A generic approach to network experiment automation

Alina Quereilhac

UNIVERSITE NICE SOPHIA ANTIPOLIS
ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour l'obtention du grade de
Docteur en Sciences
de l'Université Nice Sophia Antipolis

Mention : Informatique

présentée et soutenue par
Alina L. QUEREILHAC

A Generic Approach to Network Experiment Automation

Une Approche Générique pour l'Automatisation des Expériences sur les Réseaux Informatiques

Thése dirigée par Walid DABBOUS et Thierry TURLETTI
et préparée au sein du laboratoire INRIA, équipe DIANA
soutenue le 22 Juin 2015

Jury :

M. Walid DABBOUS	INRIA, Sophia Antipolis	Directeur
M. Thierry TURLETTI	INRIA, Sophia Antipolis	Codirecteur
M. Roberto CANONICO	UNINA, Naples	Rapporteur
M. Serge FDIDA	UPMC, Paris	Rapporteur
M. Thomas HENDERSON	University of Washington, Seattle, WA	Rapporteur
M. Luigi IANNONE	Telecom ParisTech, Paris	Examinateur
M. Thierry RAKOTOARIVELO	NICTA, Sydney	Examinateur

A Generic Approach to Network Experiment Automation

Abstract

Many network evaluation platforms are commonly used to empirically study and develop networking technologies through experimentation using simulated, emulated, and live evaluation conditions. In most cases, running experiments requires considerable manual work, which leads to human errors and makes studies difficult to reproduce. The lack of abstraction in the experimentation process and the lack of uniformity of interfaces and tools across platforms makes it hard to standardize best practises for rigorous experimentation and to use different platforms for result cross-validation. A solution to address both platform usability and experimentation rigour lies in the automation of the experiment life cycle. Automation minimizes human intervention and relies on well-defined experiment descriptions and reproducible orchestration mechanisms. Existing network experiment automation frameworks target specific platforms or networking research domains, making it difficult to adopt them as general solutions or to use them to combine different platforms in a same study.

This thesis proposes a generic approach to automate network experiments for arbitrary evaluation platforms, and for scenarios targeting any networking research domain. The proposed approach is based on abstracting the experiment life cycle for different platforms into generic steps that are valid for simulators, emulators, and testbeds. Based on these steps, a generic experimentation architecture is proposed, composed of an experiment model, an experimentation interface, and an orchestration algorithm. The feasibility of this approach is demonstrated through the implementation of a framework capable of automating experiments in simulators, emulators, testbeds, or combinations of them. Three main aspects of the framework are evaluated: Its extensibility to support heterogeneous platforms, its efficiency to orchestrate experiments, and its flexibility to support diverse use cases for different networking research domains, including education, platform management, and experimentation with testbed federations, and cross-platform and multi-platform scenarios. The results show that the proposed approach can be used to efficiently automate experimentation on heterogeneous evaluation platforms, for a wide range of scenarios.

Keywords

Network experiments, automation, reproducibility, simulation, emulation, testbeds, testbed federations.

Une Approche Générique pour l'Automatisation des Expériences sur les Réseaux Informatiques

Résumé

Plusieurs plates-formes d'évaluation, tels que des simulateurs, des émulateurs et des bancs d'essai, sont couramment utilisées pour développer les technologies réseaux et les étudier empiriquement. Dans la plupart des cas, l'exécution de ces expériences nécessite un travail manuel considérable, ce qui entraîne de nombreuses erreurs et bogues et rend des études difficiles à reproduire. D'autant plus que le manque d'abstraction dans le processus d'expérimentation et l'absence d'uniformité des interfaces et des outils entre les plates-formes réseaux rendent la standardisation des pratiques d'expérimentation rigoureuse difficile. Ces aspects compliquent également l'utilisation des différentes plates-formes dans une même étude pour la validation croisée des résultats. L'automatisation du cycle de vie des expériences réseaux est une solution pour établir un processus d'expérimentation rigoureux tout en rendant les plates-formes plus facilement utilisables. En effet, l'automatisation minimise les interventions humaines en s'appuyant sur des descriptions d'expériences bien définies et vérifiables ainsi que des mécanismes d'orchestration reproductibles. Toutefois, les systèmes d'automatisation d'expérimentation réseau existants ciblent des plates-formes et des domaines de recherche réseau spécifiques, ce qui rend leur généralisation difficile sans pour autant autoriser l'utilisation de plates-formes différentes au sein d'une même étude.

La présent thèse propose une approche générique pour automatiser les expériences réseaux sur toute type de plates-formes réseaux, indépendamment de leur domaine d'application. L'approche proposée est basée sur l'abstraction du cycle de vie de l'expérience en étapes génériques qui sont valides pour des simulateurs, des émulateurs et des bancs d'essai. Une architecture d'expérimentation générique est proposée basée sur ces étapes, composée d'un modèle d'expérience abstrait, d'une interface d'expérimentation universelle, et d'un algorithme d'orchestration générique. Le faisabilité de cette approche est démontrée par la mise en œuvre d'un système capable d'automatiser les expériences dans des simulateurs, des émulateurs, des bancs d'essai, et même des combinaisons de ces plates-formes. Les trois aspects principaux du système sont évalués : son extensibilité pour s'adapter aux plates-formes hétérogènes, son efficacité pour orchestrer des expériences réseaux et sa flexibilité pour permettre des cas d'utilisation divers. Ceci inclus les usages à but d'enseignement mais aussi la gestion de plates-formes et la mise en scene des expériences complexes regroupant bancs d'essai et fédérant des plates-formes de natures différentes. Les résultats montrent que l'approche proposée peut être utilisée pour automatiser efficacement des expériences sur des plates-formes hétérogènes, pour un large éventail de scénarios.

Mots clés

Expériences réseaux, automatisation, reproductibilité, simulation, émulation, bancs d'essai, fédération des bancs d'essai.

Contents

Chapter 1

Introduction

An important role of science in society is to produce knowledge and technological innovation for the improvement of life conditions, such as improving life expectancy, producing non polluting energy, and fostering economical growth and cultural development. In particular, innovation in computer networks has helped to interconnect people and to create new markets, and has changed to a large extent the way people relate to technology, making ubiquitous communications something as indispensable as tap water or electricity.

As major actors of the innovation process, researchers and engineers are faced with several challenges. In domains like computer networks, where new technologies are rapidly made obsolete, timing is an essential factor and ideas that are not realised rapidly might miss their market opportunity. In order to effectively transfer innovation from the laboratory to the public, networking researchers and engineers not only need to be time efficient, they also need to implement robust software that can work correctly in production environments. This requires thorough evaluation of initial ideas, as well as of the intermediate and final software implementations.

Due to the high complexity of modern networking systems, the use of purely analytical evaluation methods, even to study simple ideas, is often unfeasible. Empirical evaluation methods are of great importance in applied computer sciences [1, 2] and provide an alternative to overcome the limitations of analytical methods. Networking researchers and engineers make extensive use of empirical evaluation platforms, such as simulators, emulators, and testbeds, to develop and evaluate networking solutions. However, manipulating these platforms can be time consuming and error prone, specially when many technologies and diverse factors must be taken into account, or when considerable manual work is required to conduct experiments.

Ensuring a rigorous experimentation methodology presents an additional challenge on top of the difficulties of platform usage. A rigorous methodology depends largely on the good practises adopted by individual experimenters. If a platform is hard to use and requires extensive manual work, more attention and effort are necessary to avoid errors and to ensure a controlled and reproducible experimentation process. One way to achieve better control and reproducibility in the experimentation is by using automation.

Automating the technical parts of experiments, such as experiment deployment and result collection, can help to save time and to prevent human errors by reducing manual intervention. Automation also improves reproducibility by depending on well defined experiment descriptions and on re-usable experimentation procedures.

In the last decade, different solutions have been proposed in the area of computer networks to simplify experimentation, to enforce rigorous experimentation, and to address the construction of complex experiment scenarios in the form of tools [3, 4], frameworks [5, 6, 7], and testbed federations [8, 9]. However, existing solutions usually support only a subset of evaluation platforms or target specific networking research domains.

The goal of this thesis is three fold. First, to propose a generic solution to automate experimentation on arbitrary networking evaluation platforms and research domains. Second, to simplify the usage of multiple platforms in a same study. Third, to provide support for rigorous experimentation across platforms.

1.1 Terminology

Platform and Environment. The terms platform and environment are used interchangeably to refer to a software or infrastructure used to conduct network experiments. Platforms can be of different types and for the purpose of this work they are divided into simulators, emulators, testbeds, and testbed federations.

Evaluation Method. Evaluation method refers to the technique used to conduct an experiment. This thesis considers three evaluations methods: simulation, emulation, and live experimentation.

Repeatibility, Reproducibility, and Replicability. In this thesis, the terms *repeatability*, *reproducibility*, and *replicability* are used to refer to different concepts. All three refer to the action of *cloning* an aspect of an experiment. Repeatability is used to indicate cloning experiment results, reproducibility is used to indicate cloning the procedure used to conduct an experiment, and replicability is used to indicate cloning the experiment scenario on a different platform. Not all platforms support repeatability, only those that provide controlled conditions that permit to obtain the same results when reproducing an experiment. All platforms should support reproducibility, this is the ability to re-play the steps of an experiment in order to obtain statistically relevant results. Replicating a scenario on different platforms requires similarities between the platforms in order for the results to be comparable.

Resource. A resource is any component modeled as part of an experiment that an experimenter can manipulate or obtain data from. Following this definition, a host in a testbed, an application running on a host, and an entry on a routing table are all examples of experiment resources.

Host and Node. The terms host and node are used interchangeably to refer to either physical, emulated, or simulated hosts in an experiment.

1.2 A Plethora of Evaluation Platforms

The growing variety of networking technologies and evaluation scenarios has led to a proliferation of evaluation platforms tailored for different needs. Many platforms focus on specific technologies or research domains, such as Wifi networks [10, 5], or Internet services [11, 12], or use a particular evaluation method, such as simulation [13, 14], emulation [15, 16, 17], or live experimentation [18, 19, 20]. Platforms can range from general-purpose, multiple-domain and multi-method, supporting different evaluation methods for various research domains [7, 21], to single-purpose, specific-domain and specific-method, supporting a single method for a specific research domain [22, 23].

Experimenters use these platforms in various ways. They can conduct a study with only one platform, they can independently replicate their study on multiple platforms [24, 25] to obtain complementary results, or they can interconnect platforms together to build cross-platform scenarios [26].

The diversity of platforms in the networking ecosystem attests for the fact that no single-platform, or evaluation method, has been capable of satisfying all experimentation needs. As networking technologies evolve, it is likely that new scenarios will continue to encourage the creation of new platforms and the use of various evaluation methods. This section gives an overview of the different types of evaluation platforms commonly employed in the networking research community, and their usages.

1.2.1 Platform Types

1.2.1.1 Simulators

Simulators imitate the behavior of real systems using models to simplify the study of the interactions between system components. They provide controlled and repeatable evaluation conditions. However, since simulators simplify reality, in some cases they might produce biased results [27, 28]. Their biggest strength is that they make possible the study of scenarios otherwise prohibitive or too costly on more realistic platforms, such as emulators and testbeds.

Simulators can be general purpose or domain specific. Examples of general purpose network simulators are ns-2 [29], ns-3 [13], and OMNET++ [14]. Examples of domain specific simulators are SENSE [30], for wireless sensor networks, OverSim [23], for overlay networks, and ndnSim [22], for named data networking.

Some of the benefits of simulators compared to emulators and testbeds are the following:

- Running and debugging simulations, if not physically distributed, tends to be easier than running and debugging live distributed experiments.
- Simulators can support larger scale experiments and a larger variety of technology models than live environments.

- Simulators allow to evaluate scenarios that are prohibitive in live environments due to the high cost of the live infrastructure, to the impossibility of controlling live traffic, or to usage restrictions, e.g., simulating a large scale attack.
- Simulators provide highly controllable environments that support perfect experiment repeatability.

The main drawback of simulators is their limited level of realism. They are not useful to capture unpredictable behaviors, which is needed for instance when validating production software. Discrete event simulators can also be extremely slow if too many events are generated, making simulation impractical for scenarios where large networks generate large amounts of traffic and the simulation granularity is on a per-packet level.

1.2.1.2 Emulators

Emulators combine system models with real system components, and emulated experiments have both synthetic-controllable and live-realistic aspects. Emulators can be seen as providing realism and controllability at three different levels of a network: the device level, the protocol level, and the application level. According to these levels, they can range between two extreme cases: hardware based and software based.

Software based emulators, such as DCE [15], model the behavior of the physical network and support execution of real applications and protocols. DCE is capable of executing unmodified applications inside an ns-3 simulation [13]. Hardware based emulators, or emulation testbeds, such as Flexlab [17], use real network devices, while providing host virtualization and synthetic link and traffic conditions. Flexlab [17] uses realistic Internet conditions measured from PlanetLab [11] to shape traffic between hosts in the Emulab testbed.

