems as a whole.

The complexity of the systems is not the only difficulty, however. Quite often the *description* of processes (including descriptions of scientific experiments) that run on them is complex, incomplete or even erroneous. This is another menace to *reproducibility* which is generally considered a hallmark of science.

Among the techniques that may improve both understandability and reproducibility of scientific research is *provenance*. Traditionally understood as information about origins and/or a chain of custody of a historical object, it has found another meaning in computing and science as a representation of origin and transformation of a given data object during computation. In this sense it is, in fact, a form of documentation. It improves understandability and reproducibility by tracking how the processes transform data and by capturing the context where these processes take place.

However, obtaining useful provenance information is not an easy task. The problems are conceptual (e. g., what should be tracked and to which level of detail?) and technical (e. g., how to store and query provenance information efficiently?). Collection of provenance may be in conflict with performance or even correctness of the system by inadvertently changing its behavior.

This chapter makes four contributions. First, we analyze provenance collection techniques in computer science with the intention to improve their

use in experimental distributed system research. Second, building upon the previous observations, we classify provenance into three distinct but interrelated types. We then design a provenance system that follows this distinction and can provide answers to a range of queries. Finally, we evaluate our prototype of a system based on this design.

The chapter is structured as follows. In Section 5.2 we make our first contribution by making a thorough analysis of provenance in various domains. In Section 5.3, as another contribution, we propose a new classification into three types of provenance. Then, in Section 5.4, we make our third contribution by considering implications of this classification for experimental distributed systems research and sketch a design of a provenance system. This design is materialized in the prototype evaluated in Section 5.5. Finally, we draw final conclusions and describe our future work in Section 5.6.

5.2 PROVENANCE IN COMPUTER SCIENCE

In this section, we look at provenance as an object of study on its own, and then gradually narrow the domain and discuss its use and support in general computing, scientific workflows, control-flows and, finally, in experimental research in distributed systems.

5.2.1 *General provenance*

In this work, *provenance* is a collection of metadata associated with a run of a computational process that provides *any* kind of useful information as to how it was executed. This is a much broader term than *data provenance* which is a prevailing notion of provenance in computational life and earth sciences. Collection of data provenance is an active domain of research that meets much success in computational natural sciences [173].

Provenance can be *prospective* (i. e., obtained via static analysis) and *retrospective* (i. e., obtained postmortem) [55, 56]. Additionally, one may differentiate by the level of abstraction that provenance provides [14]. Four levels of provenance can be distinguished: *L0* (abstract experiment description), *L1* (service instantiation), *L2* (data instantiation) and *L3* (run-time provenance) of increasing precision and decreasing level of abstraction. Formal ways to the representation of provenance [49, 126, 131], and generic standards for interchange of provenance information have been proposed[1].

Efficient storage and querying of provenance information (which may be voluminous) is another important aspect. Using efficient representation of provenance based on its type is a standard approach [22].

A common way to construct and evaluate provenance systems consists in defining *queries* that the provenance has to answer [49]. Such a use-case driven approach is common in the domain as is shown by provenance challenges evaluating capabilities of provenance systems [132].

1 http://www.w3.org/TR/prov-overview/

Hierarchical logging, which is essential to our approach to track the provenance of experiments, is often used to provide a way of looking at series of events in a way that would be otherwise difficult with a linear representation. Recently, *systemd*[2] benefited from this approach to improve the logging of Unix services.

From this overview we conclude that there are numerous aspects of provenance and fragmented initiatives to provide it. The lack of general provenance tracking is mainly due to different requirements imposed by different domains. In the next sections, we will observe how provenance collection is addressed in different domains of computer science. To this end, we turn to provenance in general computing, scientific workflows, control-flows and in distributed systems research.

5.2.2 *Provenance in general computing*

In this section, we explore how provenance is provided in a general context (programming languages, scientific computing, data analysis, etc.). Provenance in general computing is rarely addressed, at least explicitly. First, collection of provenance always incurs overhead that may make it unfeasible to use (e. g., in high-performance computing). Moreover, each subdomain of computing calls for a different approach and therefore can be impractical and tedious to implement in each case. We will see that provenance is, with some exceptions, often addressed in *ad hoc* manner and in a very limited sense.

Some notions of provenance in database systems has been proposed [47], the most common describing relationship between a data source, a query and the results of query execution.

Software documentation is a form of prospective provenance information that explains how the software works in usually unstructured way. *Literate programming* [115] proposes to have a verbose, natural-language description interwoven with code in a single document. Similar initiatives have been proposed in the scientific context (e. g., *literate experimentation* [174]).

A useful source of provenance is provided by *instrumentation* and *monitoring*. Deep instrumentation or monitoring may be intrusive for the execution, and change the behavior of the studied system or even cause it to malfunction.

The history of how experiments evolved over time is also a form of provenance and is generally provided by version control systems (Git, Subversion, Mercurial, etc.). These systems trace the content of individual files and have no semantic knowledge about the whole system. Many systems, programming languages being a prime example, offer therefore language-specific *software archives* that host code that can be referenced (e. g., PyPI for Python or Hackage for Haskell). Studies of large, language-agnostic software repositories have been done as well, showing interesting aspects of long-term soft-

2 http://freedesktop.org/wiki/Software/systemd/

ware evolution (e. g., Debsources [36]). Some retrospective studies trace authorship of modern BSD systems, as far as 40 years into the past [176].

There are solutions that build on version control systems and aim at the automated capture of experiment context and data files for easier reproducibility of research [57]. More recently, some researchers propose workflows using the Git branching model *and* literate programming with Org-mode [178].

5.2.3 *Provenance in scientific workflows*

Scientific workflows describe the set of tasks needed to carry out a computational experiment [61]. Their role usually consists in carrying out the computation using a given infrastructure (e. g., a computational grid or virtual machines in a cloud), but without going into details about how exactly these operations are executed. Therefore scientific workflow systems provide a high-level abstraction of computing and are used even by non-technical researchers. The efficient scheduling and execution of scientific workflows is a vivid domain of research.

The standard representation of scientific workflows uses acyclic *data-flows* to describe transformations of input data in a structured way. The acyclic graph structure implies a natural way to collect data provenance by workflow systems. The data-centric nature of scientific workflows is a well-known fact: it has been observed that the *data preparation* (i. e., initial transformation of input data to useful representation) accounts for more than 50 % of workflow structure [90].

The history of how experimental workflows evolved is rarely tracked, with some exceptions. For example, VisTrails [87] enables backtracking from an unsuccessful approach by storing historical changes in a tree.

Similarly, the details of the underlying platform are also rarely collected. Since the premise of scientific workflow systems is to abstract the details of the computing platform away and still obtain qualitatively equivalent results, this is understandable. This type of provenance can be still useful, for example for debugging, but is not essential.

The examples of scientific workflow systems include Kepler, Pegasus, Taverna, and VisTrails; see [180, 208] for surveys of scientific workflow systems. The details of provenance support in scientific workflow systems have been explored thoroughly [86].

5.2.4 *Provenance in control-flows*

For the purpose of this work, we define *control-flows* as workflows consisting of a set of activities that are performed under causal, temporal and spatial constraints to achieve a specific goal, such as, in the context of this work, the collection of experimental results. This definition is closely related to the one of *business processes* in Business Process Modeling [116]. The differences between control-flows and data-flows are studied quite extensively [16, 120],

including the expressiveness of both formalisms [52]. The most important distinction is that data-flows are data-centric, contrary to control-flows.

The provenance collection in BPM and control-flows does not seem to be very much explored yet. This may be due to the mentioned difficulties and due to the proprietary nature of many BPM systems. To our knowledge, our work may be the first to explore provenance tracking in control-flows to a larger extent.

5.2.5 *Provenance in experimental distributed systems research*

Research in distributed systems developed a wide range of methods to tame the complexity associated with running complex experiments. These methods can be grouped into four methodologies: simulation, benchmarking, emulation and *in-situ* experiments [101]. The *in-situ* methodology, which we focus on in this work, consists in running a real system on a real platform, and arguably requires the most extensive provenance coverage among all methodologies.

There are various solutions that control executions of *in-situ* experiments on real platforms (e. g., Plush [8], OMF [156]). The support for the collection of provenance in these tools is almost nonexistent, as has been shown in Section 3.5.

Recording and subsequently restoring the state of platform configuration is another aspect of provenance addressed to some extent by system configuration management tools like Puppet, Chef or Salt. NixOS [69] takes a more generic approach of declarative and stateless description of a full system.

5.3 NEW CLASSIFICATION OF PROVENANCE

As has been observed above, there are many different ways to provide provenance information and methods differ between domains. In this section, we will observe that for any form of computation executed on a system like grid, cloud, or any computing platform, the general provenance can be split into 3 different types. More precisely, we will show that apart from the *provenance of data* two other types of provenance exist: the *provenance of description* and the *provenance of process*, and that all three are useful and even necessary for a complete provenance system. Although this work concentrates on the domain of experimental distributed systems research, the discussion in this section is general and applies to scientific workflow systems and even outside the scientific context.

To explain the existence of these three types of provenance, we make the following observations about *entities* that are present in an arbitrary computation on a computing platform. First, from an abstract point of view, there are two elements necessary to run it: its abstract *description* and a physical platform. The platform consists of physical *machines*, network *equipment*, installed *software* and other details. The execution of the given computation may have useful *runtime* information and may produce, receive or transform

Table 6: Summary of the three proposed provenance types and their position in existing classifications. L2 provenance is not present in most distributed research experiments.

Name	Entities	Moment of collection ([56])	Level ([14])
Data	Results, monitoring data, platform configuration	Retrospective	L3 (and L2, if present)
Description	Experiment description, platform specification	Prospective	L0 and L1
Process	Runtime information	Retrospective	L3

arbitrary *data*. All these objects can be separated into 3 different classes (see Table 6 for a summary).

Note that L2 provenance (data instantiation) is not covered, since experiments we focus on do not take raw data as an input. However, if need be, L2 provenance can be classified under the provenance of data.

PROVENANCE OF DATA is information on how data objects were created and transformed during the execution of a given computation. This is the type of provenance that is largely synonymous with provenance itself due to its successful application in scientific workflow systems. Moreover, data provenance is *implied* by the structure of data-flows, that is, its interpretation and representation is derived from the original structure of a data-flow.

According to the existing classifications (see [14, 56]), the provenance of data is of *retrospective* and *runtime* (L3) type. It may also cover L2 provenance (data instantiation), however it is not the case in our domain.

PROVENANCE OF DESCRIPTION is information on how the description of the computation evolved as a function of time and how it came to be (e. g., who authored it). This provenance type is a form of documentation, but has multiple other uses. In particular it may track dependencies of the computation, as well as authorship information, among others. We will see that in our proposed approach even more features are present (see Section 5.4.2).

This provenance is *prospective* and covers the levels L0 and L1 of provenance.

PROVENANCE OF PROCESS constitutes metadata that document details of how the execution of the computation progressed. In particular, it includes information on how it behaved in time (e. g., *when* parts of

it executed) and in space (e. g., which machines were involved). This kind of information is useful to understand the inner workings of the computation and the system, and resolve problems when they happen. Additionally, it documents the execution for reproducibility purposes. The provenance of process is implied by the control-flow structure, just like the provenance of data is implied by the structure of scientific workflows (see Section 5.4.3).

The provenance of process is *retrospective* and of *runtime* (L3) type.

One can argue, that all the mentioned supplementary types of provenance can be considered like any other data that, by definition, is tracked by the provenance of data. There are nevertheless a few reasons that warrant such a distinction.

First, the provenance of description operates at a higher level than the others. Indeed, the information it provides would not normally require the execution of the given computation, unlike the others. Second, as we will see in Section 5.4, different types of questions can be posed for each type of provenance. The presented distinction leads to more efficient storage and access, as well as more appropriate representation and visualization. Finally, provenance information is difficult to query without structured data and may be overwhelming both conceptually and in terms of resource requirements. For this reason, virtually every approach models provenance information in one way or another.