Other examples of emulators are Mininet [16] for OpenFlow emulation, WISER [31] for MANET protocol emulation, SUNSHINE [32] for sensor network emulation, and ModelNet [33] for scalable Internet emulation.

The benefits and drawbacks of emulation, compared to simulation and live experimentation, depend on the characteristics of the emulator. The main factors to consider is how well they scale, support controllable and reproducible experimentation, and what degree of realism they offer for different aspects of an experiment.

1.2.1.3 Testbeds

Testbeds are dedicated network infrastructures used to conduct experiments on real systems. In a testbed, the devices, protocols, and applications are real. Live experimentation allows to study a system without simplifications, exposing complex interactions between network components that could pass unnoticed on simulators or emulators.

Testbeds usually use specialized management frameworks [34] to control resources and provide services to the users. Testbed services provide resource control, experiment control, and data control. Resource control services include account registration,

authentication, resource discovery, resource reservation, and resource provisioning. Experiment control includes experiment deployment, execution, and monitoring, and data control includes instrumentation and data collection.

Examples of testbeds and their management services are: PlanetLab [11] Central and Europe testbeds for Internet experimentation using MyPLC for resource control, w-iLab.t [35], NitLab [36], and NORBIT [37] for wireless experimentation using OMF [6] for experiment orchestration, Grid5000 [18] for grid software evaluation using OAR [38] for resource reservation, and SensLab [39] for sensor networks using the SensLab management software.

The main advantage of testbeds compared to simulators and emulators is their high level of realism. However, testbeds usually target a specific networking research domain, so in order to achieve actual realism in the results of an experiment the characteristics of the testbed must match the requirements of the scenario. For instance, a grid environment will not accurately reflect Internet conditions and an Internet testbed will not be appropriate for realistic grid computing evaluation.

Compared to simulators, testbeds present the following drawbacks:

- Deploying experiments in a testbed is time consuming since it involves accessing and synchronizing distributed components.
- Testbeds provide limited scalability in terms of network size and a limited technology variety, since physical resources are scarce and expensive.
- Testbeds might impose usage restrictions. Resources are usually shared among many users, limiting their availability.
- Testbeds are not fully controllable or predictable environments, making it difficult to achieve experiment repeatability.

1.2.1.4 Testbed Federations

Testbed federations have the objective of transparently sharing resources across different administrative domains, i.e., independently managed testbeds. This transparent integration of testbeds gives experimenters access to a larger number of resources and to a larger variety of technologies with a same set of authentication credentials. Federations provide uniform services across testbeds, showing uniform interfaces for resource control, experiment control, and data control.

There are multiple testbed federation projects featuring different degrees of testbed integration, offering different services, and targeting different research domains. Homogeneous federations, like the PlanetLab Central and PlanetLab Europe federation [40], require all testbeds to use a same management framework. Heterogeneous federations, like the PlanetLab-Emulab federation [41], are capable of integrating testbeds that use different management frameworks.

Some federations use a centralized management approach, like in the case of the PlanetLab-Emulab federation, which provides a single point of access to browse and reserve resources across the federation. In contrast, other federations propose a decentralized management approach using distributed algorithms and services to browse and

allocate resources across testbeds. This is the case of the GENI [9] federation in the US and the FIRE [42] federations in Europe.

Certain federations focus on a specific research domain, like OFELIA [43] for OpenFlow research, BonFire [44] for Cloud service research, CREW [45] for cognitive radio research, and WISEBED [46] for wireless sensor research, whereas other federations, such as TEFIS [47], Panlab [48], Openlab [49], and Fed4FIRE [8], support a broad scope of domains.

Testbed federations solve several problems of isolated testbeds, like network size scalability and technology diversity. However, they do not address other limitations of testbeds such as the lack of experiment controllability and repeatability.

1.2.2 Platform Uses

1.2.2.1 Single-Platform Evaluation

Single-platform evaluation consist in conducting a network study using only one platform. An advantage of using a single platform is that no extra time needs to be invested in learning how to use additional platforms or to write multiple scripts and programs to run experiments on different platforms. However, using only one platform limits the scenarios that can be evaluated and might provide only a partial perspective on the problem under study. For instance, using only an Ethernet testbed to study a transport layer protocol will not permit to evaluate the impact of wireless channel conditions.

1.2.2.2 Multi-Platform Evaluation

Multi-platform evaluation consists in the independent use of complementary platforms to evaluate a same scenario. For example, using two testbeds with different physical network technologies, or using a simulation and an emulation platform independently.

Some experimentation platforms, like JiST/MobNet [50] and Netbed [7], natively support replicating an experiment using simulated, emulated, and live conditions. Federations also provide uniform access to different testbeds, making it easier to replicate a same scenario on different infrastructures. Experimenters can replicate their experiments by manually adapting the steps they used to run the experiment to different platforms, although this is usually a costly process. Some tools, like BonnMotion [51], provide scenario translation across a group of platforms.

The main advantage of multi-platform evaluation is that it permits to validate results by gathering complementary data. A drawback is that using multiple platforms often requires more time and effort than using only one.

1.2.2.3 Cross-Platform Evaluation

Cross-platform evaluation is the simultaneous use of different platforms in a same experiment. It requires integrating platforms to exchange traffic or data between resources during experiment execution. Examples of cross-platform evaluation include interconnecting simulated, emulated, and live platforms to mix realistic and controlled

components in a same experiment, or interconnecting testbeds to run large scale experiments with many nodes. In this sense, testbed federations are natively capable of cross-platform evaluation if they allow to integrate independent testbeds in a single experiment.

Platforms like the CORE emulator [21] and the NSE/Emulab infrastructure [52] natively provide a way of interconnecting simulated, emulated, and live components.

The main advantage of cross-platform evaluation is enabling experimental scenarios that are difficult to construct using a single platform. For example, cross-platform evaluation permits to test a model for a future physical network layer using real traffic from the Internet, by connecting a simulator with a live network. A main drawback is that most platforms do not natively support interconnection to other platforms, and interconnecting them might required considerable effort.

1.3 The Cost of Experimentation

As seen in the previous section, the networking research community has access to a wide variety of evaluation platforms to conduct experiments, including simulators, emulators, testbeds, and testbed federations. Different platforms are accessed and used in different ways, through specific user interfaces, tools, and services. As an example, while a network simulator might be used by writing and executing a C++ program [13, 14], a testbed might require to first use tools to browse and allocate physical resources, then to manually configure and start scripts and programs, and finally to generate network measurements and gather data to analyse [11, 6].

The time and the effort required to conduct an experiment and collect data are tightly related to the user tools and services available on each platform. Services provide support to manage resources and automate experiment operations, such as host reservation or software installation, and user tools give experimenters a way of accessing those services. Some services can be externalized, for instance software installation or application execution can often be performed by external tools [4, 53], while others services like host reservation depend on platform internal mechanisms [10]. Since user tools and services vary greatly among platforms, the cost of conducting an experiment is platform specific. Using several platforms in a same study increases the complexity of the scenario and the time and effort needed to run the experiment, specially when no common tools can be used across platforms.

The cost of running experiments can be decomposed into the following factors:

Learning cost. The time spent to master platform interfaces, tools, and services at the minimum level required to run the experiment.

Design cost. The time spent to write the scripts and programs needed to deploy and execute the experiment.

Deployment cost. The time spent to allocate and configure resources for the experiment, for example, virtual machine creation, host reservation, software installation, etc.

Execution cost. The time spent to start scripts and programs, to synchronize resources, to generate measurements, and to monitor the experiment.

Data collection cost. The time spent to retrieve and archive data generated during the experimentation.

A platform-independent solution to automate experimentation can help to reduce these costs and decrease the time and effort needed to conduct networking studies for arbitrary platforms.

1.4 The Need for Rigorous Experimentation

Scientific knowledge is based on the verifiability of results and demonstrations. Verifiable results must be reproducible and must also permit to derive further predictions from the initial hypothesis, making it possible for science to build upon previous discoveries to advance the state of knowledge.

In research fields like computer networks, which rely heavily on empirical evaluation, result verifiability requires a rigorous experimentation methodology, which must take the following into account:

- *Result validation*
- *Experiment reproducibility*
- *Data archiving*

Result validation consists in verifying that experiment results are correct with respect to the assumptions made. In particular, this requires validating the behavior of the platform used for experimentation. In the case of simulators for example, model validation is necessary to assure that simulated experiments yield meaningful results [54]. Good practices regarding platform validation include documenting the behavior of the platform, for instance by documenting which cases, technologies, or standards it supports, and using benchmarking techniques for verification [5]. Results can be validated by reproducing an experiment many times, under different conditions or even using different platforms to gather complementary data.

Experiment reproducibility consists in re-playing [55] all the steps that were carried out originally, in order to produce additional data to corroborate or disproof the original results. Reproducibility is a property of the experimentation procedure. It requires the steps of the procedure to be re-playable, and for this the experiment configuration and procedure must be detailed and clearly documented. Experiment reproducibility is not the same as experiment repeatability, reproducibility is a necessary but not sufficient condition for repeatability. Repeating an experiment requires that the same, or close to the same, results be obtained when reproducing the experimentation steps. Repeatability is a property of the platform, which must provide a highly controllable experimentation environment. Simulators can support perfect repeatability, since they

are self-contained and highly controllable environments. In contrast, live experimentation platforms like PlanetLab do not support repeatability due to their unpredictable and non-controllable nature [56].

Data archiving consists in storing results, measurements, and experiment configurations in an orderly and well documented manner. Archived data permits other researchers to analyse the results of a study and to validate the conclusions of the original experimenters. Archiving is partly a property of the experimentation procedure, which must involve data storing and documentation, and partly a property of the platform, which must provide adequate instrumentation support. The difficulty in sharing experimental data and detailed instructions to reproduce experiments among the community damages the ability to verify results and to build upon those results to produce further innovation.

Concern with rigorous experimentation is not new to the networking community. Issues related to validity, archiving, reproducibility, and repeatability, have been discussed for many years. Different works have addressed these issues by suggesting model validation [24, 17], platform benchmarking [57, 58] and calibration [59], platform active monitoring [60, 61, 62], platform instrumentation [63, 64], availability of full experiment records [65], platform orchestration support [55, 66, 67, 68], synthetic experiment descriptions [6, 16], and experiment automation tools to help users in reproducing studies [69, 70, 71, 4].

Nevertheless, support for rigorous experimentation remains platform or domain specific, since most tools and services that help enforcing it are tied to specific platforms, instead of being implemented as generic tools or standards that are platform, domain, and infrastructure independent.

1.5 Network Experiment Automation: Overview on the State of the Art

Several works have so far addressed automation of network experiments by abstracting experiment representation and providing experiment orchestration services like automated topology generation, resource allocation, execution control, and data collection. Existing tools often respond to the needs of specific communities of experimenters, focusing on certain platforms or research domains. Some tools and frameworks focus on modeling the application level of distributed systems, others automate experimentation for individual platforms, and others are able to work across platforms that implement a same backend, e.g., a same management framework.

1.5.1 Experiment Automation for Distributed Systems

Tools that address experiment automation for distributed systems are generally focused on the description of the application level of experiments, and provide little flexibility to model or modify the protocol and physical characteristics of the network. In most cases they support automation on testbeds only.

Plush [72, 4], its descendent Gush [73], and Splay [3, 53] automate evaluation of distributed applications on testbeds such as PlanetLab, ModelNet, and Emulab. In the three cases, an experiment controller that communicates with agent processes on the testbed nodes is in charge of experiment automation. In the case of Plush and Gush, the controller runs on the user side, whereas in Splay the controller runs inside the testbed and receives instructions from the user through a Web interface or command-line tool. Plush can deploy and start the agent processes in the testbed nodes, in this sense Plush does not rely on a pre-installed backend in the testbed, other than a SSH daemon running in the hosts. Splay, on the contrary, requires the Splay backend to be pre-installed in all testbed hosts.

Plush and Gush use an XML specification file to describe experiments, whereas Splay uses a programmatic approach based on the Lua language. Gush and Splay support describing emulated resources, and Splay supports also describing simple emulated topologies as graphs. Plush supports the definition of application workflows using processes, barriers, and workflow blocks, whereas Splay supports workflows and incremental deployment of experiments using job scheduling and events.

ExCovery [74], focuses on dependability analysis of distributed processes. ExCovery uses an XML schema to represent experiments, describing processes to be executed, input factors, fault injections, and environment manipulation operations. The XML experiment description is passed to an experiment Master program, which interprets it and sends instructions to the client nodes inside the platform. The Master communicates with the nodes using XMLRPC calls through a dedicated communication channel. ExCovery imposes several requirements on the testbeds, such as full privileged access to all nodes.

Weevil [75] focuses on the evaluation of highly distributed systems that deal with services delivered across a large number of access points. It represents experimentation as a two-phase process: workload generation, and experiment deployment and execution. The workload is generated synthetically, by recording client service calls through off-line simulation, and replaying the calls sequence during the experiment run on a testbed. Experiment deployment and execution rely on a central-controller architecture in which a master script is in charge of the coordination of the system components, the deployment of scripts, and the execution of the workload. Weevil automatically generates the master, and other scripts, from a configuration file created by the user. Weevil requires access to the testbed through a user-level remote shell in order to automate experiment deployment, execution, and data collection.

Expo [76, 77] focuses on running distributed applications on grid architectures and distributed testbeds. Its main feature is the ability to efficiently distribute commands issued by the experimenter over heterogeneous distributed resources. The experimenter interacts with the testbed resources through an interactive interface, which communicates with the API testbed and with the Experiment Controller components. The API testbed wraps services provided by the testbeds and interacts with a Reservation System to allocate resource for the experiment. The Experiment Controller issues commands to resources and logs and stores information about the experiment. A Task abstraction permits to associate a command to a particular resource. Tasks can be executed

synchronously or asynchronously. Experiments are described using a Ruby scripts.

XFlow [78] is a workflow engine that executes experiments as control flows. Experiments are modeled as directed graphs composed of activities, and are described using a domain-specific language. Activities represent tasks to be executed on testbed resources, they can be aggregated at different levels forming a hierarchical structure, and arranged into composite workflows that can be re-used in different experiments. Experimenters can use the predefined activities provided by the XFlow core libraries, or implement new ones using a low-level programming language. A same experiment workflow can be adapted to different testbeds by replacing specific activities.

Execo [79] is a tool to automate the execution of parallel distributed operations on Unix hosts. Experiments are modeled in terms of local or remote processes using a Python API. Remote hosts are accessed through SSH and testbed specific services, like resource reservation, can be supported by building a layer on top of Execo. The execo_engine is the module responsible for experiment execution. It provides a basic experiment life cycle, and can be extended to define complex workflows. Among other features Execo supports parameter sweeping, file upload and retrieval, and result aggregation.

1.5.2 Experiment Automation for Specific Platforms

Various automation solutions have been developed to address the needs of user communities working with specific platforms, usually targeting specific research domains. These tools and frameworks are usually designed to take advantage of the lower level features of the target platforms. They are capable of specifying and configuring network topology and protocol aspects of an experiment, instead of being limited to only modeling the application level of an experiment.

WISEBED [80] is a domain specific framework to automate experimentation on wireless sensor testbeds. It implements the concept of virtual testbeds (VTBs), similar to the PlanetLab slice concept, in order to provide isolated access to groups of resources, potentially including simulated, emulated, and live elements. VTBs are defined using the WiseML XML schema and expose a uniform experimentation interface to the experimenters through a Web services interface (iWSN) installed in the testbed. After instantiating a VTB, experimenters can modify it on real time and execute commands by invoking VTB operations through a GUI or using a command-line controller.

SAFE [70] is an experiment automation environment for the ns-3 simulator. It supports parallel execution of simulations, automated data collection and storage, and data analysis and visualisation. Experimenters run batch simulations by submitting a C++ ns-3 program, and a configuration file describing the parameters to explore, to the SAFE infrastructure. SAFE then launches parallel instances of a same simulation using different parameters, in order to sweep parameter ranges in the problem space. The computation effort of running multiple instances of an ns-3 simulation is distributed among a pool of nodes controlled by a server. Each simulation runs until a termination detector stops it, and then the data is automatically collected back to the server for processing.

Emulab Workbench [65] automates execution of experiments in Emulab testbeds. Workbench experiments are described using the NS syntax, which divides the experiment definition into a static and a dynamic part. The static part describes the hosts, devices, and links, the program agents used to execute programs, and the topology configuration, whereas the dynamic part describes the runtime behavior of the experiment using events. Integration of ns-2 simulation elements with live hosts is supported by the NSE simulation backend [81, 52], and portions of the experiment description can be marked as simulated using a special block delimiter. During runtime, experiments are controlled and monitored by the testbed software using a distributed events system. Emulab Workbench also automates resource allocation and configuration based on the information provided in the experiment description.

1.5.3 Experiment Automation for Heterogeneous Testbeds

Automation of experiments on heterogeneous testbeds is done mostly through testbed federation. Several initiatives for testbed federation have evolved in the past years. Two of the most ambitious and large scale ones are GENI [9] in the US and Fed4FIRE [8] in Europe. Both integrate testbeds with different backends as well as other sub-federations, such as ProtoGENI [82] in GENI and Panlab [48] in Fed4FIRE.

Heterogeneous federations provide cross-testbed experiment automation tools to help experimenters to describe experiment topologies, discover and reserve resources, execute applications, and collect data. In order to achieve automation, these tools rely on common backends, i.e., control and management frameworks, that must be deployed in all testbeds in the federation. These backends allow to standardize experimentation services across testbeds, and provide a common interface for experimenters to use. An example of a backend is the Slice Federation Architecture (SFA) [83], adopted by both GENI and Fed4FIRE. SFA provides services to discover, reserve, and provision resources across testbeds in a distributed and testbed independent way.

Whereas Fed4FIRE follows an homogeneous approach and imposes a same backend on all testbeds, GENI accepts an heterogeneous approach where different federation clusters are allowed to adopt their own backend. In GENI, this gives place to sub-federation clusters with different backends, including ProtoGENI, Open Resource Control Architecture (ORCA) [84], cOntrol and Management Framework (OMF) [85], and PlanetLab backends. Conversely, the Fed4FIRE architecture is based on the strict adoption of the same backend by all testbeds and sub-federations. This backend combines SFA and the Federated Resource Control Protocol (FRCP) [85] to provide a comprehensive framework for federated experimentation. SFA is used to allocate resources for the experiment and FRCP is used to control the resources during experimentation.

SFA is added to Fed4FIRE testbeds by extending the SfaWrap [86] bundle in order to expose testbed specific resources using a SFA compliant interface. Resources exposed through the SfaWrap can be browsed and allocated using the MySlice portal [87], or any other user tool capable of executing SFA queries.

FRCP is based on the OMF testbed management framework. It defines a messaging protocol to control testbed resources and a publish/subscribe infrastructure to deliver

messages to target resources inside a testbed. An experiment controller, such as the OMF Experiment Controller (EC), can orchestrate experiments by translating an experiment description defined by the user into FRCP messages that are sent to the publish/subscribe server of the testbed. Upon reception of the messages, the server forwards them to the Resource Controller (RC) processes that manage individual testbed resources. The OMF EC supports describing experiments in terms of nodes, links, and applications, and supports the usage of events to define experiment runtime behavior.

Although not standardized in Fed4FIRE, PanLab offers an alternative to the SFA/FRCP backend. PanLab management services are coupled with the Teagle experiment automation tool. It supports the creation of a virtual testbed environment, through the VCT tool, as well as the generation of executable workflows, through its orchestration engine. Additionally, Teagle supports the creation of ns-3 simulation scripts from a graphical user interface [88], allowing to execute a same scenario both in testbed and in simulated mode.

GENI does not feature common tools to automate experiment execution or data collection, and each federation cluster relies on specific orchestration tools such as Gush and OMF EC. The PrimoGENI [89] project, derived from Emulab management software, distinguishes itself by providing an integrated development environment, called slingshot, to manage the complete experiment life cycle. Additionally, it provides simulation and emulation support. Experimenters can schedule commands on slingshot for execution on emulated hosts.

1.5.4 Comparison of Experiment Automation Tools

Table D.1 compares different network experiment automation tools and frameworks on the basis of their ability to support multiple research domains (Domain), to support different platform types, including simulators, emulators, and testbeds (Platform), to be independent of any framework pre-installed in the platform (Backend), to support the specification and control of runtime experiment behavior (Workflows), and to support interactive experiment execution (Interactive).

A complementary comparison of experiment management tools is available in the survey by Buchert et al. [93]. The comparison in Table D.1 focuses on the generality and adaptability of existing tools and frameworks to support experimentation on diverse platform types and domains. The list of existing tools and frameworks is extensive, and the objective of this comparison is not to be exhaustive but to give a general overview of different existing approaches.

In general, existing tools are either specialized in a certain domain or platform type, or depend on a backend that must be pre-installed in the target platform. Certain approaches were born of specific problematics or within specific communities, e.g., Distributed Systems, and their objective was never to be generalized to all research domains and platforms. Other approaches try to provide integral solutions that not only allow user-side experiment automation but that also take care of platform management and resource control through the use of a backend, e.g., testbed federations.

Tool	Domain	Platform	Backend	Workflows	Interactivity
Plush/Gush [72, 4, 73]	D/S	Testbed	SSH	✓	✓
Splay [3, 53]	D/S	Testbed	Splay	✓	✓
ExCovery [74]	D/S	Testbed	ExCovery	✓	
Weevil [75]	D/S	Testbed	Any	✓	
Expo [76, 77]	D/S	Testbed	Any	✓	✓
XPFlow [78]	D/S	Testbed/Emulator	Any	✓	
Execo [79]	D/S	Testbed	SSH	✓	✓
Wisebed [80]	Sensors	Testbed	Wisebed		✓
SAFE [70]	Any	Simulator	ns-3		
Workbench [65]	Any	Testbed/Simulator	Emulab/NSE	✓	✓
OMF EC [85, 6]	Any	Testbed	FRCP	✓	✓
PrimoGENI [89]	Any	Testbed	ProtoGENI	✓	✓
Teagle [88]	Any	Testbed/Simulator	Teagle/ns-3	✓	✓
NEPI 2.0 [90, 91, 92]	Any	Any	Any		

Table 1.1 – Comparison of tools and frameworks to automate network experiments based on the domain of research they support, the type of platform they can control, the backend restrictions, and their ability to support workflows and interactive experiment execution. In the values, 'D/S' stands for distributed systems and 'Any' for not restricted.

The backend dependent approach simplifies federation of testbeds and makes it easier to generalize experiment automation, since the same services and interfaces can be expected across platforms. Nevertheless, adopting a common backend might not be feasible for all platforms, such as sensor testbeds with lightweight devices, and might prevent the integration of independent local resources into the experiment, such as the experimenter's own desktop or server.

The work presented in this thesis extends and formalizes previous work done on an early version of the Network Experiment Programming Interface (NEPI) [90, 91, 92], NEPI version 2.0. NEPI was conceived from the beginning to be a generic tool for the automation of network experiments. Its approach is to decouple the experiment automation interface from the platform interface, making it adaptable to arbitrary platforms while offering a unified experiment automation interface to the user. Nevertheless, the original experimentation model proposed in NEPI was not flexible enough, and it did not support the definition and execution of user defined workflows, or the possibility to modify experiments during runtime. Runtime modification and interactive experimentation are features offered by many platforms, in particular emulators and testbeds. NEPI 2.0 could only model static experiment life cycles, and this limited the platforms it could support.

This work generalizes and simplifies the NEPI object model and user interface, proposing a generic life cycle that better adapts to testbeds and testbed federations, as well as simulators and emulators. It incorporates an orchestration engine capable of resolving orchestration dependencies between arbitrary resources, supporting user

defined workflows and interactive experimentation with runtime modifications of the experiment scenario. Additionally, this work focuses on supporting rigorous experimentation and enforcing experiment reproducibility, replication, documentation, and data archiving.

1.6 Research Hypothesis

The research question that motivates this work is whether it is possible to *define a generic solution to automate network experiments on arbitrary experimentation platforms and for arbitrary network experimentation scenarios*.

Automating network experiments means minimizing or eliminating human intervention in the experiment life cycle by providing mechanisms to perform the experimentation steps in place of the user. These automation mechanisms must be consistent across experiment runs and allow to reproduce the same experimentation procedure.

Compatible with arbitrary platforms means that the approach chosen to automate experimentation must be potentially adaptable to any experimentation platform, existing or future.

Arbitrary network experimentation scenarios means that the automation can not be limited to a subset of scenarios or networking domains, and that it must be flexible to capture potentially any scenario involving any research domain. In particular, it should be possible to combine platforms for multi-platform and cross-platform experimentation.

Following these considerations, the research hypothesis formulated in this thesis is the following:

It is possible to design and implement a technique to automate the life cycle of network experiments on any networking evaluation platform, or combination of them, in order to to evaluate any experiment scenario involving any networking research domain.

This hypothesis is motivated by the existence of experiment automation solutions that partially solve the problem of experiment automation on different network experimentation platforms, and the possibility of further generalizing experiment automation to arbitrary platforms based on the assumption of a common network experiment life cycle that is valid for all platforms.

1.7 Approach

The approach proposed in this thesis to automate network experimentation on arbitrary platforms does not attempt to replace existing automation solutions, but to integrate and complement them, and to foster re-usability and cooperation among existing experiment tools and platforms. This approach is based on the following principles:

Generic Experiment Life Cycle, Interface, and Model. Experiment automation is generalized to arbitrary platforms and resources by defining a generic experimentation interface and a generic experiment model. The generic experimentation interface is based on the operations derived from the steps of a generic experiment life cycle that is

valid for arbitrary platforms. The generic experiment model is based on abstracting the representation of arbitrary scenarios using three generic entities: Experiment, Resource, and Data.