5.4 DESIGN OF A PROVENANCE SYSTEM

In Section 5.3 we introduced a new classification of provenance into three types. This section proposes a design of a system that takes as an assumption the control-flow structure of experiment description. Our decision is dictated by promising results obtained while running large-scale, challenging experiments represented as business processes (see Chapter 4). To represent experiments, the BPM workflow patterns [189] are used, extended with *experimental patterns* that include parallel execution of commands, failure handling, etc. We have shown previously that under sensible assumptions large-scale experiments can be run robustly.

First, let us explicitly state assumptions guiding our discussion and design:

- The experiment follows *in-situ* methodology in distributed systems research.
- The experiment description is a control-flow based on workflow patterns.
- The data processing does not constitute a large fraction of the experiment execution. If it is not the case then it would be reasonable to use scientific workflow systems which are more suited for this task.

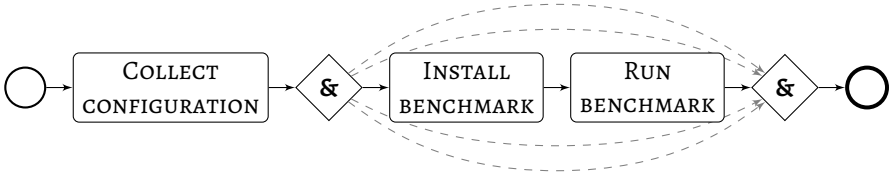


Figure 30: High-level workflow description of the exemplary experiment (modified BPMN notation). First, the configuration of the nodes is collected. Then, on each node in parallel, a benchmark is installed and run.

For such an experiment, one can ask a question about requirements of a prospective provenance system. We take the following approach: (1) we find *entities* that can be distinguished in the experiment, (2) we define *questions* that can be asked about them.

The first part has been already done in the previous section. As for the second step, we start with a general principle that for a given entity X the two following pieces of information describe fully its provenance: (1) the origin of X and (2) the logical, spatial and temporal context of X.

For the objects stored as the provenance of experiment data, the prospective questions ask which activities created the given datum (logical), which physical nodes or equipment was involved (spatial) and when that datum was collected (temporal). In the case of the provenance of experiment description this leads to questions about authorship of code (logical), about the dependencies between modules (logical) and about the changes to the experiment in time (temporal). There is no spatial context for the provenance of experiment, however, since it is only present in an abstract form. Finally, in the context of the provenance of experiment process this information reduces to details on how the experiment activities executed with respect to each other and the experiment description (logical), where they executed (spatial) and when they executed (temporal). This analysis leads to examples of questions presented in Table 7.

In the following sections, a design of a provenance system is presented. It consists of three subsystems, each capturing a single type of the defined provenance. We observe that the natural representation for the provenance of data, the provenance of description and the provenance of process is: by a directed graph, by a rooted acyclic graph, and by a hierarchical tree, respectively. We illustrate the discussion with an abstract example of a benchmark executed on multiple nodes (see Figure 30 for its workflow description).

5.4.1 *Provenance of experiment data*

As observed before, capturing and storing the provenance of data is a challenging task with many difficulties. However, due to our special use case (i. e., control-flow based experiments in distributed systems research) we were able to make a few assumptions. In particular, since the data transformations does not constitute a significant portion of experiments that we are in-

Table 7: Summary of the provenance types and examples of queries that one may ask for each of them.

Type of provenance	Examples of queries
Data	*Which node produced the highest benchmark result?* *What was the runtime system configuration of the nodes?*
Description	*What are the dependencies of the experiment?* *Are there newer versions of modules?* *Who is the author of the activity X?*
Process	*What is the Gantt diagram of the experiment?* *What is the critical path of the experiment?* *What are the failure rates of activities?*
Data & Description	*Did the system specification reflect reality?*
Data & Process	*What activities executed at the node X?*
Description & Process	*Who authored a change that caused the experiment to fail?*
All types	*Who is the author of a module that produced the result X?*

terested in, we can store the provenance of data (and data itself) in mostly unstructured way. As a result, we propose a simple key-value store, e. g., BerkeleyDB [139]. Distributed key stores may be preferred if high-availability or scalability is requested (e. g., Dynamo [60]). Although in principle data objects stored as the provenance of data may reference each other, the design does not support directly queries involving these relations. Nevertheless, in general, data provenance is an *arbitrary directed graph*, which presumably is sparse.

Each datum in the store has its *name*, *type*, *value* and *context*. The *name* is an identifier (not necessarily unique) of the given data artifact in the context of the current experiment run. Its *type* defines a group of objects it belongs to (e. g., nodes, results) and can be used to optimize queries by narrowing them down to a subset of elements in the store. The *value* of the datum is its raw value, it may be, for example, the result of a benchmark on a node. Finally, the context is used to link the data to the provenance of process, and contains information such as the node that the result comes from, or a timestamp when this data object was stored. The context allows also to link the stored data to other types of provenance.

The data stored as provenance must be explicitly marked as such. There are two reasons behind this decision: (1) contrary to data-flows, in control-

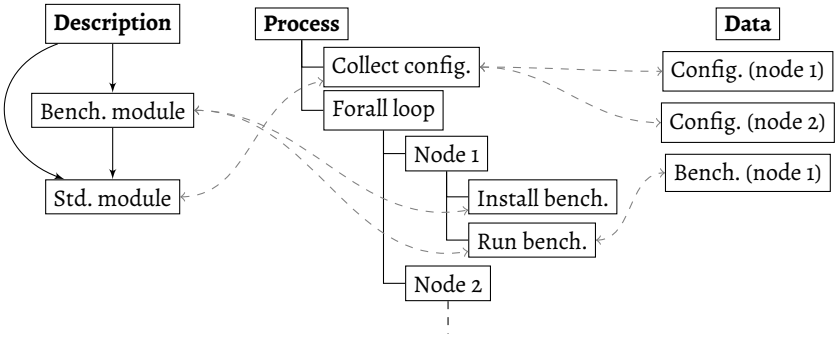


Figure 31: Relations between the three types of provenance. The central role is occupied by the process provenance. It refers two-sidedly to the description provenance and to the data provenance, with no direct links between them.

flows it is not explicitly known what *constitutes* data, (2) it narrows down the scope of what is collected and improves performance. Nevertheless, some elements of data provenance can be collected automatically, the configuration of the platform, for example.

The data provenance collected in the exemplary experiment (Figure 30) consists of benchmark results (*benchmark* type) and runtime configuration of nodes (*configuration* type). The benchmark results point to the nodes that collected them and to the instances of activities that produced them (see Figure 31).

5.4.2 *Provenance of experiment description*

In this section, we provide a design for the representation of experiments that traces the provenance of the experiment description. By *experiment description* we mean the workflow of actions executed as the experiment.

Our solution to this problem is inspired by software engineering, more precisely, by (1) a version control system (Git) to track the evolution and authorship of the description and (2) a module system based on programming languages (Go) to improve reproducibility, document dependencies and facilitate collaboration. Note that creative use of Git branching model is nothing new [57, 178].

However, as Git tracks content at the level of files, it does not meet all our needs. For that reason we propose a simple, yet powerful and easy to use module system that is built on top of it. It adds an additional layer that tracks dependencies of experiments. The approach is modular and ensures reproducibility and consistency of the experiment description, while remaining easy to use.

The experiment and its modules are tracked in a Git repository, and *tags* represent its evolution. Dependencies of the experiment follow the same versioning scheme and are referenced as a pointer to a *tag* in another *repository*. It implies that a pair (*repository*, *tag*) unambiguously defines the experiment

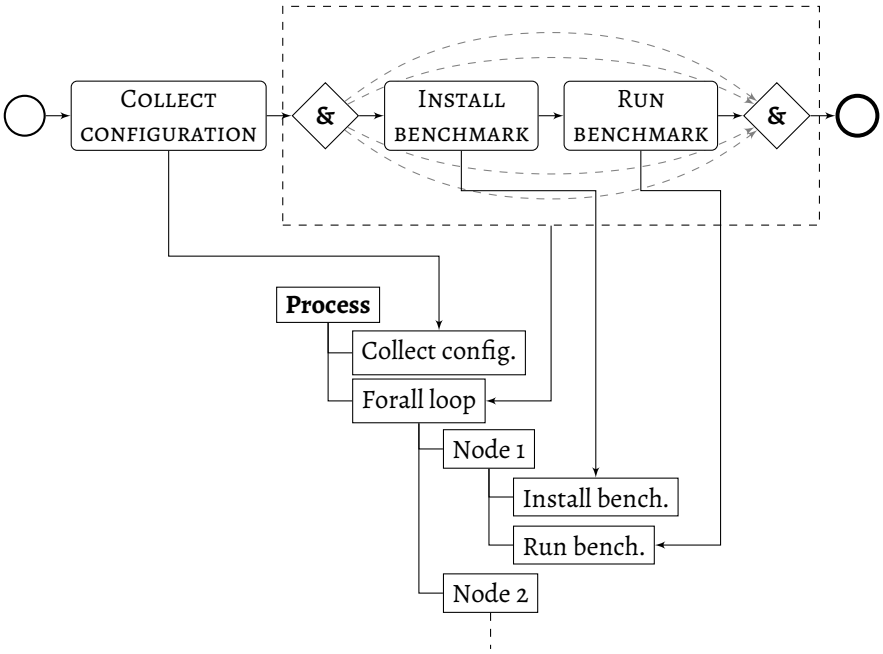


Figure 32: Example of a mapping between the control-flow and its log. The nesting of workflows implies the hierarchical structure of the log, as is shown with arrows (cf. Figure 31).

description with its all transitive dependencies. Cycles in the dependency graph are forbidden, hence the provenance of description is a *directed, acyclic graph*.

In our exemplary experiment (Figure 30), the main workflow references the benchmark module (module *Benchmark*) and the standard library module (module *Standard*). Moreover, the benchmark module depends on the standard module too (see Figure 31).

The repositories containing the workflows can be hosted at social software repositories like GitHub, which offer useful features supporting collaboration, innovation, knowledge sharing and community building [54].

5.4.3 *Provenance of experiment process*

In this section we start with the analysis of control-flows (based on workflow patterns from Business Process Modeling) and observe that their structure maps to a hierarchical log. We then use this structure as the representation of the provenance of process.

Just like scientific workflows imply the structure of data provenance, the structure of an experiment represented as a control-flow implies a form of process provenance. From a high-level point of view, a BPM-like data-flow can be defined either as an *activity* (a basic, atomic action) or as a *pattern* (e. g., a sequence of activities). Other patterns have been defined in the literature [189] and by us, in Section 4.3.

In Figure 32, we see the same workflow that was shown in Figure 30, and associated structure of the provenance of process. In particular, subworkflows map deeper into the log hierarchy. We see therefore that the natural representation for the provenance of process is a *hierarchical tree*.

Each log entry has some predefined data recorded: a timestamp when its execution started, timestamp when its execution finished and the node that executed this activity. The log entries annotate their log with relevant information, such as a pointer to the source code. The log can be stored in the key-value store used to store the provenance of data.

5.5 EVALUATION

In this section, we introduce the details of the implemented provenance system and evaluate it with a simple, but telling example. The introduction in Section 5.5.1 focuses on differences between the system presented in this chapter and the one discussed in Chapter 4. Moreover, it mentions a few technical details specific to the new prototype. Then, in Section 5.5.2, the description of an experimental scenario is given. It is a simple experiment crafted to explore the important aspects of the collection of provenance. In Section 5.5.3, a discussion of the experiment from the point of view of the provenance types is presented. Finally, in Section 5.5.4 we draw final conclusions that are drawn from this evaluation.