Adaptation Through External Drivers. Supporting a same experiment automation interface across platforms, without imposing any requirements on the platforms themselves, is achieved by decoupling the experimentation interface exposed to the user from the experimentation interfaces of individual platforms. Individual platforms are adapted to the generic interface thanks to external adaptor modules, in the same way a network device is adapted to the socket API using a specific network device driver.

Resource State-Based Workflows. Supporting configurable runtime behavior for arbitrary resources is done by using a same state machine to describe the basic behavior of all resources and allowing users to define additional state transition constraints to model user-defined workflows. Like in Emulab [65], the proposed experiment description model is divided in two parts, a static part to describe the resources used in the experiment, and a dynamic part to model the state-based workflows.

Generic Orchestration Engine. The execution order of the orchestration steps for arbitrary resources is resolved by a generic orchestration engine component. This engine uses an on-line black-box scheduling algorithm to resolve the execution order of life cycle steps for all resources, taking into account dependencies between resources and user-defined workflows.

1.8 Contributions

This work makes the following contributions:

1. **The formalization and specification of a generic approach to automate network experiments for arbitrary platforms and scenarios.** The approach is designed to support automation of network experiments for different evaluation methods, including simulation, emulation, and live experimentation, for single-platform, cross-platform, and multi-platform scenarios. The following associated contributions are made:

    - **Generic Experiment Life Cycle**. The identification of a generic experiment life cycle that is suited for simulators, emulators, testbeds, and testbed federations.
    - **Generic Experiment Interface**. The specification of a set of operations that encompass all tasks needed to automate experimentation on simulators, emulators, testbeds, and testbed federation, and the specification of additional operations to offer experiment reproducibility, experiment replication, and data archiving and sharing.
    - **Generic Experiment Model**. The definition of a generic model to represent arbitrary experiment scenarios based on three simple abstractions: Experiment, Resource and Data, and supporting the definition of experiment workflows to specify experiment runtime behavior.

- **Generic Orchestration Engine**. The specification of an algorithm for experiment orchestration capable of resolving arbitrary dependencies between life cycle steps of all resources in an experiment. This algorithm does not require complete centralized knowledge of the dependency tree, and naturally supports the execution of arbitrary life cycle operations.

2. **An implementation of the generic automation approach as a programming framework.** The framework supports a high-level description of arbitrary experiment scenarios and is capable of automating orchestration for simulation, emulation, and live network experiments. It is implemented as an open source Python library that can be used and extended freely by anyone. The following associated contributions are made:

   - **Extensible ResourceManager Class Hierarchy**. An extensible class abstraction to model resources on arbitrary platforms.
   - **Generic Orchestration Engine Implementation**. An implementation of the orchestration algorithm, capable of resolving arbitrary dependencies between resources and dynamically responding to changes in the experiment scenario. The generic orchestration algorithm is implemented in the Scheduler module of the ExperimentController entity responsible for experiment coordination.
   - **Automated Result Collection and Archiving**. Support for data collection and archiving at runtime using a Collector Resource abstraction. The Collector Resource is an example of how the ResourceManager class hierarchy can be extended to provide arbitrary experimentation services.
   - **Error Detection and Failure Policies**. User customizable policies to define experiment failure conditions and automated detection of errors during experiment deployment and execution.
   - **Experiment Reproduction**. A mechanism to automate re-execution of experiments until a condition defined by the user is satisfied.
   - **Experiment Replication**. Support for automated topology generation to facilitate execution of a same experiment scenario on different platforms.
   - **Simulation Support**. Support for simulation using the ns-3 simulator.
   - **Emulation Support**. Support for software emulation using the DCE emulation extension for ns-3, and support for the NetNS emulator based on Linux namespace virtualization.
   - **Live Experimentation Support**. Support for live experimentation on Linux testbeds with SSH key authentication, on PlanetLab and OMF testbeds, and on the Fed4Fire testbed federation.

3. **Framework Evaluation** The evaluation of the proposed approach, carried out using the framework implementation, takes into account three fundamental aspects: extensibility, efficiency, and flexibility.

- **Extensibility**. An evaluation of the capability of the framework to be extended to arbitrary platforms, including simulators, emulators, and testbeds, is done based on the analysis of the development effort undertaken during the past two years to support platforms and resources by extending the ResourceManager class hierarchy.
- **Efficiency**. An evaluation of the orchestration efficiency of the framework, in terms of time and memory needed to run experiments, is carried out by means of three benchmarks. The benchmarks take into account different factors that might impact orchestration efficiency, such as the delay to execute tasks and the number of threads used to parallelize orchestration. The first benchmark evaluates the performance of the framework using idealized conditions, whereas the second and third repeat the evaluation for real platforms using the ns-3 simulator and the PlanetLab testbed, respectively.
- **Flexibility**. A demonstration of the ability of the framework to support diverse usages and scenarios is presented based on five use cases: *Platform Maintenance*, *Teaching Support*, *Federated Experimentation*, *Cross-Platform Experimentation*, and *Multi-Platform Experimentation*.

1.9 Thesis Outline

The remaining of this thesis is divided into the following six chapters:

Chapter 2: Generic Network Experiment Automation, defines a generic experiment life cycle for simulators, emulators, and testbeds. It also defines an architecture composed of a generic experimentation interface and a generic experiment model, and proposes a generic algorithm to orchestrate arbitrary experiments based on a black-box approach for dependency resolution.

Chapter 3: Automation Framework Implementation, describes an implementation of the architecture defined in Chapter 2, detailing all the features provided, in particular the extensible ResourceManager class hierarchy and the experiment orchestration internals.

Chapter 4: Framework Evaluation: Extensibility, evaluates the feasibility and the cost of supporting heterogeneous experimentation platforms using a single framework, based on the extensions done to the ResourceManager class hierarchy.

Chapter 5: Framework Evaluation: Efficiency, evaluates the performance of the orchestration algorithm through three benchmarks using a Dummy platform, the ns-3 platform, and the PlanetLab platform.

Chapter 6: Framework Evaluation: Flexibility, provides use case examples to demonstrate the ability of the framework to support a variety of scenarios involving different networking research domains.

Chapter 7: Conclusions, summarizes the outcome of this work, and discusses the benefits and limitations of the chosen approach, and the directions for possible extensions.

Chapter 2

Generic Network Experiment Automation

This chapter presents the approach proposed in this thesis to automate network experiments for arbitrary experimentation platforms and scenarios. Experiment automation permits to avoid manual tasks, reducing the time needed to run experiments and minimizing human errors. It also contributes to a rigorous experimentation methodology by simplifying experiment reproduction, replication, documentation, and data archiving, independently of the platforms used in the experimentation.

Experiment automation requires two things: a structured and detailed description of the scenario to run, and mechanisms to automatically translate the scenario description into a running experiment. These two requirements can be respectively satisfied by a description model, to represent the experiment scenario, and an orchestration model, to generate and execute operations based on the scenario description.

The approach adopted in this work consists in defining a generic experiment description model, based on simple experiment abstractions, and a generic orchestration mechanism, to automate deployment and execution of arbitrary experiments. The experiment model is paired with an experimentation interface that exposes common experiment operations derived from a generic experiment life cycle, and that can be adapted to all types of platform.

Platforms do not need to be adapted to the generic experimentation interface. On the contrary, the generic interface is adapted to the interfaces and services of specific platforms by implementing adaptor modules. These adaptor modules extend the common life cycle operations and the experiment model. Two things are achieved by adapting *to the* platforms instead of adapting *the* platforms. The first is more flexibility in the type of platforms and scenarios that can be supported. It is simpler to adapt to a platform than to convince platform owners to adapt their platforms, since they might be unable or unwilling to implement the necessary changes. The second is that externally adapting to the platforms empowers users. Users can create their own adapter modules, orchestration mechanisms, or external services, without the need to convince platform owners to implement the changes in the platform.

Other solutions [6, 68, 52, 89] have also addressed similar problems related to abstract experiment description, orchestration automation, and support for rigorous experimentation. However, existing proposals are limited to specific research domains or take a platform-centric perspective. A platform-centric perspective gives full control of automation strategies to the platform owners, and relies on agreement among them to converge to a common strategy.

The approach adopted in this work is user-centric. A user-centric approach gives users control over the experimentation models and orchestration mechanisms, and the possibility of extending or changing those mechanisms.

The rest of this chapter defines the architecture and specification of the proposed automation approach. It starts with the definition of a generic experiment life cycle, which sets the basis for the proposed automation architecture. Then it presents the main components of the architecture: a generic experiment model, a generic experimentation interface, and a generic orchestration engine.

2.1 A Generic Network Experiment Life Cycle

An experiment life cycle is defined by the sequence of steps that are taken to conduct an experiment from beginning to end. These steps might vary from one platform to another. This thesis is based on the hypothesis of the existence of a generic life cycle, composed of common steps, that can be used to model the experimentation process on arbitrary network evaluation platforms.

A mechanism capable of automating experimentation on arbitrary platforms must take into account the common operations that are needed to conduct experiments on any type of platform. These common operations can be derived from the steps of a generic experiment life cycle. In order to identify these generic steps, the life cycles of several platforms including simulators, emulators, testbeds, and testbed federations were studied, leading to the isolation of nine basic steps: a) *platform setup*, b) *experiment conception*, c) *experiment design*, d) *experiment deployment*, e) *experiment execution*, f) *experiment monitoring*, g) *data collection*, h) *experiment termination*, and i) *data processing*. Appendix A provides examples of studied simulators, emulators, testbeds, and testbed federations and their respective life cycles.

Table 2.1 gives the definition of the generic life cycle steps and Figure 2.1 shows the interactions between the generic life cycle steps. The life cycle steps are divided into *Execute*, *Measure*, and *Monitor* activities, and grouped into three categories: those that take part of the *Experiment run*, steps d) to i), those to take part in the *Experiment life cycle*, steps c) to i), and the rest, steps a) and b).

Step a), *platform set up*, is performed only once per platform and per experimenter, and since it is not repeated for every experiment it is not considered to be a part of the *experiment* life cycle. Step b), *experiment conception*, is also considered to be outside the life cycle because it relies on the creativity of the experimenters and for this reason it cannot be automated.

Step	Description
a) Platform setup	Actions required to initialize or gain access to an experimentation platform. Examples are account creation and software installation.
b) Experiment conception	Conception of an experiment scenario that meets the objectives of a study. This includes deciding which resources will be used in the experiment, in which order experiment events will occur, and what data will be collected.
c) Experiment design	Specification of the resources used in the experiment and their configuration, the actions and events that must occur during the experiment, and the data to be measured and collected. The specification must be compatible with the interfaces and services provided by the target experimentation platform. This includes writing scripts, programs, and configuration files.
d) Experiment deployment	Preparation of resources and environments to execute the experiment. This includes identifying, reserving, provisioning and configuring resources for the experiment.
e) Experiment execution	Activation of resources, execution of processes and events, and generation of data. This includes executing scripts and programs, starting, stopping, and pausing applications and services.
f) Experiment monitoring	Verification of liveliness and well functioning of experiment resources, and taking the necessary actions in case of a failure, e.g., interrupting the experiment.
g) Experiment termination	Release of resources used in the experiment. This includes terminating applications and services, destroying virtual machines, etc.
h) Data collection	Retrieval and archiving of data generated by experiment measurements, and other experiment related information, e.g., experiment definition and configuration.
i) Data processing	Manipulation of collected data to obtain meaningful information. This includes validating, cleaning, formatting, analysing, and plotting the collected data.

Table 2.1 – Definition of the generic experimentation steps that can be used to abstract the experiment life cycles of simulators, emulators, testbeds, and testbed federations.

Network experiment life cycles proposed in the literature are sometimes represented as a linear sequence of steps, possibly including the ability to reproduce experiments [34]. However, this is not always the case, and in reality experimenters can re-execute or parallelize certain steps. The life cycle proposed in Figure 2.1 captures this flexibility by considering the possibility of non linear life cycles where steps can occur in parallel or be re-executed. It incorporates the idea of interaction, reproduction, and iteration of groups of steps. The proposed generic life cycle takes into account the possibility of dynamically modifying the initial experiment design, by deploying new parts of an experiment at runtime, i.e., interaction. It also considers the possibility of running a same experiment several times, i.e., reproduction, and it includes support for iterating between execution and experiment conception steps, in order to adapt a scenario to new requirements or ideas, i.e., iteration. Platform specific experiment life cycles proposed in the literature [34, 9] can be mapped to the generic life cycle proposed in this section.

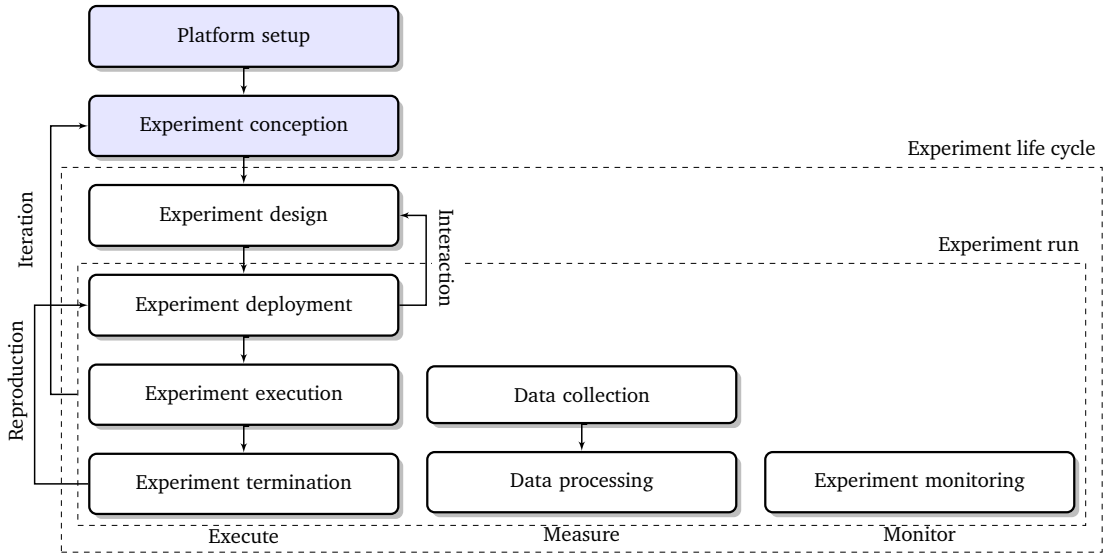


Figure 2.1 – Generic network experiment life cycle for simulators, emulators, testbeds, and testbed federations. The life cycle is composed of nine steps divided into Execute, Measure, and Monitor activities, and grouped into Experiment life cycle and Experiment run blocks.

2.2 An Architecture for Generic Experiment Automation

Based on the previous analysis of a generic experiment life cycle, this section proposes an architecture for the automation of network experiments that is independent of any specific experimentation platform or scenario.

2.2.1 Goals

The following goals were taken into account in the design of the architecture for generic experiment automation:

Simplicity. The architecture must be simple to use and easy to understand even by unexperienced experimenters. It should involve as few modeling elements as possible and provide a clear and intuitive user interface.

Extensibility. The solution should be extensible to support any networking platform or scenario, existing or future, without the need to adapt the platform.

Flexibility. There should not be restrictions to the types of experiment scenario that can be represented or to the level of detail that can be used to describe a scenario. It should be possible to represent experiments involving any networking technology, platform, and level complexity.

User-Centrality. The experimenter should be in control of the experimentation mechanisms, and should be able to extend or modify these mechanisms without depending on platform owners.

The goal of simplicity requires to avoid unnecessary complexity in the experiment modeling abstractions and in the experimentation interface. The experiment modeling abstractions and the operations in the experimentation interface must be intuitive and kept limited in number.

The goal of extensibility requires the experiment model and the experimentation interface to be adaptable to arbitrary resources on any experimentation platform.

The goal of flexibility requires the experiment model and the experimentation interface to support describing and running arbitrary scenarios.

The goal of user-centrality requires the architecture to be composed solely of user-side components, under full control of the user, instead of platform-side components, i.e., depending on a pre-installed backend.

2.2.2 Architecture Components

The proposed generic network experiment automation architecture consists of the following components:

Generic Network Experiment Model (GNEM). The generic network experiment model allows to represent arbitrary experiment scenarios in a uniform way across platforms, but exposing platform specific details. In order to do so, the experiment model must support at the same time generality and specificity, and allow to capture scenario and platform specific details in a generic way. The GNEM represents experiment scenarios using three modeling entities: Experiment, Resource, and Data. The user constructs experiment scenarios by adding Experiment, Resources, and Data elements, using design primitives exposed by the generic network experiment programming interface. The experiment run primitives exposed by the experiment programming interface permit to automate the deployment and execution of the experiment based on an experiment model constructed by the user.

Generic Network Experiment Programming Interface (GNEPI). The generic network experiment programming interface is an application programming interface (API), i.e., a user interface that can be invoked from a program. It defines a set of operations that are specific to the domain of network experimentation. These operations expose the same experimentation services to the user for all experimentation platforms. The set of operations provided by the GNEPI are derived from the steps of the generic network experiment life cycle.

Generic Orchestration Engine (GOE). The generic orchestration engine is responsible for automating the experiment run. Given an experiment model constructed by the experimenter, it must ensure that all the operations needed to complete the experiment are invoked in the correct order. For this, it needs to be able to find a valid order of invocation for all the operations, taking into account any possible dependencies between resources that might impose restrictions in the order of invocation.

Figure 2.2, shows the interactions between the experimenter, the GNEPI, GNEM, GOE, and the platforms. To design an experiment, the user manipulates the GNEM by

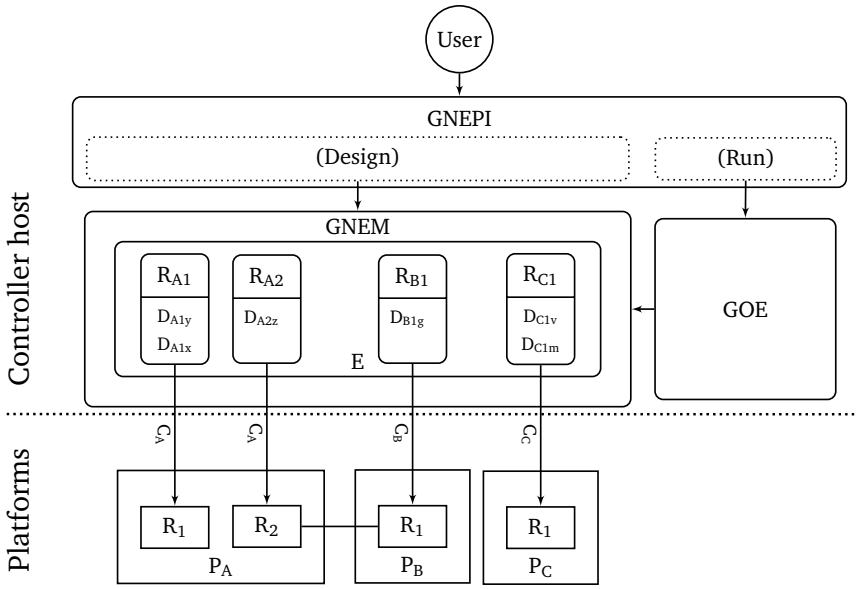


Figure 2.2 – Chain of interactions between the user and the platform resources, mediated by the GNEPI, GNEM, and GOE. The user interacts directly with the GNEPI to manipulate the *Experiment* E and the *Resources* R_{A1}, R_{A2}, R_{B1}, and R_{C1} in the GNEM during *Experiment Design*. During *Experiment Run*, the GOE interacts with the GNEM to orchestrate the experiment. *Resources* in the GNEM implement platform specific communication mechanisms, C_A, C_B, and C_C, to perform actions on platforms P_A, P_B, and P_C, during runtime. Data elements D_{A1y}, D_{A1x}, D_{A2z}, D_{B1g}, D_{C1v}, and D_{C1m} capture the output generated by resources in the platforms.

invoking design operation on the GNEPI. The user can add and configure *Resources* in an *Experiment*, and enable *Datas* in the *Resources*.

To trigger the experiment run, the user invokes run operations on the GNEPI. The GNEPI then interacts with the GOE, and delegates to it the responsibility of invoking orchestration actions on individual GNEM Resources. GNEM Resources respond by communicating with platform resources in order to perform the requested runtime actions. The communication mechanisms used to perform these actions are abstracted from the user by the *Resources* and are platform specific.

Resources also implement data collection actions in a platform specific way. Different types of platform resources produce different data outputs. The data generation and collection can be activated in a Resource element by activating a Data element.

From the point of view of an Experiment, all Resources are alike. They all expose a same set of operations that can be invoked to automate experiment actions. Resources abstract the behavior of experiment elements in a platform, like a host or an application. However, a Resource could also model additional user-defined functionality like a component that processes data once retrieved from the platform.

Resources adapt their behavior to control platform resources using the control mechanisms provided by the platform. Resources can be specialized to use any communication protocols or programming APIs to interact with a given platform because no inter-

face is specified in the architecture between the Resources and the platforms.

2.2.3 Generic Network Experiment Model

The Generic Network Experiment Model represents experiment scenarios using three entities: *Experiment*, *Resource*, and *Data*.

An Experiment entity is a conceptual unit that represents an *experiment run*, containing all resources used in the experiment and all data to be collected. A Resource entity represents anything that can be configured, performs an action, or generates data in the context of an experiment. A Data entity represents an output produced by a Resource. An instance of an entity is called an *element*. Experiment elements contain Resource elements, and Resource elements own Data elements. These entities are defined as follows:

Experiment. A container of Resource and Data elements.

Resource. Any component of an experiment that performs a specific action within the experiment and can be configured by the experimenter. Resources model elements in a platform, including hardware elements, such as hosts or devices, software elements, such as applications, and information elements, such as routing table entries.

Data. Output to be collected in the context of an experiment. A Data element is associated to the resource that generated it and the output format is not restricted.

Many platforms abstract experiments using Experiment and Resource abstractions. For example, Emulab workbench [65] defines the concept of *Instance* as "container of testbed resources", which is similar to the idea of the GNEM *Experiment* entity. Likewise, the ns-3 simulator [13], represents resources in terms of elements such as *Node*, *Channel*, and *Devices*, which can be mapped to GNEM *Resource*.

Other platforms, like XFlow [78], model network experiments exclusively in terms of operations or *activities*. Both entities and operations are intuitive ways of representing experiments. Entities intuitively capture the static part of an experiment, e.g., the topology, whereas operations allows to represent the runtime behavior of the experiment. Because of the need to intuitively model both the static and dynamic, i.e., runtime, aspects of an experiment, the GNEM uses both entities and operations to model experiments.

Generic Network Experiment Model Requirements

The following are additional requirements for the generic network experiment model:

Preservation of platform details and design granularity. A uniform experiment model does not need to present all platforms as if they were the same. Different platforms model experiments using different abstractions and exposing varying degrees of design granularity. As an example, the ns-3 simulator allows to specify what protocols will be part of the network stack of a node, whereas in PlanetLab

the protocol stack of nodes can not be modified. These platform specific details might be important parameters of an experiment, and as such, they should be exposed to the user in order to allow fine grained control of the platforms.

Support for cross-platform and multi-platform scenarios. Defining cross-platform or multi-platform scenarios should not be more complex than defining single-platform scenarios, assuming that the adaptor modules needed for the integration of Resources across platforms exist. The modelling abstractions used to describe experiments should be the same in all cases. In particular, it should be easy to describe scenarios mixing complementary evaluation platforms, i.e., simulation, emulation, and testbeds.

Support for expert and non expert users. In order to be suited for a large audience of users, the experiment model should be intuitive and adapted to both experienced researches, and non-experienced experimenters, such as students. Expert users should not be limited in the type of scenarios they can describe, or in their ability to tweak low level aspects of their experiments, while at the same time it should be possible for students to learn how to run simple experiments quickly.

2.2.4 Generic Network Experimentation Programming Interface

The generic network experimentation programming interface (GNEPI) has the objective of providing experimenters with a simple interface to automate experimentation on arbitrary platforms. For this, it exposes generic operations that are needed to support the steps in the generic experiment life cycle. These operations can be invoked by the user directly, or automatically by the experiment orchestration engine. GNEPI Operations are divided into two categories, life cycle operations, which derive directly from the life cycle steps, and methodology operations, which provide support for a rigorous experimentation methodology.

Generic Network Experiment Programming Interface Requirements

Additional requirements derived from the experiment life, but that are not captured directly as experimentation operations are Iteration, Reproduction, Interaction, and Error detection.

Support for Iteration. Iteration is the process of refining an experiment scenario by going back to the experiment conception step. Iteration is supported by using a previously created experiment model as a template that can be incrementally modified to incorporate new elements or a different configuration.

Support for Reproduction. Reproduction is supported by reusing a same experiment model, without modifications, for multiple experiment runs. Methodology operations provide further support for experiment reproduction, more details are provided in the remaining of this section.

Support for Interaction. Interaction permits experimenters to dynamically modify an experiment instance at runtime, for example, by adding or removing resources,

changing their configuration, or stopping and starting them at any moment. Interaction is supported by allowing operations that apply to Resources or Data to be invoked multiple times while the experiment is running. Groups of Resources can be deployed one after another, incrementally changing the original experiment.

Support Error Detection. On the fly error detection permits to detect when an experiment, or a part of an experiment, failed during runtime. Nevertheless, the occurrence of an error does not necessary imply an experiment failure. Experimenters must be able to specify the experiment failure criteria through *Failure Policies* that are adapted to the needs of the experiment scenario.

2.2.5 Generic Orchestration Engine

The Generic Orchestration Engine complements the Generic Network Experiment Model and the Generic Network Experiment Programming Interface by automating experiment deployment, execution, and termination. The Orchestration Engine provides the logic to resolve the order of execution of all Resource operations in an Experiment. In order to produce a valid order of execution for the operations, and successfully run the experiment, the Orchestration Engine must be able to resolve all dependencies between operations. An example of a dependency between operations is when a Resource must be deployed before another Resource. In this case the *deploy* operation of the first Resource will need to be executed before the *deploy* operation of the other.