5.5.1 *Introduction and technical details*

The implementation is based on the general methodology presented in Chapter 4, but is a separate framework (although much of the previous code is reused). The use of workflow patterns is retained, but the model and implementation details differ. In particular:

- The provenance collection is integrated according to the high-level design presented in the previous sections.
- The execution model differs in some aspects that make the design of the original domain-specific language simpler, but not without breaking backwards compatibility. However, this change also enables features that were impossible to attain before. It also integrates much better with the proposed provenance model.
- Some features present in original XPFLOW are not yet present in the new implementation (e. g., scalable command execution, testbed integration, many workflow patterns). These shortcomings are due to the current focus on the aspect of provenance collection. There is nothing inherently conflicting between the new model and previously implemented features.
- On the other hand, we implemented some new features that were not available in original XPFLOW. Prominently, decentralized experiment execution has been implemented in the new framework.

```
process :main do
    sequence :a => "a1" do      # context is { :a => "a1" }
        show_vars :b => "b1"    # context is { :a => "a1", :b => "b1" }
        show_vars :a => "a2",
                  :b => "b2"    # context is { :a => "a2", :b => "b2" }
    end
end

activity :show_vars do          # requires :a and :b in the context
    log "({:a}, {:b})"
end
```

Figure 33: Execution model of the provenance-enabled XPFLOW. The context can be modified inside a workflow as it is done with *sequence* pattern and two calls to *show_vars*. The context is not global – the changes apply only to the activities deeper in the structure. The execution of *main* will result in two lines: `(a1, b1)` and `(a2, b2)`.

```
process :main do
    a :node => 'A'
    other :node => 'B'
    set :nodes, [ 'A', 'B' ]
    foreach :node => :nodes do
        both
    end
end
```

```
activity :a do
    log "executes on A"
end

process :other do
    log "executes on B"
end

activity :both do
    log "executes on {:node}"
end
```

Figure 34: Example of decentralized workflow execution. See Figure 35 for a readable log of its execution.

The decision to rewrite the system was motivated by inflexible implementation of the previous system and the fundamental constraints posed by its execution model. The new prototype is available for download[3].

In the following sections, the focus is on the set of features that are either distinct or new when compared with the previous implementation. First, we discuss the new model of execution that gives rise to simpler implementation and new features. Second, the decentralized execution of workflows, a powerful new feature based on this model, is the subject of the next section. Then the caching subsystem is presented and explained, followed by the explanation of how the state of the experiment, as well as provenance are stored in a generic key-value store.

5.5.1.1 *Execution model*

The principal departure from the previous model of execution is the removal of unconstrained variables in the workflows. Instead a context (or a *scope*) is passed between the workflow steps. It contains the provenance of process such as parameters to activities, the current position in the workflow, the

3 http://xpflow.gforge.inria.fr/thesis/experiments/prov2015.tar.xz

name of the executing node, etc. This information is everything that is necessary to execute a given activity or a workflow. The fact that there is no global, shared state paves the way to the decentralized execution of experiments and caching of partially executed experiments.

Activities and workflows do not explicitly accept parameters as it was done in the previous execution model. Instead, the parameters are contained in the execution context and therefore are *implicitly* required as the result of being referred inside an activity or a workflow. This is illustrated in Figure 33.

This model may bear resemblance to the *workitem* as it is often used in Business Process Management systems. It is a state that is passed between workflow steps and generally modified by them. A common analogy is that of a dossier with files that is transferred and amended by responsible people or administrative units. It is therefore a form of shared state, something that we entirely avoid in our approach and that enables interesting features and scalability.

Node failures are treated as before, that is, by default the failures cause the workflow execution to fail unless special patterns are used. However, in the new model the patterns store provenance data concerning their execution. For example, the *try* pattern stores provenance of failed executions.

5.5.1.2 *Decentralized execution*

The new model enables the implementation of decentralized execution of experiments (see *Control structure* in Section 3.3.9). Although this feature does not contribute to the collection of provenance directly, it makes the implementation simpler and has more general advantages as well. First, the user does not have to treat remote nodes in any special way, but simply make the workflow execute transparently in the context of a remote node. The results of remote execution, including provenance information, are sent back to the main node and stored in a database. Second, remotely executed workflows can refer to other remote nodes too, making the remote execution of experiments (e. g., from a personal machine of the experimenter) or using nested testbeds such as the one presented in Figure 20, straightforward to implement. Finally, decentralized execution has the potential of improving scalability.

Remote execution of workflows is triggered by the change of the `node` context variable. The new value is interpreted as the name of a node that will execute a workflow pattern or an activity. For instance, the workflow `:main` in Figure 34 executes the activity `:a` on the node `A`, and the workflow `:other` on the node `B`, all that in a transparent way.

The nodes of the experiment must have their clocks synchronized (NTP precision is sufficient for all practical purposes), since otherwise the time of events could violate causal relationships. A possible solution that does not require fine-grained synchronization consists in measuring the time of subservient workflow execution with respect to the node that initiated its execution (cf. Lamport timestamps). Although it avoids the violation of causality, the latency between the nodes skews each relative time measurement.

```
Start [main]
|-- Start [process]
|   |-- Start [a]
|   |   executes on A
|   |   Finish [a] (0.00 s)
|   |-- Start [other]
|   |   |-- Start [process]
|   |   |   |-- executes on B
|   |   |   Finish [process] (0.00 s)
|   |   Finish [other]
|   |-- Setting 'nodes' to '["A", "B"]'
|   |-- Start [foreach]
|   |   |   |-- Start [both]
|   |   |   |   executes on A
|   |   |   |   Finish [both] (0.00 s)
|   |   |   |-- Start [both]
|   |   |   |   executes on B
|   |   |   |   Finish [both] (0.00 s)
|   |   Finish [foreach] (0.63 s)
|   Finish [process] (1.34 s)
Finish [main] (1.34 s)
```

```
Start [main] (cached)
|-- Start [process] (cached)
|   |-- Using cache for 'a'
|   |-- Start [other]
|   |   |-- Start [process]
|   |   |   |-- executes on B
|   |   |   Finish [process] (0.00 s)
|   |   Finish [other]
|   |-- Setting 'nodes' to '["A", "B"]'
|   |-- Start [foreach] (cached)
|   |   |   |-- Using cache for 'both'
|   |   |   |-- Using cache for 'both'
|   |   Finish [foreach] (0.63 s, cached)
|   Finish [process] (1.34 s, cached)
Finish [main] (1.34 s, cached)
```

Figure 35: Caching of workflow execution. The execution log on the left executes with an empty cache, whereas the right one uses the previously populated cache. Although activities of the workflow are not actually executed, making the execution faster, the timing information remains the same and the cache is not modified.

Finally, it is also assumed that all participating nodes have access to the same description of the experiment. This means that it must be accessible via a network file system (e. g., NFS) or distributed manually to all nodes participating in the experiment before.

5.5.1.3 *Caching*

The presented model of execution makes caching of already finished workflows and activities simple to implement. By default, XPFlow caches the execution of both patterns and activities. However, the workflow patterns are executed even on subsequent executions, but the cache is used instead of the actual executions. This preserves the structure of the workflow in the textual output, but does not overwrite previous runtime information. This is illustrated in Figure 35.

The caching system implemented in the new framework provides an alternative to the checkpointing introduced in Section 4.3.3.3. Indeed, the state of the experiment is automatically cached as the execution progresses, and restarting it will pick up directly after the last successful step. The insertion of named checkpoints, such as those available in the previous framework, should be possible to implement without much effort.

The disadvantage of the current caching system, which is also shared by the previous implementation, is that the cache is not invalidated when the underlying experiment description changes. Although XPFlow will properly execute an activity that was *appended* to an already cached workflow, it will not behave correctly if the activities are *removed*, *reordered* or *modified*, and

in some cases such changes may lead to unexpected and wrong results. The solution to this problem is not obvious, but would arguably require a robust method to find differences between cached workflow execution and the new description, and then migrate or invalidate cache entries. The difficulty of doing this is somewhat similar to the problem of migrating a database from one schema to another.

5.5.1.4 *Use of key-value store*

The current implementation relies heavily on a persistent key-value store that contains all provenance information, as well as the whole state of experiment execution is stored in it. On the other hand, the provenance of description is not stored directly, but is still referenced by the provenance of process in enough detail to make the provenance complete.

The execution of an experiment consists in executing workflows and activities which store runtime information in this key-value store. All activities and workflow patterns store their timing information automatically. Similarly, when a datum is stored in the store, it is annotated with the context of its execution and the current timestamp.

The current implementation uses a centralized storage. All stores, including initiated by the remote workflow execution, are redirected to the principal node which stores the information on disk. Although this approach suffers from potential scalability problems, it enables the use of XPFlow with experiments spanning isolated networks, or the remote management of experiments from the machine of the XPFlow user. More scalable and distributed key-value stores can be used as well, but they normally require all the nodes to be in the same network. Ideally, a hybrid approach could be used to take advantage of any solution depending on the particular needs.

The stores from remotely executing activities are immediately pushed to the principal node to ensure that it is stored as soon as possible. However, other strategies could be used: for example the data could be sent at the end of the remote execution in a large batch, or even downloaded at the very end of the experiment execution. Although it has the potential of improving scalability and overall performance, it negates most advantages of caching, as the values may not be stored on durable storage soon enough.

5.5.2 *Description of the experimental scenario*

The experiment described in this section is purposely simple to show important aspects of our approach and implementation. It consists in running a simple experiment that consists of the following steps:

1. Installation (unless already present) of necessary software on participating nodes. We use the *STREAM* benchmark which is one of the benchmarks offered by the *hpcc* suite of benchmarks. Its purpose is to measure the speed of RAM.

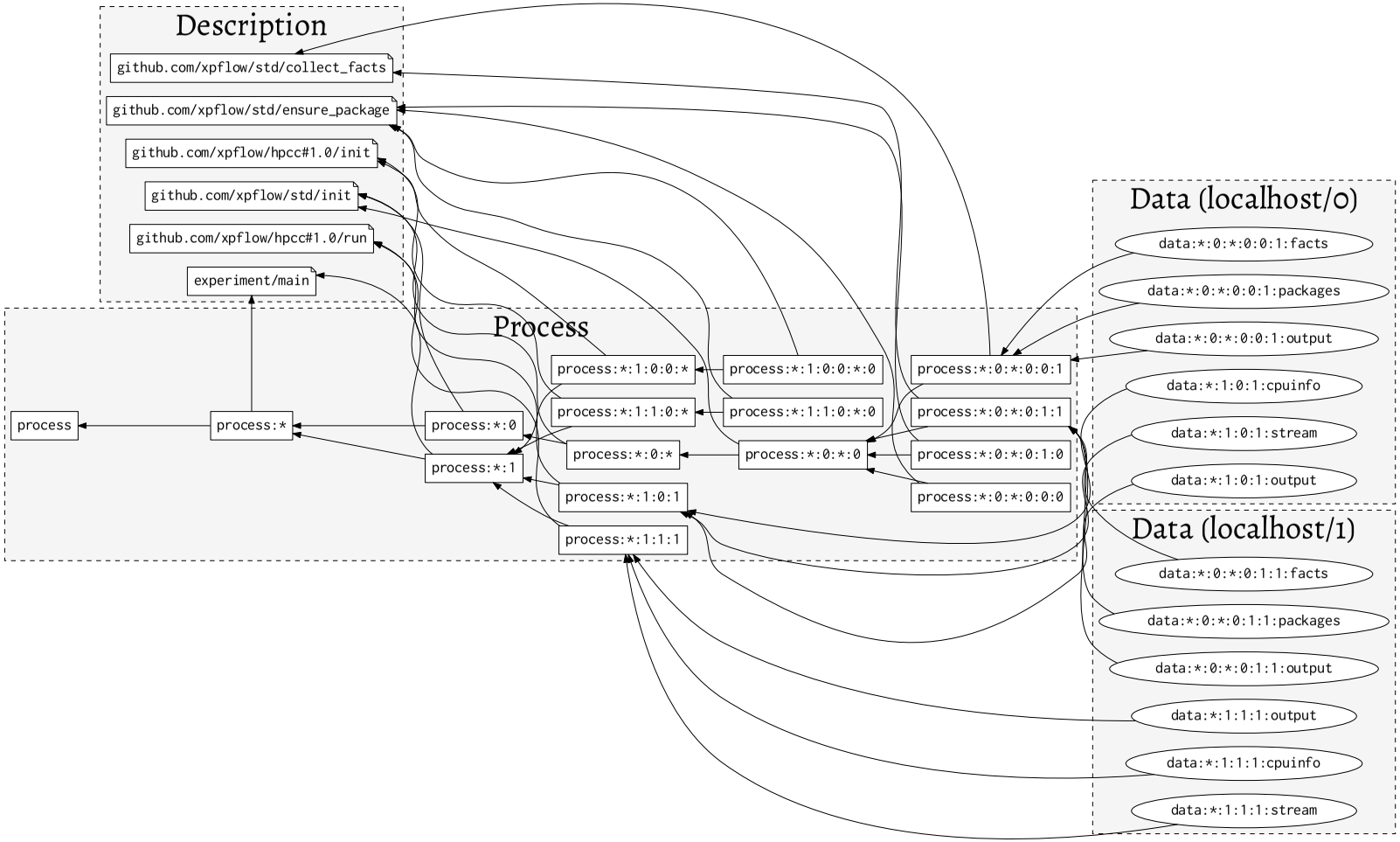


Figure 36: Complete provenance graph containing all provenance artifacts. Three types of provenance interact with each other: the provenance of description (boxes with a fold), the provenance of data (ellipses), and the central provenance of process (regular boxes). Note the same structure as in Figure 31.