Generic Orchestration Engine Requirements

The following are the requirements that the orchestration engine must satisfy:

Support for Arbitrary Resources. Since a Resource in the GNEM can potentially represent any element or abstraction in a platform, the GOE must be capable to deal with the orchestration of arbitrary platform elements.

Support for Arbitrary Dependencies between Resources. Any type of dependency might exist between Resources. The GOE must be able to resolve the precedence between Resource operations no matter how complex these dependencies are.

Support for Dynamic Experiment Orchestration. Following the requirement of interactive experimentation, the GOE must support runtime modifications of an experiment description, in particular it must support incremental deployment of Resources.

2.3 Specification of Architecture Components

This section presents the specification of the Generic Network Experiment Model, the Generic Network Experiment Programming Interface, and the Generic Orchestration Engine.

2.3.1 Experiment Model Entities Specification

The following section formalizes experiment description based on the generic network experiment model. As defined previously, the experiment model proposed in this work is based on three main entities: *Experiment*, *Resources*, and *Data*. E, R, and D denote the sets of all Experiment, Resource, and Data elements respectively:

$$e \in E \qquad r \in R \qquad d \in D \tag{2.1}$$

2.3.1.1 Experiment Entity

An Experiment element e is defined as a 3-tuple of experiment identifier, date, and subset of resources R_e. The experiment identifier, eid, is a human readable string assigned by the experimenter, and the date, $date$, is automatically assigned at the creation of e.

$$e = (eid \in String, date \in Date, R_e \subseteq R) \tag{2.2}$$

2.3.1.2 Resource Entity

A Resource element r is a 5-tuple formed by a global unique identifier (GUID), a resource type, a state, a subset of attributes, and a subset of Data elements.

Global unique identifiers (GUID) are unique in the scope of an Experiment. Valid GUIDs take value in E_{guid}, which can be the set of positive integers.

The resource type of a Resource is normally a string identifier, and it belongs to the set of all possible resource types, denoted R_{type}. The resource type associates a Resource to a specific implementation of the GNEPI operations, and in this way it determines the behavior of the Resource.

A Resource is found in one of few possible states at any point of its life. These states are associated to the life cycle steps. The possible states of a Resource are denoted as the set S, and will be described in detail later in this section.

Resources are configured through a set of attributes denoted by A_r. The type of a Resource determines the subset of attributes that it exposes. Resources also expose a list of data elements D_{r}, which is also determined by the type of the Resource.

$$r = (guid \in E_{\text{guid}}, rt \in R_{\text{type}}, s \in S, A_r \subset A, D_r \subset D) \tag{2.3}$$

Resource Attribute

An attribute a is a 3-tuple of name, type, and a value. Attributes represent Resource configuration parameters that can be manipulated by the user. The set of attributes associated to a Resource is determined by the resource type.

The name of an attribute is a string identifier. The attribute type at is defined in the A_{type} set, which includes basic variable types such as String, Integer, Double, Boolean,

but could potentially include complex compound types as well. The value of an attribute is of type at and takes values in A_{value}.

$$a = (name \in String, at \in A_{\text{type}}, value \in A_{\text{value}}) \tag{2.4}$$

Resource State

In order to generalize the control of arbitrary Resources, Resources are expected to transition through a common set of states. These states are derived from the analysis of the generic experiment life cycle, and they represent the life cycle of generic platform resources. The possible states of a Resource belong to S.

$$S = \{New, Ready, Started, Stopped, Released, Failed\} \tag{2.5}$$

2.3.1.3 Data Entity

A Data element d is a 4-tuple of data type, content, format, and enabled state. The type of a Data element, D_{type}, determines the Resource output that is represented by the Data element. The content is the output itself, and the format defines how the output is presented. A Data element can be in one of two states, enabled (1) or disabled (0). Disabled Data elements do not actively generate data.

$$d = (dt \in D_{\text{type}}, content \in String, format \in String, enabled \in \{0, 1\}) \tag{2.6}$$

2.3.1.4 Runtime Behavior Model

This section specifies the behavior of Resource entities based on a generic Resource state machine and the relations between individual Resources.

Resource State Machine

The behavior of individual Resources follows a *life cycle* defined by the state machine in Figure 2.3. A Resource starts in state *New* and can either finish in state *Released* or in state *Failed*. The operations *deploy*, *start*, *stop*, and *release* are exposed through the GNEPI. These operations are invoked automatically by the Orchestration Engine on all Resources in an Experiment.

The execution of an operation results in the transition of a Resource from one state to another. If the execution is unsuccessful, the Resource transitions to state *Failed*. However, if an operation is invoked but the Resource is not able to execute it at that time, the Resource stays in its current state until the operation is invoked again at a later time. This means that invoking an operation has three possible outcomes: a state transition to the next state, a transition to state *Failed* or no state transition. For instance, invoking the *deploy* operation on a Resource in state *New* can result in the execution of the *deploy* operation and its subsequent transition to state *Ready*. If the deployment of

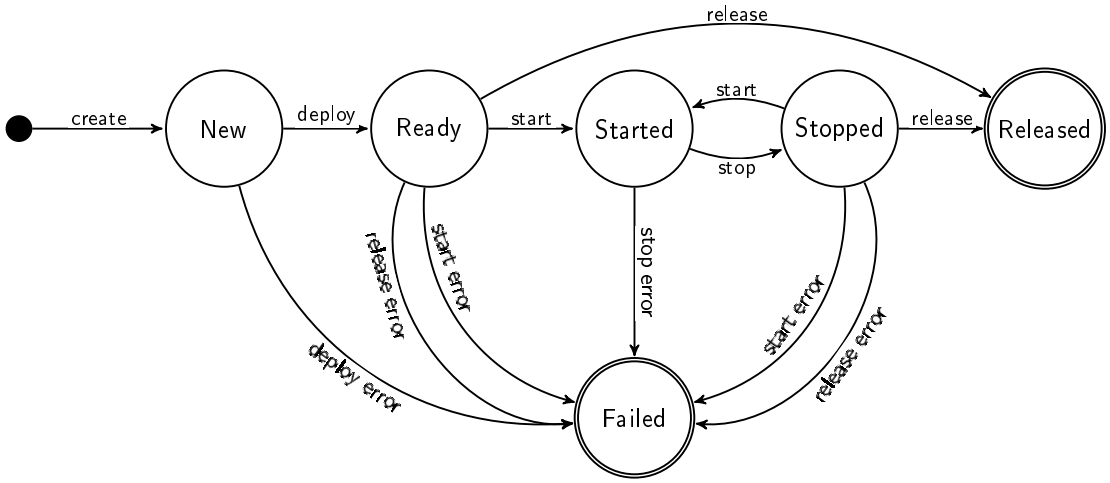


Figure 2.3 – State machine that models the run time behavior of Resources based on the Resource states *New, Ready, Started, Stopped, Released, and Failed*.

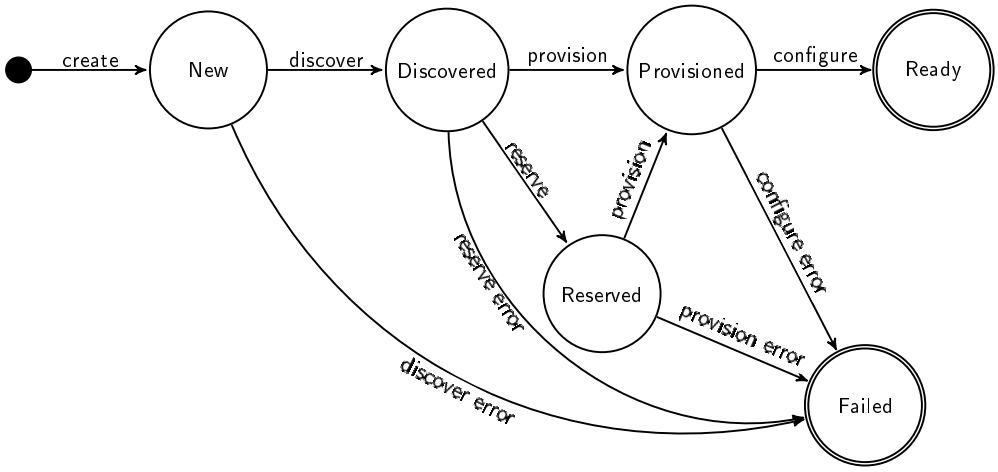


Figure 2.4 – State machine modeling the extended state transitions for Resources between states *New* and *Ready*.

the Resource fails then the Resource will pass to state *Failed*. If the Resource can not be deployed when the operation is invoked, for example because it needs to wait until another Resource is deployed first, then the *deploy* operation is not executed and the Resource remains in state *New*.

If no state change occurs after invoking an operation, then the operation can be invoked again on that Resource at a later time. The actions performed by a Resource when an operation is executed are determined by the implementation of the operation, which is determined by the Resource type.

As shown in Figure 2.4, the transition between states *New* and *Ready* can be further decomposed into the intermediate extended states *Discovered*, *Reserved*, and *Provisioned* in S_{extended}. Transitions between those states are triggered by the operations *discover*, *provision*, *reserve*, and *configure*.

$$S_{\text{extended}} = \{Discovered, Reserved, Provisioned\} \tag{2.7}$$

The compact 6-states Resource Machine from Figure 2.3 is adopted for simplicity in the rest of this chapter.

Resource Relations

To model an experiment it is not sufficient to define and configure Resources. It is also necessary to indicate how Resources relate to one another. Relations between resources can be *topology based*, e.g., a node connected to a channel or an application running on a node, or *state based*, e.g., constraining the precedence of state transitions between resources such as imposing a client application to start after a server application. When modeling an experiment, users can describe topology based relations between Resources using *connections*, and state based relations using *conditions*.

Connections define *static* interactions between Resources. They are used to describe the topology of an experiment, i.e., how Resources that represent network components, protocols, and applications are associated to each other. A connection is an undirected association between two Resources. An Experiment topology can be represented as an undirected graph of Resources where the connections are the edges.

Conditions define *runtime* dependencies between Resources that modify the base behavior defined by the Resource state machine. Conditions allow users to model experiment *workflows* that incorporate additional constraints for state transitions of Resources. A condition is a directed association between a Resource-state pair and a Resource-operation pair, and it can optionally specify a time delay. Workflows defined by the user can be represented as a directed weighted graph with two types of vertices: Resource-state and Resource-operation, and the optional delay as the edge's weight.

Connections imply semantics that depend on the type of the Resources connected. For example, connecting a Resource that represents a host in platform A with another Resource that represents an application in that same platform might mean *execute application in host*, whereas associating the same host Resource to another host Resource in platform B might mean *establish a tunnel between the two hosts*. These semantics will be determined by the resource type of the connected Resources.

Conditions define the dynamic or temporal behavior of an experiment. They create additional constraints for the state transitions of Resources. Conditions are used to define *workflows* between resources, for instance to define when Resources should be deployed, started, or released with respect to other Resources. A condition might specify

that the *start* operation for an application x Resource must be invoked after another Application y is in state *Started*.

Users can modify the implementation of Resource operations to change the Resource behavior. The new implementation can consider a different criteria to transition from one state to another. The ability to define workflows using state conditions provides an easy way for users to modify the default behavior of Resources without having to modify the implementations of operations.

Conditions are directed relations between two tuples: R, S and R, O, where R is the set of Resources in an Experiment, S is the set of possible states, and O is the set of operations:

$$O = \{deploy, start, stop, release\} \tag{2.8}$$

A condition is denoted as follows:

$$R_x, operation \rightarrow (delay)R_y, state \tag{2.9}$$

This condition expresses that "the operation $operation$ on Resource x must be executed $delay$ time after Resource x has reached state $state$". The delay is an optional parameter.

The idea of separating the experiment description into a static and a dynamic part is also proposed by other experiment automation tools. Examples of this are Emulab [65] and OMF [85], both use resource abstractions to define the static part of an experiment and events to model its dynamic behavior.

The experiment model proposed in this work is compatible with the federation model proposed by Wahle [68]. The GNEM and GNEPI abstractions correspond to the external SET component in the architecture proposed by Wahle. The SET component is the user's entry point to the federation architecture. Unlike the federated experiment description model proposed by Wahle, the GNEM does not define hierarchies between Resources. In the GNEM, both connections and conditions define horizontal non-hierarchical relationships between Resource pairs. The reason for this choice is to avoid the additional complexity that arises from describing and resolving hierarchical constraints between Resources. Furthermore, representing relations between any type of Resource in a uniform way simplifies the description of cross-platform experiments, since the same rules are followed to interconnect Resources from a same or different platforms.

2.3.2 Experimentation Operations Specification

This section defines the operations exposed by the Generic Network Experiment Programming Interface. The operations are divided into two categories: Life Cycle Operations, which are derived from the Generic Experiment Life Cycle, and Methodology operations, which are additional operations to provide support for a rigorous experimentation methodology.