2. Collection of information about participating nodes (and storing it as data provenance). This is achieved with the *facter*[4] utility that queries the host system about its peripherals, parameters, network addresses, and other useful pieces of information.
3. Running *hpcc* on all nodes and collecting the results of the *STREAM* benchmark.

The experiment is run on a single machine, with "remote" nodes being reached over *localhost*, with names of the form *localhost/0*, *localhost/1*, etc. The operating system is Debian Linux. The only requirements of the current prototype are: a Ruby interpreter (the 2.x line), an SSH client (if remote execution is used), and git (if some dependencies of the experiment are not yet available locally).

5.5.3 *Discussion*

This sections discusses the experiment in detail by looking at the three types of provenance collected before and during its execution.

5.5.3.1 *Provenance of description*

The graph of dependencies between modules involved in the experiment is presented in Figure 37. The experiment consists of 3 modules that together form a directed, acyclic graph. The main module, *experiment*, is stored locally, contains the main workflow, and imports workflows and activities from *hpcc* module (at version *1.0*). The *github.com/xpflow/std* module is a dependency shared by *all* modules, as it is implicitly imported by each one of them (with the sole exception of *itself*, because it would cause a cyclic dependency).

Special commands are available to download the complete dependency tree, verify consistency of the experiment (e. g., check whether all referenced symbols exist and no cycles in the dependency graph exist), or prepare the complete archive with all dependencies and XPFLOW itself. The latter command can be used to package the experiment description for a publication. It can be also used to automatically distribute the experiment description among the nodes participating in the experiment.

The listing of the most important parts of the experiment description is presented in Figure 38. The activities of the involved modules are also referenced in the global overview of the collected provenance in Figure 36.

5.5.3.2 *Provenance of process*

The provenance of process forms a base that consists of the state of the experiment execution. As it was described previously, its form is hierarchical with subworkflows pointing to their enclosing workflows. This can be seen in Figure 36, where the provenance of process is a tree rooted at the node named *process*.

4 https://puppetlabs.com/facter

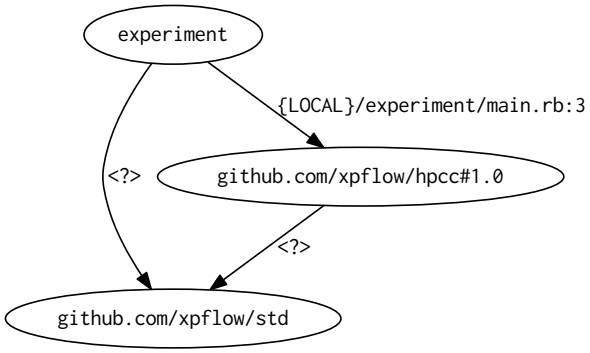


Figure 37: Dependency graph of modules involved in the experiment. The standard module is imported by default, hence the edges pointing to it have no context information. The import of *hpcc#1.0*, however, was done in the `main.rb` file (line 3).

In Figure 39, the object that represents one of the nodes is shown. It also contains various pieces of information:

- the name of the activity (`:collect_facts`),
- the start, the end and the duration of its execution,
- the context (scope) of the execution, which in particular contains the name of the executing node, pointer to the location in the provenance of description, and other details.

The identifiers of the nodes in the provenance of process are made unique by encoding the information on how the workflows are nested inside each other. For example, *process:** represents the workflow `:main` (cf. Figure 38), whereas the node *process:*:1* contains information pertaining to the execution of the `foreach` pattern. In general, the unique identifier is formed by appending a unique suffix to the key of the parent workflow (but a special case exists for some patterns, such as `foreach` loop). Not only this scheme makes the keys globally unique within a single experiment, but is additionally put to good use by the caching subsystem. Moreover, it can be used to extract and obtain the workflow structure purely from the collected provenance of process.

The provenance of process can be queried and analyzed, for example to obtain visualizations such as the one in Figure 36. Moreover, it can be exported in a portable format and analyzed elsewhere.

5.5.3.3 *Provenance of data*

The provenance of data consists of data stored during the execution along with metadata that explains the context where it was collected (see Figure 40). In our example only a small number of data artifacts is collected and they are shown as ellipses in the overview in Figure 36. This includes data collected automatically (such as standard output and error streams of executed commands), but also explicitly stored values. For this purpose the `data` command exists, that stores the given value with its context and associated name. This

```
# module: std

activity :ensure_package do
    pkg = get :pkg
    if Std.packages.include?(pkg)
        log "Package #{pkg} present."
    else
        log "Installing #{pkg}"
        output("files/install.sh")
    end
end

activity :collect_facts do
    facts = output('facter --json')
    facts = JSON.load(facts)
    data :packages, Std.packages()
    data :facts, facts
end

process :init do
    foreach :node => :nodes do
        ensure_package :pkg => 'facter'
        collect_facts
    end
end
```

```
# module: experiment

import "github.com/xpflow/hpcc#1.0"

process :main do
    std.init
    foreach :node => :nodes do
        hpcc.init
        hpcc.run
    end
end


# module: hpcc

activity :run do
    out = output('files/run_hpcc.sh')
    stream = out.scan(/^(.*STREAM.*)=(.+)$/)
    h = {}
    stream.each do |k, v|
        h[k] = v.to_f
    end
    data :stream, h
    file :cpuinfo, '/proc/cpuinfo'
end

process :init do
    std.ensure_package :pkg => 'hpcc'
end
```

Figure 38: Listing of the experiment code. The main experiment consists of a workflow that references external modules (via `import` directive). Some details are simplified or omitted for brevity.

```
{
    "duration": 1.226482629776001,
    "finish": 1442023171.3166747,
    "key": "process:*:1:*:0:0:1",
    "name": "collect_facts",
    "scope": {
        "chain": [ "localhost", "localhost/0" ],
        "ident": [ "*", 1, "*", 0, 0, 1 ],
        "location": "{STD}/std/pkg.rb:36",
        "module": "github.com/xpflow/std",
        "name": "collect_facts",
        "node": "localhost/0",
        "nodes": [ "localhost/0" ],
        "options": {},
        "pstack": []
    },
    "start": 1442023170.090192
}
```


Figure 39: Example of the provenance of process. This object is stored in the key-value store under the unique key *process:*:1:*:0:0:1*. It contains the context pertaining to the execution of *collect_facts* activity from module *github.com/xpflow/std*. It can also be seen that it executed on a node named *localhost/0*.

is illustrated by `:collect_facts` activity (Figure 38) that stores the list of installed packages and the information obtained from the *facter* program.

The same activity uses the `file` command to collect the contents of a file as the provenance of data. Therefore, instead of downloading the files manually they can be stored transparently in a key-value store as any other data artifact, and then retrieved back. This functionality partially implements the feature *File management* mentioned in Section 3.3.8.

In this example, the data objects do not reference each other and more generally the current implementation does not give means to express such links. Although an implementation of such feature is not difficult, the context of our domain does not require to trace such relations.

A simple interface was implemented to query and access the provenance of data. It can be used to extract relevant data from the key-value store into a common format (e. g., CSV, JSON, YAML) and import it into an environment better suited for further analysis. The query consists of a *filter* and a *selector* (corresponding to *WHERE* and *SELECT* clauses from SQL), written in a domain-specific language based on Ruby.

For example, to get the list of all nodes with their IP addresses and kernel versions, one can execute the following query (cf. Figure 40):

```
Filter: { name == "facts" }
Selector: { [ scope.node, value.ipaddress, value.kernelversion ] }
```

Similarly, to extract the results of the *STREAM* benchmark along with the time when it was collected, one can run:

```
{
    "key": "data:*:1:*:0:0:1:facts",
    "name": "facts",
    "process": "process:*:1:*:0:0:1",
    "scope": { ... },
    "tags": [],
    "time": 1442023171.3049905,
    "value": {
        "architecture": "amd64",
        "blockdevice_sda_model": "SAMSUNG SSD PM83",
        "kernelversion": "4.1.0",
        "ipaddress": "10.4.0.3",
    }
}
```


Figure 40: Example of the provenance of data. Each datum has a non-unique name, a value, optional tags, its context and reference to the provenance of process (under the key *process*). The pair consisting of the name and the reference to the provenance of process make the datum unique. Note that the scope is omitted as it is similar to the one in Figure 39.

```
Filter: { name == "stream" }
Selector: { [ scope.node, value.SingleSTREAM_Copy, time ] }
```

The same principle can be used to query the provenance of process. For example, the next query lists the duration of `:collect_facts` activity on each node (cf. Figure 39):

```
Filter:   { name == "collect_facts" }
Selector: { [ scope.node, duration ] }
```

This approach inherits all benefits of having a complete programming language (i. e., the entirety of Ruby standard library can be used), while remaining easy to use at the same time. The language does not support the advanced data analysis which should be performed with external software more suited to this task.

5.5.4 *Conclusions*

The proposed model of provenance was implemented in the prototype that is able to collect provenance of all types and query it later. Some observations were made that may serve as a future guidance for researchers and implementers.

First, it must be stated again that the provenance model relies heavily on the structured representation of experiments based on workflow patterns, as it was assumed in Section 5.4. In particular the structure of the provenance of process is directly implied by the workflow structure of an experiment. Existing approaches may or may not map well to this model of provenance. For

example, the *imperative* approaches to the description of experiments (cf. Table 3a) can be arguably ruled out (i. e., *Naive approach*, *Workbench*, *Expo*, *OMF*, *NEPI* and *Execo*) as they do not have a structured way to look at the experiment description. Among the remaining tools, *Plush/Gush* has a *declarative* approach to the description of experiments, but only superficially manages the execution of the proper experiment. *Weevil* has the most structured model and could implement the proposed provenance model for its predefined and rigid workflow.

On the other hand, the provenance of description and the provenance of data as currently designed and implemented are mostly independent from the workflow model. It is uncertain, however, whether their use without the binding of the provenance of process is useful at all. It was mentioned a few times that the provenance of process occupies the central position in the design, and in our implementation it is even equivalent to the state of the workflow execution.

The evaluation made it clear that the new execution model (as described in Section 5.5.1.1) is more suitable for the purpose of provenance tracking than the previous one. Having a single object representing the whole state of execution tremendously simplifies implementation and enables a group of features that were unavailable before, such as decentralized execution of workflows. It is highly preferable to use this model for future realizations of similar systems.

Moreover, the original observation that a key-value store will be sufficient to store provenance proved to be correct. This is the result of a few technical decisions as well as the original assumption that the experiment does not produce large volumes of data nor operates on these data in complex ways. The fact that our model does not trace transformations is the main difference when compared with the collection of provenance in scientific workflows. It is conceivable that a more data-centric approach to the provenance of data could be tried, but its use in the studied domain is probably unnecessary.

Although currently the storage is centralized, there is nothing conceptual or technical that forbids the use of a distributed key-value store. Again, this is enabled by the decentralized model of execution and the use of key-value store as a storage for the whole state of the experiment, including collected provenance. It is recommended to use this kind of architecture in similar projects as it offers scalability when controlling large systems.

5.6 SUMMARY

This chapter made four contributions. First, we analyzed the provenance in different disciplines of computer science. Then we observed that provenance can be split into three different types: the provenance of data, the provenance of description and the provenance of process. After that, we designed a provenance system for distributed systems research that can capture all of them (see Figure 31). Finally, we presented our early prototype of the experiment management engine that is based on workflow patterns and collects much

of the provenance information automatically. The exemplary case study was presented to show the structure of provenance, the details of the currently developed prototype and other useful observations. In particular, the implementation provides three more features identified previously in Chapter 3: *Provenance tracking*, *File management* and decentralized *Control structure*.

With that disappearance, (...), came the end,
the final end of Eternity. - And the beginning of Infinity.

— Isaac Asimov, *The End of Eternity*

6 CONCLUSIONS AND FUTURE WORK

This chapter concludes the research presented in this thesis. First, the summary of the main results is given. The summary is then followed by the structured analysis of possible ways to improve this work, the results and the presented approaches.

6.1 SUMMARY OF THE RESULTS

The main goal of this work was to study how the experimentation in the domain of distributed systems can be improved with the methods and approaches used in Business Process Management and related domains. The focus was on the large-scale, *in-situ* experiments which are more and more common given the current trends, modern architectures, and the limits imposed by other methodologies. Our research provides three principal observations.

The analysis of the existing literature, current problems and developed solutions in Chapter 2 and Chapter 3, allowed us to understand and partially formalize our problem. This analysis ended with the creation of a framework to evaluate experiment management tools. Not only it uncovered the missing functionality of existing systems, notably *Provenance tracking*, but also was used by us to evaluate our solution later in this work.

In the light of the goals stated in Section 1.2, the general conclusion coming from this work is that the model based on workflow patterns and other methods of Business Process Management, shows promise in helping the researchers that want to run challenging, large-scale experiments. In particular, the evaluation in Chapter 4 showed that our approach can be used to manage large-scale experiments in a robust, fault-tolerant and scalable manner. We also developed a set of properties that experimental workflow must possess to provide such properties, as well as the set of patterns that focus the domain-specific needs of large-scale experimentation.

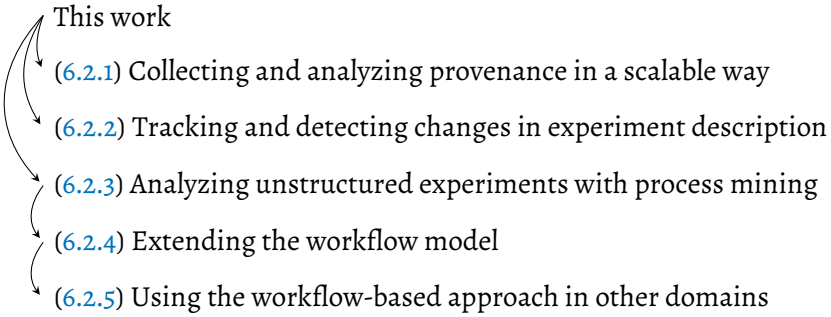
Finally, by following the implications of using workflow-based model for experiment execution, the collection of provenance in experiments was addressed in Chapter 5. This led to the classification of provenance into three types: the provenance of description, process and data. Consequently, a design of a system collecting these types of provenance was proposed and then implemented. The prototype was shown to collect provenance that can improve the understanding and reproducibility of experiments.

All in all, the representation of experiments based on workflow patterns identified in Business Process Management is a viable alternative to existing ways to design, run and reason about experiments. First, our current implementation implements 13 binary features identified in the survey presented in Chapter 3, and three more (*Provenance tracking*, *File management* and decentralized *Control structure*) in the new prototype. This only falls short of the more established tools such as OMF and NEPI, but the future work should alleviate this. Importantly, however, *Provenance tracking* stands out as a feature that only one tool implemented before (*Emulab Workbench*), and, as was shown in Chapter 4, the workflow-based approach shows good scalability, robustness and fault-tolerance when running large-scale experiments.

All our results show that the structured way of representing experiments, in this case by means of workflow patterns, is an advancement over many existing approaches and can be used to improve the quality of existing and future research in large-scale experiments in distributed systems research. Finally, there are many promising ways to improve it even more, as will be discussed in the following section.

6.2 FUTURE WORK

There are a few promising ways to improve and build upon our work that can be split into two mostly independent paths of investigation. The first group consists of two research directions presented in Section 6.2.1 and Section 6.2.2 that build directly on our work and our implementation. Note that both directions can be taken independently as they do not depend on each other. The second group takes a more general view on the approach and the obtained results to draw conclusions that are not tied to implementation details, but applying to the general methodology instead. These three research directions are presented in Section 6.2.3, Section 6.2.4 and Section 6.2.5. In this case, the presentation order respects the recommended path of investigation. This is presented in the following summary:



6.2.1 *Collecting and analyzing provenance in a scalable way*

This work showed that our methods can be used to manage large experiments with success. Whether this scalability applies equally to the collection

of provenance discussed in Chapter 5 is an obvious next step towards robust execution of large-scale experiments together with the collection of provenance.

Successful research in this area would enable the automatic collection of provenance in large-scale experiments in distributed systems research, an activity that is almost entirely missing in this domain. The advantages of collecting provenance were already mentioned, but they apply even more to large-scale experiments for which it is conceptually impossible to track every component in the experiment or collect provenance manually.

The first challenge in this research direction is the technical difficulty of scaling the collection of provenance to the large number of nodes and large volumes of collected data. Many low-level problems will inevitably arise, but some of them were already discussed in Section 5.5.1.4.

Additional and more interesting challenge comes from the fact that large-scale, complex experiments will necessarily produce more provenance information. This may lead to difficulties common to the domains that operate on large volumes of data, often referred to as *Big data*. Not only storage and efficiency problems are likely to arise, but also the interpretation of the data may become non-trivial. These are problems commonly addressed by *Big data*, *data mining* and *data visualization* communities and may apply to our problem as well.

A possible strategy is to export data from our provenance model to standards such as *The Open Provenance Model* [131], and then use tools that can use it. Approaching the problem from this angle has the advantage that a larger community could profit from positive results. Nevertheless, a few challenges still exist. First, this model may be not suitable for the representation of provenance that is collected by our approach as our model is far from being data-centric. Second, *The Open Provenance Model* does not specify protocols for storing and querying the provenance, hence a lot of work may be needed to obtain a practical solution.

6.2.2 *Tracking and detecting changes in experiment description*

Currently, the description of experiments is not tracked at the level of workflow representation, but only at the level of files in a version control system. It would be advantageous to be able to track and detect changes made to workflows and therefore better represent changes and history of the description of experiments.

The first application of this capability directly contributes to reproducibility by allowing one to meaningfully compare provenance data between different variations of a single experiment. For example, a researcher could modify the experiment and compare newly obtained results with the original ones, despite the differences in the experiment description. More generally, the historical provenance of experiments could be stored along the description of experiments and then analyzed to see if and how it evolved. This functionality generalizes the features of VisTrails (see Section 5.2.3).

Moreover, this work can address the problem of migrating the state of the executing experiment according to the changes in its description (see Section 5.5.1.3). This would enable the more traditional, iterative approach to the development of experiments, as well as save precious time required to rerun experiments.

The main difficulty consists in finding the basic operations that can be applied to workflows and then designing an algorithm that maps differences between workflows to a list of these operations. In general, the result may not be unique, hence heuristics must be applied that exploit domain-specific knowledge about how the workflows are usually modified. A well developed algorithm of this type could even find common patterns between unrelated experiments, something potentially useful in relation to *the analysis of unstructured experiments with process mining* described in Section 6.2.3.

Research in similar areas area exists, although with the focus on the *conceptual graphs* [58]. Interestingly, most of the related research seems to focus on computing *similarity measures* between graphs and on the *visualization* of differences, less so on finding what the differences exactly are.

6.2.3 *Analyzing unstructured experiments with process mining*

In Chapter 3 we identified the features of experiment management tools, not by looking what researchers would like to have, but by analyzing the existing work in this area. Similarly, the experimental patterns discussed in Chapter 4 were inspired by our own experience in this domain. Alternative approach is to apply process mining to analyze and understand experiments, and eventually turn the informal and unstructured activities of a researcher into a formal representation that is easier to work with (see also Section 2.5.4).

Positive results coming from such a research have immediate and practical applications. First, the study of experimental habits could be done, possibly uncovering interesting trends, patterns and domains that need more focus and improvement. Second, the prospective mining methods could be used to migrate existing *ad hoc* experiments to structured approaches, such as the one studied in this work. This has the potential of improving the general quality of experimental research in the domain of distributed systems and encouraging the use of structured approaches to experimentation.

A range of process mining methods exists in the domain of Business Process Management. The common approach consists in collecting a series of logs from a studied system and inferring a workflow structure from them [3]. It is worth exploring whether these methods apply to this problem, even if some challenges exist. First, our case poses additional, domain-specific problems, notably the fact that there are experimental patterns (e. g., parallel execution on multiple nodes) that are not covered by the standard workflow patterns. Second, it is not clear whether the scale, the complexity and the amount of data produced by existing computer systems and stored in logs can be analyzed to obtain any useful insight. Moreover, there are so many

alternative ways to interact with computer systems that it is not obvious if any technique may capture all of them.

6.2.4 *Extending the workflow model*

The next research direction consists in exploring a different model of execution than the one proposed in Chapter 4 and subsequently modified in Chapter 5. The proposed model may be too restrictive or lack features and patterns that are useful for running experiments.

Ultimately, the goal of extending the model is to bring more features to the control of experiments, in particular the features that were identified in Chapter 3 and are still missing in our approach. Among the features that may particularly require changes and improvements to the model are: *Low entry barrier*, *Fault injection*, *Workload generation* and *Platform monitoring* (cf. Table 5). If such changes could be introduced without harming previous functionality, it would make it the most feature complete experimentation tool.

A couple of difficulties can be expected. First, some of the mentioned features, for example *Fault injection* and *Workload generation* may require an ability to execute multiple parallel processes in parallel. This is non-trivial considering that the current model is essentially synchronous and that it interferes with the caching. Second, another difficulty lies in making decisions on how the model should be changed. Some of them may lead to changes to the core elements of the main approach and hence invalidate the findings made in this work. Fortunately, the results of the research direction proposed in Section 6.2.3 should help in this respect.

6.2.5 *Using the workflow-based approach in other domains*

It was stated a few times that the methods presented in this work target *in-situ* experiments in distributed systems research. This allowed us to make some simplifying assumptions leading to a simpler design. However, it does not make it impossible to use this or similar approach in other domains, even if some changes may be necessary.

A successful application to other domains would bring all or a subset of features that were explored in this work. Although the main focus is on the scientific domains, one can imagine applying it to tasks unrelated to science, such as configuration management, monitoring, testing or benchmarking.

The main difficulty is caused necessarily by the assumptions that were made in this work. We rely on *in-situ* testbeds which provide full access to the experimental platform and provide a Unix environment. This is not the case in other methodologies used in experimental distributed systems research and some non-standard testbeds, such as those using low-performance, sensor nodes. Moreover, considering the collection of provenance, an assumption was made that the experiments do not produce or transform large volumes of pure data. This led to a simpler model, but also a model that does not

store information about data transformations, contrary to scientific workflows.

To help with this task, the outcomes of work in Section 6.2.3 could be used to better understand the targeted domains. Similarly, an extended and more general model introduced as the result of the work in Section 6.2.4 may prove to be easier to apply in other domains.

BIBLIOGRAPHY

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

[2] Michael J. Adams, Arthur H.M. ter Hofstede, and Marcello La Rosa. Open source software for workflow management : the case of YAWL. *IEEE Software*, 28(3):16–19, 2011.