2.3.2.1 Life Cycle Operations

Table 2.2 lists the life cycle operations derived from the generic experiment life cycle steps in Section 2.1. The life cycle operations apply to *Experiment* (E), *Resource* (R), and *Data* (D) model entities. The subindex on the name of the operation denotes which type of entities the operation applies to.

Operations that apply to Resources include the ones previously discussed in relation with the Resource state machine. These are the *deploy*, *start*, *stop*, and *release* operations, as well as the four operations associated to the extended Resource states, *discover*, *reserve*, *provision*, and *configure*. These operations are runtime operations automatically invoked by the Generic Orchestration Engine.

An Experiment entity is constructed using the design operations *create*, *add*, *connect*, and *condition*. *create* is used to instantiate an Experiment entity, and *add*, *connect*, *condition*, *set* are used to describe Resources. A user-defined *failure policy* for the Experiment can be specified in the *create* operation.

The *set* and *get* operations are used during experiment design as well as during experiment runtime to query and modify resource configuration. The *measure* operation is used to activate measurement probes, i.e., Data elements.

The orchestration of an Experiment, or a part of it, is triggered when the user invokes the *deploy* operation on the Experiment instance[1]. Then Generic Orchestration Engine is notified of this, and proceeds to deploy the selected Resources. An Experiment instance can be modified dynamically, and Resources can be continuously added, connected, configured, and deployed by invoking the corresponding design operations.

Both Experiment and Resources can be monitored during runtime using the *state* and *status* operations respectively. The *inspect* and *collect* operations can be used to obtain information about collected data and to download and archive the data in a local storage during runtime or once the Experiment is over. The *process* and *plot* operations are used to analyse and visualise collected data or other Experiment information.

Invoking the *shutdown* operation causes the GOE to invoke the *release* operation on the Resources in the Experiment. This is a blocking operation.

[1]There are two deploy operations, one that applies to Resources and the other that applies to Experiments.

Step	Operation	Description
Experiment design	Create $_{E}$	Instantiate the experiment model that will be used to describe the experiment scenario.
	Add $_{E, R}$	Declare a new resource in the experiment model instance.
	Set $_{R}$	Modify the configuration of a resource.
	Connect\Condition $_{R}$	Establish a relation between two resources in the model. There are two types of relation between resources, Connections and Conditions.
	Measure $_{R, D}$	Activate a measurement point on a resource so specific data can be collected.
Experiment deployment	Deploy $_{E}$ Deploy $_{R}$	Perform all necessary actions, to prepare a resource or group of resources for execution.
	Discover $_{R}$	Obtain a list of resources available in the platform, possibly matching specific criteria requested by the experimenter.
	Reserve $_{R}$	Obtain a time slot to access and control a resource or group of resources.
	Provision $_{R}$	Allocate a resource or group of resources and make it ready to be accessed and used by the experimenter.
	Configure $_{R}$	Apply initial configuration to a resource or group of resources according to the requirements of the experiment.
Experiment execution	Start $_{R}$	Activate a resource or group of resources so they can take part in the experiment and perform the actions that they are required to perform.
	Stop $_{R}$	Deactivate a resource or group of resources so they interrupt the actions they were performing as part of the experiment.
Experiment monitoring	State $_{E}$	Retrieve information about the success or failure of the experiment as a whole.
	Status $_{R}$	Retrieve information about the liveliness of a resource and its state with respect to the steps of the experiment life cycle.
	Get $_{R}$	Retrieve information about the configuration of a resource.
Experiment termination	Shutdown $_{E}$	Terminate the experiment. Release the resources used in the experiment and perform any necessary clean up routines.
	Release $_{R}$	Deallocate a resource or group of resources so that they are left in the same state in which they were found initially.
Data collection	Inspect $_{D}$	Retrieve data generated during the experiment execution as a stream.
	Collect $_{D}$	Retrieve data generated during the experiment execution and persist it to local files.
Data processing	Process $_{D}$	Transform data in order to extract meaningful information.
	Plot $_{D}$	Generate a graphical representation of the data or other experiment information.

Table 2.2 – Experimentation operations derived from the generic experiment life cycle steps. Operations can be performed on three types of entity: *Experiment*, *Resources*, and *Data*, denoted by sub indices E, R, and D.

2.3.2.2 Methodology Operations

Table 2.3 lists additional operations, exposed by the generic network experiment programming interface, aimed at providing support for a rigorous experimentation methodology. A rigorous experimentation methodology should provide documentation to describe in detail the experiment scenario and produce well organized result archives. For traceability purposes, it should be possible to map collected results to the elements in the experiment that generated them. A rigorous experimentation methodology should also permit to reproduce the experimentation procedure.

The methodology operations in the GNEPI provide support for experiment documentation and data archiving and sharing. The *reproduce* operation permits to automatically re-run an experiment. The *replicate* operation helps to translate a scenario to a different platform, simplifying multi-platform experimentation. The *load* and *save* operations are used to document an experiment definition in human readable form, and the *archive* operation is used to archive experiment results.

Operation	Description
Reproduce	Re-run an experiment multiple times.
Replicate	Run an experiment scenario on different platforms. This requires mapping scenarios across multiple platforms.
Save	Persist the definition of an experiment in a human readable format, so it can be used both as experiment documentation and to re-run the experiment.
Load	Construct an experiment instance from an experiment definition persisted using the *save* operation.
Archive	Store experiment results in a directory structure defined by the user, adding information about the experiment execution time and other contextual information, such as the identifier of the Resources that generated the results.

Table 2.3 – Additional experimentation operations to support the requirements of a rigorous research methodology.

2.3.3 Orchestration Algorithm Specification

This section introduces an orchestration algorithm based on modeling the orchestration problem as a scheduling problem. The proposed algorithm is an on-line scheduling algorithm that uses a black-box approach to satisfy dependencies between operations.

2.3.3.1 Experiment Orchestration as a Scheduling Problem

On the basis of the proposed experiment model, experiment orchestration consists in executing the *deploy*, *start*, *stop*, and *release* operations for all Resources in an Experiment in a valid order. A valid order of execution must respect the dependencies between operations, taking into account the state of the Resources and the constraints imposed by the Resource state machine and the workflows defined by the user. Relations between

resources impose constraints on when operations can be executed, making state transitions in one Resource dependent on the state of other Resources. Operations might also impose internal constraints for their execution that are specific to a type of Resource, such as waiting for an external service or infrastructure to become available. The Generic Orchestration Engine is responsible for executing all the operations in an order that satisfies all constraints.

Taking into account the characteristics of the model chosen to describe experiments, the problem of experiment orchestration can be seen as a job shop scheduling problem [94] (JSP), where the jobs are the operations and the precedence graph is given by the constraints in the order of execution of operations. A job shop scheduling problem consists in finding an optimal assignment of N jobs to M identical machines, taking into account the job durations and the precedence between jobs, that minimizes the total time needed to execute all jobs, i.e., the makespan. If operations can be executed in parallel, then the number of machines M corresponds to the number of threads used to parallelize operation execution.

Variations to the JSP Problem

The experiment orchestration problem addressed in this work present several differences with a classical job shop scheduling problem.

Job duration not known in advance. The first difference is that the job duration (the operation execution time) is not known. Operations perform on arbitrary platform resources, including live resources, which are uncontrolled and do not guarantee response times. For this reason, it can not be assumed that the durations of all operations are known. Without knowing the operation execution time, it is not possible to find an optimal job scheduling that minimizes the total makespan. In this case the optimization problem becomes an ordering problem whose only goal is to find a valid order of execution instead of an optimal order of execution.

Precedence graph changes dynamically or not fully known. The second difference is that all the operation constraints might not be known in advance. Since a Resource can potentially model any element or service in a platform, existing or future, and since a design choice was to specify the interface of operations but not their implementation, there are no limitations on what an operation can do or what constraints it might impose for its execution. This means that although there are some general constraints that must be respected, such as those imposed by the Resource state machine, operations might impose any other constraints for their execution. A given operation might for instance depend on an external event that might occur at an unknown time. Furthermore, since Experiments can be modified dynamically, adding a new Resource might change the precedence graph while the experiment is being orchestrated.

Off-line vs On-line Scheduling Algorithms

There are two main approaches to resolve scheduling problems: off-line algorithms and on-line algorithms. An off-line scheduling algorithm is a NP-hard optimization problem, where all the required information to compute the solution is known in advance: full dependency graphs, job durations, etc. It might be possible, though computationally expensive, to find an optimal schedule using an off-line algorithm. An off-line algorithm is not well suited for the orchestration problem since it presents several unknown parameters.

On-line algorithms [95] make scheduling decisions on the fly, knowing only the history of scheduled jobs but not the jobs that might arrive in the future. A well known on-line scheduling algorithm is the List scheduling algorithm [96, 97], or List algorithm, considered one of the best and simplest on-line algorithms [98]. Graham's List algorithm is $(2 - 1/M)$competitive[2] where M is the number of machines, and it finds an optimal solution for 2 and 3 machines.

Algorithm 1 Simplified Graham's List scheduling algorithm

Require: set of identical machines, priority list of jobs, partial ordering of jobs

```
1: function SCHEDULE
2:     while jobs in list do
3:         if available machine then
4:             find next job with no pending predecessors
5:             remove job from list
6:             assign job to machine
7:         end if
8:     end while
9: end function
```

Graham's List algorithm is well suited for the orchestration problem since it supports dealing with partial precedence information and dynamic arrival of jobs. The remaining issue to resolve is how to represent job precedence information.

White-box vs Black-box Approach for Job Precedence

Precedence relations between operations are determined by their dependencies, e.g., workflows, state machine, internal logic. Dependencies between operations can be modeled in two ways: using a white-box approach, which consists of explicitly codifying dependencies between operations and exposing them to the scheduling algorithm, or using a black-box approach, where dependencies are hidden from the scheduling algorithm and are only known by the operation implementation.

The white-box approach requires defining an explicit and generic way of describing dependencies between Resource operations, which must allow to define a set of rules

[2]The competitiveness, i.e., competitive analysis, is a measure of the performance of an on-line algorithm compared to its optimal off-line algorithm.

to construct a global dependency tree. The leaf operations of the dependency tree are those that have no pending dependencies and can be executed right away. An algorithm can then traverse the dependency tree executing operations with no pending dependencies from leaf to root. The white-box approach presents advantages such as allowing to detect dependency loops, which would deadlock the experiment orchestration. Nevertheless, it also imposes certain limitations.

The first limitation is that the dependencies that can be supported by the orchestration algorithm are restricted to those that can be expressed in the dependency description language. In particular, this might not allow to define dependencies with external systems that are not necessarily modeled directly as Resources in the Experiment, for example if the deploy operation for a certain Resource type needs to wait for the confirmation of an external reservation system in a platform before it is able to execute. It might also pose problems if dependencies between operations change dynamically, for instance if new Resources added during the orchestration modify the dependency tree, forcing to recompute the tree after the execution of each operation. Also, dependencies might be conditional, e.g., an operation depending on the occurrence of an event, which adds extra complexity to the description and resolution of precedence between operations.

In the black-box approach the global dependencies tree is never known, and the scheduling algorithm can only know whether an operation has pending dependencies or not by querying it. Operations only hold partial dependency information concerning their neighborhoods and only local dependency relations needs to be evaluated. The black-box approach provides more flexibility than the white-box approach to support arbitrary dependencies, because dependencies do not need to be explicit described, not depending on the expressiveness of the dependency description rules. Nevertheless, in a black-box approach dependency analysis and loop detection are not as straightforward as in a white-box approach. Dependency loops can be prevented by carefully analysing the implementation of operations and their constraints, and by monitoring the progress of experiments in order to detect a blocking in the orchestration.

2.3.3.2 Proposed Algorithm for Experiment Orchestration

The orchestration algorithm proposed in this work is based on Graham's List algorithm. The job priority is given by the job execution time. There might be several jobs with the same execution time waiting to be executed. The initial execution time of a job is assigned when the job is first inserted in the jobs list and it is normally the insertion time, but it can also be a time in the future. Jobs can be inserted at any moment in the list.

No explicit partial ordering for the jobs is provided as input of the algorithm. Instead, at any moment a job can be queried about its ready state, and it can either indicate that it is ready for execution, i.e., no pending dependencies, or that it is not. If a job is not ready for execution, but its execution time has elapsed, it is rescheduled, i.e., re-inserted in the list, with a new execution time in the future, equal to $t_{now} + \Delta t$.

The pseudo code of the algorithm is given in Algorithm 2. Jobs are ordered by their execution time and the order is kept when rescheduling jobs. Whenever a machine is available, the next job with the earliest execution time is queried for its state. If the job is ready, then it is assigned to the machine. If it is not ready, then it is re-inserted in the list with a new execution.