[3] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining Process Models from Workflow Logs. In *Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '98, pages 469–483, London, UK, UK, 1998. Springer-Verlag.

[4] Bob Aiken, Victor Bahl, Bob Bhattacharjee, Bob Braden, et al. Report of NSF workshop on network research testbeds. *National Science Foundation Directorate for Computer and Information Science and Engineering (CISE) Advanced Networking Infrastructure & Research Division*, 2002.

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

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

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

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

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

[10] M. David Allen, Adriane Chapman, Barbara Blaustein, and Lisa Mak. What do we do now? Workflows for an unpredictable world. *Future Generation Computer Systems*, 42(0):1 – 10, 2015.

[11] A.I. Avetisyan, R. Campbell, I. Gupta, M.T. Heath, S.Y. Ko, G.R. Ganger, M.A. Kozuch, D. O'Hallaron, M. Kunze, T.T. Kwan, K. Lai, M. Lyons, D.S. Milojicic, Hing Yan Lee, Yeng Chai Soh, Ng Kwang Ming, J-Y. Luke, and Han Namgoong. Open Cirrus: A Global Cloud Computing Testbed. *Computer*, 43(4):35–43, April 2010.

[12] Sepideh Azarnoosh, Mats Rynge, Gideon Juve, Ewa Deelman, Michal Nieć, Maciej Malawski, and Rafael Ferreira da Silva. Introducing PRECIP: An API for Managing Repeatable Experiments in the Cloud. *IEEE Cloudcom*, 1:1, 2013.

[13] Mircea Bardac, Razvan Deaconescu, and Adina Magda Florea. Scaling Peer-to-Peer testing using Linux Containers. In *9th RoEduNet International Conference*, pages 287–292, June 2010.

[14] Roger S. Barga and Luciano A. Digiampietri. Automatic capture and efficient storage of e-Science experiment provenance. *Concurrency and Computation: Practice and Experience*, 20(5):419–429, 2008.

[15] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[16] Adam Barker and Jano Hemert. Scientific Workflow: A Survey and Research Directions. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 4967 of *Lecture Notes in Computer Science*, pages 746–753. Springer Berlin Heidelberg, 2008.

[17] Anirban Basu, Simon Fleming, James Stanier, Stephen Naicken, Ian Wakeman, and Vijay K. Gurbani. The state of peer-to-peer network simulators. *ACM Comput. Surv.*, 45(4):46:1–46:25, August 2013.

[18] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. *SIGCOMM Comput. Commun. Rev.*, 36(4):3–14, August 2006.

[19] Louis Bavoil, Steven P. Callahan, Carlos Eduardo Scheidegger, Huy T. Vo, Patricia Crossno, Cláudio T. Silva, and Juliana Freire. VisTrails: Enabling Interactive Multiple-View Visualizations. In *IEEE Visualization*, page 18, 2005.

[20] Khalid Belhajjame, Marco Roos, Esteban Garcia-Cuesta, Graham Klyne, Jun Zhao, David De Roure, Carole Goble, Jose Manuel Gomez-Perez, Kristina Hettne, and Aleix Garrido. Why Workflows Break - Understanding and Combating Decay in Taverna Workflows. In *Proceedings of the 2012 IEEE 8th International Conference on E-Science (e-Science)*, E-SCIENCE '12, pages 1–9, Washington, DC, USA, 2012. IEEE Computer Society.

[21] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds – Part I.

[22] Olivier Biton, Sarah Cohen-Boulakia, Susan B. Davidson, and Carmem S. Hara. Querying and Managing Provenance Through User Views in Scientific Workflows. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 1072–1081, Washington, DC, USA, 2008. IEEE Computer Society.

[23] Carl Boettiger. An introduction to Docker for reproducible research, with examples from the R environment. *CoRR*, abs/1410.0846, 2014.

[24] Thomas Bourgeau, Jordan Auge, and Timur Friedman. TopHat: supporting experiments through measurement infrastructure federation. *Proceedings of TridentCom'2010, 18-20 May 2010, Berlin, Germany.*, 1:1, May 2010.

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

[26] Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. Emulation at Very Large Scale with Distem. In *SCALE Challenge, held in conjunction with the 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Chicago, United States, May 2014.

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

[28] Tomasz Buchert, Lucas Nussbaum, and Jens Gustedt. Accurate emulation of CPU performance. In *8th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Platforms (HeteroPar)*, Ischia, Italie, August 2010.

[29] Tomasz Buchert, Lucas Nussbaum, and Jens Gustedt. Methods for Emulation of Multi-Core CPU Performance. In *13th IEEE International Conference on High Performance Computing and Communications (HPCC)*, Banff, Canada, November 2011.

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

[31] Tomasz Buchert, Lucas Nussbaum, and Jens Gustedt. Towards Complete Tracking of Provenance in Experimental Distributed Systems Research. *REPPAR 2015*, 2015.

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

[33] Anton Burtsev, Nikhil Mishrikoti, Eric Eide, and Robert Ricci. Weir: a streaming language for performance analysis. In *Proceedings of the Seventh Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 6:1–6:6, New York, NY, USA, 2013. ACM.

[34] Anton Burtsev, Prashanth Radhakrishnan, Mike Hibler, and Jay Lepreau. Transparent Checkpoints of Closed Distributed Systems in Emulab. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 173–186, New York, NY, USA, 2009. ACM.

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

[36] Matthieu Caneill and Stefano Zacchiroli. Debsources: Live and Historical Views on Macro-level Software Evolution. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 28:1–28:10, New York, NY, USA, 2014. ACM.

[37] Louis-Claude Canon, Olivier Dubuisson, Jens Gustedt, and Emmanuel Jeannot. Defining and controlling the heterogeneity of a cluster: The Wrekavoc tool. *Journal of Systems and Software*, 83:786–802, May 2010.

[38] Louis-Claude Canon and Emmanuel Jeannot. Wrekavoc: a Tool for Emulating Heterogeneity. In *Heterogenity in Computing Workshop (HCW) in International Parallel and Distributed Processing Symposium (IPDPS)*, Island of Rhodes, Greece, 2008.

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

[40] Marta Carbone and Luigi Rizzo. Adding Emulation to PlanetLab Nodes. In *Proceedings of the 5th International Student Workshop on Emerging Networking Experiments and Technologies*, Co-Next Student Workshop '09, pages 41–42, New York, NY, USA, 2009. ACM.

[41] Marta Carbone and Luigi Rizzo. Dummynet revisited. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, April 2010.

[42] Marta Carbone and Luigi Rizzo. An emulation tool for PlanetLab. *Computer Communications*, 34(16):1980–1990, October 2011.

[43] Alexandra Carpen-Amarie, Antoine Rougier, and FelixD. Lübbe. Stepping Stones to Reproducible Research: A Study of Current Practices in Parallel Computing. In Luís Lopes, Julius Žilinskas, Alexandru Costan, RobertoG. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, StephenL. Scott, Stefan Lankes, Christian Lengauer, Jesus Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 499–510. Springer International Publishing, 2014.

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

[45] Vinton G. Cerf. Avoiding "Bit Rot": Long-Term Preservation of Digital Information. *Proceedings of the IEEE*, 99(6):915–916, June 2011.

[46] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[47] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Found. Trends databases*, 1(4):379–474, April 2009.

[48] Benoit Claudel, Guillaume Huard, and Olivier Richard. TakTuk, adaptive deployment of remote executions. In *Proceedings of the 18th ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, HPDC '09, pages 91–100, New York, NY, USA, 2009. ACM.

[49] Shirley Cohen, Sarah Cohen-Boulakia, and Susan Davidson. Towards a Model of Provenance and User Views in Scientific Workflows. In *Proceedings of the Third International Conference on Data Integration in the Life Sciences*, DILS'06, pages 264–279, Berlin, Heidelberg, 2006. Springer-Verlag.

[50] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[51] Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NaaS: Network-as-a-Service in the Cloud. In *2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Berkeley, CA, 2012. USENIX.

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

[53] Vasa Curcin, Moustafa Ghanem, and Yike Guo. The design and implementation of a workflow analysis tool. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 368(1926):4193–4208, September 2010.

[54] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[55] Susan B. Davidson, Sarah Cohen Boulakia, Anat Eyal, Bertram Ludäscher, Timothy M. McPhillips, Shawn Bowers, Manish Kumar Anand, and Juliana Freire. Provenance in Scientific Workflow Systems. *IEEE Data Eng. Bull.*, 30(4):44–50, 2007.

[56] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *In Proceedings of ACM SIGMOD*, pages 1345–1350, 2008.

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

[58] A. de Moor H. Delugach. Difference graphs. In R. Meersman and Z. Tari, editors, *Proc. of Common Semantics for Sharing Knowledge: Contributions to the 13th International Conference on Conceptual Structures (ICCS 2005) (Kassel, Germany)*, pages 41–53. Kassel University Press, 2005.

[59] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[60] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, October 2007.

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

[62] Ewa Deelman, Gurmeet Singh, Mei hui Su, James Blythe, A Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, G. Bruce Berriman, John Good, Anastasia Laity, Joseph C. Jacob, and Daniel S. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13:219–237, 2005.

[63] Y. Denneulin, E. Romagnoli, and D. Trystram. A synthetic workload generator for cluster computing. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, page 243, april 2004.

[64] Peter J. Denning. Is computer science science? *Communications of the ACM*, 48(4):27–31, 2005.

[65] Edsger W. Dijkstra. Notes on Structured Programming. circulated privately, April 1970.

[66] Jeff Dike. A user-mode port of the linux kernel. In *Proceedings of the 4th Annual Linux Showcase & Conference (ALS)*, volume 4, pages 7–7, Berkeley, CA, USA, 2000. USENIX Association.

[67] Jeff Dike. User-mode Linux. In *Proceedings of the 5th Annual Linux Showcase & Conference (ALS)*, volume 5, pages 2–2, Berkeley, CA, USA, 2001. USENIX Association.

[68] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A Safe and Policy-Free System for Software Deployment. In *Proceedings of the 18th USENIX Conference on System Administration*, LISA '04, pages 79–92, Berkeley, CA, USA, 2004. USENIX Association.

[69] Eelco Dolstra and Andres Löh. NixOS: A Purely Functional Linux Distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367–378, New York, NY, USA, 2008. ACM.

[70] David L. Donoho, Arian Maleki, Inam Ur Rahman, Morteza Shahram, and Victoria Stodden. Reproducible Research in Computational Harmonic Analysis. *Computing in Science and Engineering*, 11(1):8–18, January 2009.

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

[72] Dr. Chris Drummond. Reproducible Research: a Dissenting Opinion. (no), September 2012.

[73] Olivier Dubuisson, Jens Gustedt, and Emmanuel Jeannot. Validating Wrekavoc: A tool for heterogeneity emulation. In *Heterogenity in Computing Workshop (HCW) in International Parallel and Distributed Processing Symposium (IPDPS)*, Rome Italy, 2009.

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

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

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

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

[78] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, September 2002.

[79] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an Abstract Grid Workflow Language. In *Proceedings of the 5th IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, volume 2 of *CCGRID '05*, pages 676–685, Washington, DC, USA, 2005. IEEE Computer Society.

[80] Thomas Fahringer, Radu Prodan, Rubing Duan, Jüurgen Hofer, Farrukh Nadeem, Francesco Nerieri, Stefan Podlipnig, Jun Qin, Mumtaz Siddiqui, Hong-Linh Truong, Alex Villazon, and Marek Wieczorek. ASKALON: A Development and Grid Computing Environment for Scientific Workflows. In IanJ. Taylor, Ewa Deelman, DennisB. Gannon, and Matthew Shields, editors, *Workflows for e-Science*, pages 450–471. Springer London, 2007.

[81] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. Technical report, IBM Research Division, 2014.

[82] Philip J. Fleming and John J. Wallace. How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results. *Commun. ACM*, 29(3):218–221, March 1986.

[83] Ian Foster and Carl Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications*, 11:115–128, 1996.

[84] Geoffrey Fox, Gregor von Laszewski, Javier Diaz, Kate Keahey, Jose Fortes, Renato Figueiredo, Shava Smallen, Warren Smith, and Andrew Grimshaw. FutureGrid-a reconfigurable testbed for Cloud, HPC, and Grid Computing. *Contemporary High Performance Computing: From Petascale toward Exascale, Computational Science. Chapman and Hall/CRC*, 2013.

[85] ClaudioDaniel Freire, Alina Quereilhac, Thierry Turletti, and Walid Dabbous. Automated Deployment and Customization of Routing Overlays on Planetlab. In Thanasis Korakis, Michael Zink, and Maximilian Ott, editors, *Testbeds and*

Research Infrastructure. Development of Networks and Communities, volume 44 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 240–255. Springer Berlin Heidelberg, 2012.

[86] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for Computational Tasks: A Survey. *Computing in Science & Engineering*, 10(3):11–21, 2008.

[87] Juliana Freire, CláudioT. Silva, StevenP. Callahan, Emanuele Santos, CarlosE. Scheidegger, and HuyT. Vo. Managing Rapidly-Evolving Scientific Workflows. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data*, volume 4145 of *Lecture Notes in Computer Science*, pages 10–18. Springer Berlin Heidelberg, 2006.

[88] John L. Furlani and Peter W. Osel. Abstract Yourself With Modules. In *Proceedings of the 10th USENIX Conference on System Administration*, LISA '96, pages 193–204, Berkeley, CA, USA, 1996. USENIX Association.

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

[90] Daniel Garijo, Pinar Alper, Khalid Belhajjame, Oscar Corcho, Yolanda Gil, and Carole Goble. Common motifs in scientific workflows: An empirical analysis. *Future Generation Computer Systems*, 36(0):338 – 351, 2014.

[91] Ian P Gent, Stuart A Grant, Ewen MacIntyre, Patrick Prosser, Paul Shaw, Barbara M Smith, and Toby Walsh. How not to do it. *Research Report Series - University of Leeds, School of Computer Studies*, 1997.

[92] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. crypto.stanford.edu/craig.

[93] Garth Gibson. Reflections on Failure in Post-Terascale Parallel Computing, Keynote at 2007 Int. Conf. on Parallel Processing, Xi'An, China. http://web.cse.ohio-state.edu/ lai/icpp-2007/Gibson-PDSI-ICPP07-keynote.pdf, 2007.

[94] Yolanda Gil, Varun Ratnakar, Jihie Kim, Pedro A. González-Calero, Paul T. Groth, Joshua Moody, and Ewa Deelman. Wings: Intelligent Workflow-Based Design of Computational Experiments. In *IEEE Intelligent Systems*, 2011.

[95] Jeremy Goecks, Anton Nekrutenko, James Taylor, and The Galaxy Team. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome Biology*, 11(8):R86, 2010.

[96] Omar S. Gómez, Natalia Juristo, and Sira Vegas. Replications types in experimental disciplines. In *Proceedings of the 4th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ESEM '10, pages 3:1–3:10, New York, NY, USA, 2010. ACM.

[97] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS)*, PADS '08, pages 7–14, Washington, DC, USA, 2008. IEEE Computer Society.

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

[99] Philip J. Guo. *Software Tools to Facilitate Research Programming*. PhD thesis, Stanford University, May 2012.

[100] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. DieCast: Testing Distributed Systems with an Accurate Scale Model. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 407–421, 2008.

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

[102] Bill Howe. Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science and Engineering*, 14(4):36–41, 2012.

[103] Sascha Hunold and Jesper Larsson Träff. On the State and Importance of Reproducible Experimental Research in Parallel Computing. *CoRR*, abs/1308.3648, 2013.

[104] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

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

[106] Darrel C. Ince, Leslie Hatton, and John Graham-Cumming. The case for open computer programs. *Nature*, 482(7386):485–488, February 2012.

[107] R. Jain. *The Art of Computer Systems Performance Analysis; Techniques for Experimental Design, Measurement, Simulation and Modelling*. Wiley professional computing. Wiley, 1991.

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

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

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

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

[112] Young-Hwan Kim, Alina Quereilhac, Mohamed Amine Larabi, Julien Tribino, Thierry Parmentelat, Thierry Turletti, and Walid Dabbous. Enabling Iterative Development and Reproducible Evaluation of Network Protocols. *Computer Networks*, January 2014.

[113] Thorsten Kleinjung, Kazumaro Aoki, Jens Franke, Arjen Lenstra, Emmanuel Thomé, Joppe Bos, Pierrick Gaudry, Alexander Kruppa, Peter Montgomery, Dag Arne Osvik, Herman te Riele, Andrey Timofeev, and Paul Zimmermann. Factorization of a 768-bit RSA modulus. Cryptology ePrint Archive, Report 2010/006, 2010.

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

[115] D. E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[116] Ryan K. L. Ko. A computer scientist's introductory guide to business process management (BPM). *Crossroads*, 15(4):4:11–4:18, June 2009.

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

[118] Friedrich Leisch. Sweave: Dynamic Generation of Statistical Reports Using Literate Data Analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Proceedings in Computational Statistics (COMPSTAT)*, pages 575–580. Physica Verlag, Heidelberg, 2002. ISBN 3-7908-1517-9.

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

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

[121] M. Mahalingam, D. Dutt, K. Duda, P. Agarwal, L. Kreeger, T. Sridhar, M. Bursell, and C. Wright. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348 (Informational), August 2014.

[122] Suresh Marru, Lahiru Gunathilake, Chathura Herath, Patanachai Tangchaisin, Marlon Pierce, Chris Mattmann, Raminder Singh, Thilina Gunarathne, Eran Chinthaka, Ross Gardler, Aleksander Slominski, Ate Douma, Srinath Perera, and Sanjiva Weerawarana. Apache Airavata: A Framework for Distributed Applications and Computational Workflows. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 21–28, New York, NY, USA, 2011. ACM.

[123] Stéphane Martin, Tomasz Buchert, Pierric Willemet, Olivier Richard, Emmanuel Jeanvoine, and Lucas Nussbaum. Scalable and Reliable Data Broadcast with Kascade. In *HPDIC - International Workshop on High Performance Data Intensive Computing, in conjunction with IEEE IPDPS 2014*, Phoenix, United States, May 2014.

[124] Dennis McCafferty. Should code be released? *Communications of the ACM*, 53:16–17, October 2010.

[125] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers.

[126] Timothy McPhillips, Shawn Bowers, Daniel Zinn, and Bertram Ludäscher. Scientific Workflow Design for Mere Mortals. *Future Gener. Comput. Syst.*, 25(5):541–551, May 2009.

[127] Deep Medhi, Byrav Ramamurthy, Caterina Scoglio, Justin P. Rohrer, Egemen K. Çetinkaya, Ramkumar Cherukuri, Xuan Liu, Pragatheeswaran Angu, Andy Bavier, Cort Buffington, and James P. G. Sterbenz. The GpENI Testbed: Network Infrastructure, Implementation Experience, and Experimentation. *Computer Networks*, 61:51–74, March 2014.

[128] J. P. Mesirov. Accessible Reproducible Research. *Science*, 327(5964):415–416, January 2010.

[129] J. Mirkovic, T.V. Benzel, T. Faber, R. Braden, J.T. Wroclawski, and S. Schwab. The DETER project: Advancing the science of cyber security experimentation and test. In *10th IEEE International Conference on Technologies for Homeland Security (HST)*, pages 1 –7, nov. 2010.

[130] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (INTERNET STANDARD), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2673, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

[131] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743 – 756, 2011.

[132] Luc Moreau, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven P. Callahan, George Chin Jr., Ben Clifford, Shirley Cohen, Sarah Cohen-Boulakia, Susan B. Davidson, Ewa Deelman, Luciano A. Digiampietri, Ian T. Foster, Juliana Freire, James Frew, Joe Futrelle, Tara Gibson, Yolanda Gil, Carole A. Goble, Jennifer Golbeck, Paul T. Groth, David A. Holland, Sheng Jiang, Jihie Kim, David Koop, Ales Krenek, Timothy M. McPhillips, Gaurang Mehta, Simon Miles, Dominic Metzger, Steve Munroe, Jim Myers, Beth Plale,

Norbert Podhorszki, Varun Ratnakar, Emanuele Santos, Carlos Eduardo Scheidegger, Karen Schuchardt, Margo I. Seltzer, Yogesh L. Simmhan, Cláudio T. Silva, Peter Slaughter, Eric G. Stephan, Robert Stevens, Daniele Turi, Huy T. Vo, Michael Wilde, Jun Zhao, and Yong Zhao. Special Issue: The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*, 20(5):409–418, 2008.

[133] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *Proceedings of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ASPLOS '09, pages 265–276, New York, NY, USA, March 2009. ACM.

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

[135] Toan Nguyen, Jean-Antoine Desideri, and Laurentiu Trifan. A Fault-Tolerant Approach to Distributed Applications. In *Parallel and Distributed Processing Techniques and Applications (PDPTA'13)*, Las Vegas, United States, July 2013. This work is supported by the European Commission FP7 Cooperation Program "Transport (incl.aeronautics)", for the GRAIN Coordination and Support Action ("Greener Aeronautics International Networking"), grant ACS0-GA-2010-266184. It is also supported by the French National Research Agency ANR (Agence Nationale de la Recherche) for the OMD2 project (Optimisation Multi-Disciplines Distribuée), grant ANR-08-COSI-007, program COSINUS (Conception et Simulation).

[136] Lucas Nussbaum and Olivier Richard. Lightweight Emulation to Study Peer-to-Peer Systems. In *3rd International Workshop on Hot Topics in Peer-to-Peer Systems (Hot-P2P)*, Rhodes Island, Greece, 4 2006.

[137] Lucas Nussbaum and Olivier Richard. A Comparative Study of Network Link Emulators. In *Communications and Networking Simulation Symposium (CNS)*, San Diego, United States, 2009.

[138] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Tim Carver, Matthew R. Pocock, and Anil Wipat. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:3045–3054, 2004.

[139] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 43–43, Berkeley, CA, USA, 1999. USENIX Association.

[140] Object Management Group (OMG). Business Process Model and Notation (BPMN) Version 2.0. Technical report, Object Management Group (OMG), jan 2011.

[141] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, 2014. USENIX Association.

[142] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0*, April 2007.

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

[144] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

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

[146] Larry Peterson, Andy Bavier, and Sapan Bhatia. VICCI: A programmable cloud-computing research testbed. Technical report, Technical Report TR-912-11, Princeton Univ., Dept. Comp. Sci, 2011.

[147] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers.

[148] Ben Pfaff, Justin Pettit, Teemu Koponen, Keith Amidon, Martin Casado, and Scott Shenker. Extending networking into the virtualization layer. In *8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, 2009.

[149] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. Failure Trends in a Large Disk Drive Population. In *5th USENIX Conference on File and Storage Technologies (FAST 2007)*, pages 17–29, 2007.

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

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

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

[153] Alina Quereilhac, Damien Saucez, Priya Mahadevan, Thierry Turletti, and Walid Dabbous. Demonstrating a Unified ICN Development and Evaluation Framework. In *Proceedings of the 1st International Conference on Information-centric Networking*, INC '14, pages 195–196, New York, NY, USA, 2014. ACM.

[154] Alina Quereilhac, Damien Saucez, Thierry Turletti, and Walid Dabbous. Automating ns-3 Experimentation in Multi-Host Scenarios. In *WNS3 2015*, Barcelona, Spain, May 2015.

[155] Martin Quinson, Cristian Rosa, and Christophe Thiery. Parallel Simulation of Peer-to-Peer Systems. In *CCGrid 2012 – The 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Ottawa, Canada, May 2012. RR-7653 RR-7653.

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

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

[158] Robert Ricci, Gary Wong, Leigh Stoller, Kirk Webb, Jonathon Duerig, Keith Downie, and Mike Hibler. Apt: A Platform for Repeatable Research in Computer Science. *ACM SIGOPS Operating Systems Review*, 49(1), January 2015.

[159] Luigi Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *SIGCOMM Computer Communication Review*, 27:31–41, January 1997.

[160] Raúl Rojas. How to make Zuse's Z3 a universal computer. *Annals of the History of Computing, IEEE*, 20(3):51–54, 1998.

[161] Tiago Pais Pitta De Lacerda Ruivo, Gerard Bernabeu Altayo, Gabriele Garzoglio, Steven Timm, Hyunwoo Kim, Seo-Young Noh, and Ioan Raicu. Exploring Infiniband Hardware Virtualization in OpenNebula towards Efficient High-Performance Computing. In *CCGRID Scale Challenge*, pages 943–948, 2014.

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

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

[164] Idafen Santana-Perez, Rafael Ferreira da Silva, Mats Rynge, Ewa Deelman, MaríaS. Pérez-Hernández, and Oscar Corcho. A Semantic-Based Approach to Attain Reproducibility of Computational Environments in Scientific Workflows: A Case Study. In Luís Lopes, Julius Žilinskas, Alexandru Costan, RobertoG. Cascella, Gabor Kecskemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, StephenL. Scott, Stefan Lankes, Christian Lengauer, Jesus Carretero, Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops*, volume 8805 of *Lecture Notes in Computer Science*, pages 452–463. Springer International Publishing, 2014.

[165] Luc Sarzyniec, Tomasz Buchert, Emmanuel Jeanvoine, and Lucas Nussbaum. Design and Evaluation of a Virtual Experimental Environment for Distributed Systems. In *PDP2013 - 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, pages 172 – 179, Belfast, United Kingdom, February 2013. IEEE.

[166] Eric Schulte, Dan Davison, Thomas Dye, and Carsten Dominik. A Multi-Language Computing Environment for Literate Programming and Reproducible Research. *Journal of Statistical Software*, 46(3):1–24, 1 2012.

[167] Hendrik Schulze and Klaus Mochalski. Internet study 2008/2009. http://www.ipoque.com/sites/default/files/mediafiles/documents/internet-study-2008-2009.pdf, February 2009.

[168] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. Making scientific computations reproducible. *Computing in Science and Engineering*, 2(6):61–67, November 2000.

[169] Dennis Schwerdel, Daniel Günther, Robert Henjes, Bernd Reuther, and Paul Müller. German-lab Experimental Facility. In *Proceedings of the Third Future Internet Conference on Future Internet*, FIS'10, pages 1–10, Berlin, Heidelberg, 2010. Springer-Verlag.

[170] Dennis Schwerdel, Bernd Reuther, Thomas Zinner, Paul Müller, and Phuoc Tran-Gia. Future Internet Research and Experimentation: The G-Lab Approach. *Computer Networks*, 2014.

[171] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

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

[173] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, September 2005.

[174] Jeremy Singer. A literate experimentation manifesto. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*, ONWARD '11, pages 91–102, New York, NY, USA, 2011. ACM.

[175] Standard Performance Evaluation Corporation.

[176] Diomidis Spinellis. A Repository with 44 Years of Unix Evolution. In *MSR '15: Proceedings of the 12th Working Conference on Mining Software Repositories*, pages 13–16. IEEE, 2015. Best Data Showcase Award.

[177] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using PlanetLab for network research: myths, realities, and best practices. *SIGCOMM Computer Communication Review*, 40:17–24, January 2006.

[178] Luka Stanisic, Arnaud Legrand, and Vincent Danjean. An Effective Git And Org-Mode Based Workflow For Reproducible Research. *SIGOPS Oper. Syst. Rev.*, 49(1):61–70, January 2015.

[179] James P.G. Sterbenz, David Hutchison, Paul Müller, and Chip Elliott. Special issue on Future Internet Testbeds. *Computer Networks*, 63(0):1 – 4, 2014.

[180] Domenico Talia. Workflow systems for science: Concepts and tools. *ISRN Software Engineering*, 2013, 2013.

[181] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. In

Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT '13, pages 217–228, New York, NY, USA, 2013. ACM.

[182] Matti Tedre and Nella Moisseinen. Experiments in computing: A survey. *The Scientific World Journal*, 2014, 2014.

[183] Stéphane Thiell, Aurélien Degrémont, Henri Doreau, and Aurélien Cedeyn. ClusterShell, a scalable execution framework for parallel tasks. In *Linux Symposium*, page 77, 2012.

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

[185] TOP500.Org. TOP500 Supercomputing Sites.

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

[187] Laurentiu Trifan. Resiliency in Distributed Workflows. Rapport de recherche RR-7435, INRIA, October 2010.

[188] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *SIGCOMM Computer Communication Review*, 36:271–284, December 2002.

[189] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distrib. Parallel Databases*, 14(1):5–51, July 2003.

[190] Wil M. P. Van Der Aalst, Arthur H. M. Ter Hofstede, and Mathias Weske. Business Process Management: A Survey. In *Proceedings of the 2003 International Conference on Business Process Management*, BPM'03, pages 1–12, Berlin, Heidelberg, 2003. Springer-Verlag.

[191] B. Videau and O. Richard. Expo : un moteur de conduite d'expériences pour plates-forme dédiées. In *Conférence Française en Systèmes d'Exploitation (CFSE)*, 2008.

[192] Brice Videau, Corinne Touati, and Olivier Richard. Toward an experiment engine for lightweight grids. In *Proceedings of the First International Conference on Networks for Grid Applications (GridNets 2007)*, GridNets '07, pages 22:1–22:8, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[193] Jacques Wainer, Claudia G. Novoa Barsottini, Danilo Lacerda, and Leandro Rodrigues Magalhães de Marco. Empirical evaluation in Computer Science research published by ACM. *Inf. Softw. Technol.*, 51(6):1081–1085, June 2009.

[194] L. Wang and J. Kangasharju. LiteLab: Efficient Large-scale Network Experiments. *ArXiv*, Nov 2013.

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

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

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

[198] Tim Wauters, Brecht Vermeulen, Wim Vandenberghe, Piet Demeester, Steve Taylor, Loïc Baron, Mikhail Smirnov, Yahya Al-Hazmi, Alexander Willner, Mark Sawyer, David Margery, Thierry Rakotoarivelo, F Lobillo Vilela, Donatos Stavropoulos, Chrysa Papagianni, Frederic Francois, Carlos Bermudo, Anastasius Gavras, Dai Davies, Jorge Lanza, and Sueng-Yong Park. Federation of Internet experimentation facilities: architecture and implementation. In *European Conference on Networks and Communications (EuCNC 2014)*, Bologne, Italy, June 2014.

[199] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. SliceTime: A Platform for Scalable and Accurate Network Emulation. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 253–266, Berkeley, CA, USA, 2011. USENIX Association.

[200] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. 978-3-540-73521-2. Springer, 2007.

[201] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. Hassan Zahraee, and H. Karl. MaxiNet: Distributed Emulation of Software-Defined Networks. In *IFIP Networking 2014 Conference*, 2014.

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

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

[204] Michael Wilde, Mihael Hategan, Justin M Wozniak, Ben Clifford, Daniel S Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.

[205] Niklaus Wirth. Program development by stepwise refinement. *Communications ACM*, 14(4):221–227, April 1971.

[206] Katherine Wolstencroft, Robert Haines, Donal Fellows, Alan Williams, David Withers, Stuart Owen, Stian Soiland-Reyes, Ian Dunlop, Aleksandra Nenadic,

Paul Fisher, Jiten Bhagat, Khalid Belhajjame, Finn Bacall, Alex Hardisty, Abraham Nieva de la Hidalga, Maria P. Balcazar Vargas, Shoaib Sufi, and Carole Goble. The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, 41(W1):W557–W561, 2013.

[207] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*, pages 13–24. ACM, 2013.

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

[209] Yuhao Zheng and David M. Nicol. A Virtual Time System for OpenVZ-Based Network Emulations. In *Proceedings of the 25th IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*, PADS '11, pages 1–10, Washington, DC, USA, 2011. IEEE Computer Society.

COLOPHON

This document was typeset using the `classicthesis` package developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography *"The Elements of Typographic Style"* and is available for download here:

http://code.google.com/p/classicthesis/

The fonts used in the document are: *Alegreya Regular* and its variants as a serif font for the main text body, *Alegreya SC* as a small caps font in section headings and titles, *Inconsolata* as a mono-spaced font, and *TeX Gyre Pagella Math* as a math font. All these fonts are freely available.

Final Version as of February 14, 2016 (`classicthesis`).

Abstract (English) Running experiments on modern systems such as supercomputers, cloud infrastructures or P2P networks became very complex, both technically and methodologically. It proved difficult to run experiments correctly and understand obtained results, even with the background on the employed technology and methods. Moreover, large-scale experiments suffer from erroneous and the unpredictable behavior of underlying software and hardware, undermining the scientific principles of experimental computer science. This worrisome state of research on large-scale distributed systems calls for new approaches to design, run and interpret experiments.

This work explores the use of control-flows (business processes) as a model for representing the large-scale experiments in research on distributed systems. We set out to find advantages, disadvantages and limitations of this approach, and practical considerations for future implementers.

We make 3 main contributions. First, we analyze the current state of experiment management tools, their limits and features to better understand difficulties that lay ahead. We construct a general framework to evaluate tools of this type. Second, we design and implement an experiment management tool which is based on the model of control-flows. We show that this methodology can be implemented and used in practice to run challenging and large-scale experiments while offering a wide set of features, some of them missing in the previous approaches. Finally, we analyze the use of provenance in computer science, and in particular in experimental research on distributed systems, and propose a provenance collection system that emerges from the control-flow model used as the representation of experiments. The design is implemented and shown to collect provenance in efficient and automatic way.

Our results show that workflows are a viable model for the design and execution of experiments in distributed systems research. With these positive conclusions in mind, we also sketch future research directions for improving our work.

Abstract (French) L'expérimentation sur les systèmes modernes comme les superordinateurs, les infrastructures cloud ou les réseaux P2P, est devenue complexe à cause des difficultés techniques et méthodologiques. La réalisation correcte d'expériences et l'analyse des résultats obtenus est difficile, même en possédant toute l'expertise nécessaire sur le domaine d'étude et la technologie utilisée. De plus, les expériences à grande échelle échouent souvent en raison du comportements aléatoires du matériel et du logiciel, menaçant les principes de la recherche expérimentale comme la fiabilité et la reproductibilité des résultats. Cette situation inquiétante de la recherche sur les systèmes distribués à grande échelle nécessite la découverte de nouvelles approches pour la structuration, le contrôle et l'interprétation d'expériences.

Ce travail explore l'utilisation de control-flows (processus métier) comme un modèle pour la représentation d'expériences à grande échelle dans le domaine des systèmes distribués. Il analyse les avantages, inconvénients et limitations de cette approche, ainsi que des considérations pratiques pour leur implantation future.

Trois contributions principales peuvent être distinguées. D'abord, nous analysons l'état actuel des outils pour le contrôle d'expériences. Nous montrons les fonctionnalités manquantes et permettons de comprendre les difficultés partagées par toutes les approches. Cette analyse se termine avec la construction d'une hiérarchie des propriétés qui peut être utilisée pour l'évaluation des outils qui contrôlent les expériences. La deuxième contribution consiste en un design et une implantation d'un système de contrôle d'expériences qui se base sur le modèle de control-flows. Nous montrons que cette méthodologie est capable du contrôle efficace et robuste des expériences à grande échelle et offre des fonctionnalités nécessaires, dont certains ne sont pas présentes dans les approches existantes. La dernière contribution porte sur la conception et l'implantation d'un système pour la collection de provenance pendant l'exécution d'expériences sur les systèmes distribués. Elle utilise intensément le modèle de control-flows et améliore l'approche présentée précédemment. Le prototype de ce système est capable d'une collection de provenance de manière efficace et automatique.

Les résultats obtenus montrent que le modèle proposé est une approche viable du contrôle d'expériences dans les systèmes distribués. De plus, les améliorations possibles sont mentionnées à la fin du document.