Algorithm 2 Orchestration algorithm

Require: set of identical machines, priority list of jobs

```
1:  function SCHEDULE
2:      while jobs in list do
3:          if available machine then
4:              find next job with smallest execution time
5:              remove job from list
6:              if job not ready then
7:                  insert job back in list with execution time t_now + Δt
8:              else
9:                  assign job to machine
10:             end if
11:         end if
12:     end while
13: end function
```

The main differences with Graham's algorithm is that jobs can be rescheduled, and that instead of having a global ordering of jobs, jobs are subjected to a binary question: either they are ready for execution or not. The algorithm uses a black-box approach for job precedence resolution that allows jobs, i.e., operations, to implement arbitrary internal constraints, possibly considering external events and services not directly modeled in as part of the experiment. The black-box approach also permits to adapt to changes in an experiment, i.e., dynamically adding new Resources, without the need of recomputing a global dependencies tree.

If there is a valid order of execution, it will be resolved naturally by attempting execution of all jobs on the list repeatedly. It is not straightforward to analyse the time complexity of the algorithm since it depends on the implementation of the sub-algorithms to find jobs and assign jobs to machines. The number of iterations needed to resolve the execution of all jobs is tied to the specific conditions imposed by each operation. Operations might need to wait for a variable amount of time before being able to execute, or depend on external events or on other operations. If many jobs are ready for execution at a same time, using multiple machines can speed up orchestration by a factor of M, but if there are unmet dependencies that prevent job execution, adding more machines might only degrade performance.

An evaluation of the efficiency of the proposed orchestration algorithm is provided in Chapter 5. The evaluation uses an implementation of the algorithm to benchmark its efficiency in terms of time and memory needed to orchestrate experiments, using a dummy platform, a simulation platform, and a testbed platform. The results show that

the algorithm implementation is able to resolve orchestration in linear time and with linear memory consumption with respect to the number of Resources in the experiment.

The algorithm as it is presented is not capable of detecting deadlocks, i.e., dependency loops. A way to improve the algorithm to support deadlock detection is to allow jobs to reply to the state query with one of three values: *ready*, *waiting for operation*, or *waiting for event*. If at a given point all operations are waiting for other operations, then the orchestration is in deadlock. Loops could be detected in user-defined workflows by analysing the dependency graph formed by the Resource conditions and the state transition constraints imposed by the Resource state machine. Nevertheless, this would ignore hidden constraints implemented in the operation logic. Deadlocks due to internal logic constraints can be prevented by being careful with the implementation of operations.

The semantics of the user-defined workflows only allow to specify *after* constraint guarantees, i.e., an operation to be executed *after* another operation. *Before* guarantees are not supported. If the execution time for a given operation is critical, then this must be reflected in the implementation of the operation. An operation can fail during execution and trigger the interruption of the experiment run.

Example

The following example considers an Experiment composed of four Resources, one node r_1, and three applications r_2, r_3, and r_4, where the three applications are connected to the node. The orchestration engine uses two threads, i.e., $M = 2$. For simplicity only the *deploy* and *start* operations are considered for each Resource, i.e., total number of operations is eight. *Deploy* operations are denoted r_i, d where i is the Resource number, and *start* operations are denoted r_i, s. Resources pass to *Ready* and *Started* states after the corresponding operation execution. Operation execution failures are not considered in this example. The orchestration starts at $time = 0$ and r_4 is added at $time = 3$.

The following additional information characterises the scenario:

Operation Duration

The following durations, in time units, were arbitrarily chosen for the operations.

$$\begin{aligned} r_1, d &= 1 \qquad & r_1, s &= 1 \\ r_2, d &= 1 \qquad & r_2, s &= 2 \\ r_3, d &= 2 \qquad & r_3, s &= 1 \\ r_4, d &= 1 \qquad & r_4, s &= 1 \end{aligned} \tag{2.10}$$

State Machine Constraints

The following constraints for *deploy* and *start* operations are imposed by the Re-

source state machine.

$$\begin{aligned} r_1, s &\rightarrow r_1, d \\ r_2, s &\rightarrow r_2, d \\ r_3, s &\rightarrow r_3, d \\ r_4, s &\rightarrow r_4, d \end{aligned} \tag{2.11}$$

User Defined Workflows

An arbitrary user-defined workflow is assumed in this example. The workflow specifies that application r_3 can only start two time units after the start of application r_2.

$$r_3, s \rightarrow (2)\ \ r_2, s \tag{2.12}$$

Internal Logic Constraints

Operations might impose internal logic constraints that define dependencies with other operations and reflect the semantics of relations between Resources. This example assumes that the applications can only be deployed after node r_1 started. Application r_4 must also wait for an external event e which occurs, at time $t = 5$, before it can start.

$$\begin{aligned} r_2, d &\rightarrow r_1, s \\ r_3, d &\rightarrow r_1, s \\ r_4, d &\rightarrow r_1, s \\ r_4, s &\rightarrow e \end{aligned} \tag{2.13}$$

Dependency Graph

Figure 2.5 depicts the precedence graph for this example, summarizing the constraints between all operations. The orchestration algorithm must execute all operations respecting all constraints but without having a full centralized view of the graph.

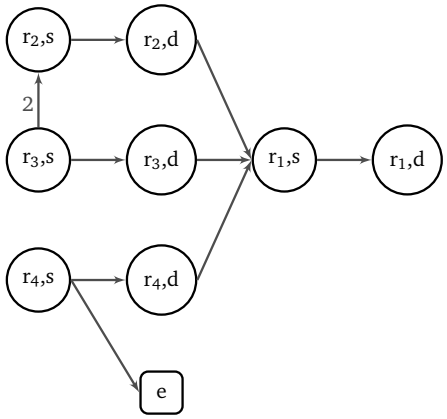


Figure 2.5 – Dependency graph showing precedence relations between all operations in the example scenario.

Experiment Orchestration

Figure 2.6 shows the step by step resolution of the orchestration for the proposed example, using Algorithm 2. For simplicity, in this example the time is discretized in steps, but the algorithm can also work in continuous time. At each time step, the list is searched until the next $M - M_o$ ready jobs are found, where M_o is the number of occupied machines. The Δt for the new execution time is arbitrarily set to 1 time unit.

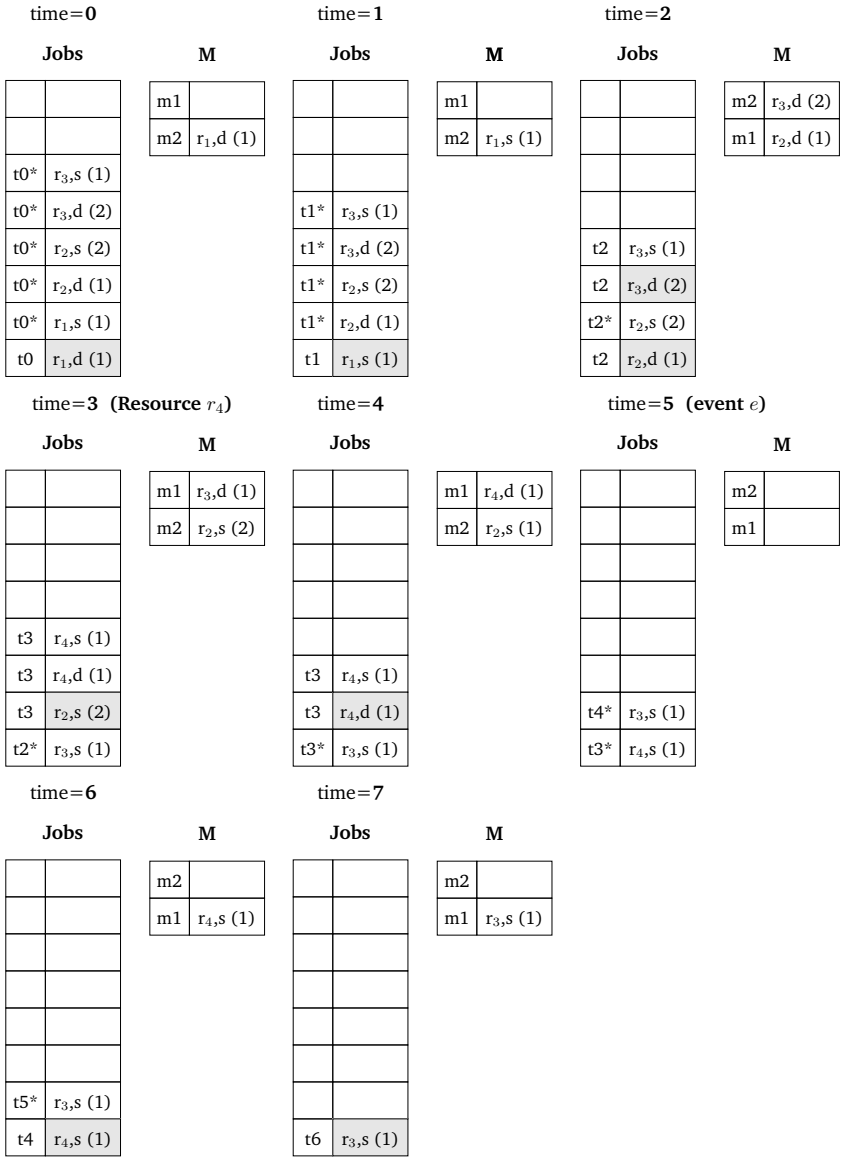


Figure 2.6 – Orchestration of the example experiment using the orchestration algorithm in Algorithm 2. A '*' in the execution time of a job marks it as rescheduled. The number in parenthesis next to a job indicates the remaining duration of the job. By convention, jobs are assigned to the available machine with lower index. To be executed, operation r_3,s must wait 2 time units after operation r_2,s is executed. Operation r_4,s can only be executed after event e occurred at time $t = 5$.

Chapter 3

Automation Framework Implementation

The generic network experiment automation approach presented in the previous chapter can be implemented in many ways. The current chapter presents one possible implementation in the form of a programming framework. This framework supports automating network experiments on various platforms, including simulators, emulators, testbeds, testbed federations, and combinations of them for cross-platform and multi-platform experiments. The goal of this framework is three fold: First, to show the feasibility of the proposed experiment automation approach by producing a working implementation. Second, to provide the means of evaluating the approach. Third, to provide a tool that can be used by the networking research community to automate network experiments.

This work originated in the Network Experiment Programming Interface (NEPI) [90] prototype, and for historical reasons the same name was kept. However, the current version 3.0 completely redefines the internal orchestration mechanisms and the experiment model used by NEPI, following the approach described in Chapter 2.

The main improvements with respect to the original NEPI version are a simplification of the experiment model, the incorporation of user workflows and dynamic experiment orchestration, the support for experiment iteration, reproduction, interaction, and error policies, and added support for rigorous experimentation, including experiment replication and data archiving. The capabilities of NEPI 2.0 for cross-platform experimentation have been adapted to the new experiment model and orchestration mechanisms. Support for multi-platform experimentation has been extended with automatic topology generation features. Previously supported platforms, such as the ns-3 simulator and NetNS, have been migrated to the new automation approach, and new platforms for emulation and live experimentation, such as DCE, PlanetLab, and OMF have been added to the framework.

3.1 The NEPI Project

NEPI is provided as a Python library that can be imported from a Python script to programmatically model and run experiments. The reason to choose a scripting language to implement the framework is the smoother learning curve of scripting languages compared to typed programming languages like C++ or Java. The fast prototyping and the low usage barrier of Python were preferred to the better performance offered by compilable languages. Python also offers a large library base for various uses, including data processing and plotting, e.g., numpy, matplotlib, as well as several network communication libraries, e.g., socket, xmpp. Furthermore, bindings for C and Java libraries can be easily implemented in Python, making it possible to interface with software simulators and emulators written in those languages.

NEPI is licensed under GPLv2. This license choice is intended to be an incentive for code sharing and collaborative development and verification of results within the network research community. Releasing the code makes it easier for anyone to review and improve it. Making the code freely extensible allows experimenters to build upon previous efforts to foster innovation. The project source code, documentation, examples, and other resources are publicly available online at:

`http://nepi.inria.fr`

User and developer lists are available to encourage exchange and community growth around the framework, although this growth has been slow and the NEPI community remains very small at this time. Measures to improve dissemination and improve visibility of the framework are planned as part as the NEPI development roadmap.

3.2 NEPI Architecture

This section describes the architecture of the NEPI framework with respect to the components of the generic network experiment automation architecture defined in Chapter 2: the Generic Network Experiment Model (GNEM) the Generic Network Experiment Programming Interface (GNEPI), and the Generic Orchestration Engine (GOE).

As shown in Figure 3.1, the architecture of NEPI is divided into three layers: *Experiment Control Layer*, *Resource Control Layer*, and *Communication Layer*. The components in these layers provide the implementation for the components of the generic experiment automation architecture. The ExperimentController (EC), the ResourceManagers (RM), and the Traces (T) implement the Generic Network Experiment Model, the Experiment API and the Resource API implement the Generic Network Experiment Programming Interface, and the Scheduler implements the Generic Orchestration Engine.

The components on each layer interact vertically to establish resource control between the user and the resources in the platforms, and provide experiment automation. The ExperimentController represents an experiment run, the ResourceManagers represent resources in the platforms, and the Traces represent data to be collected. The user

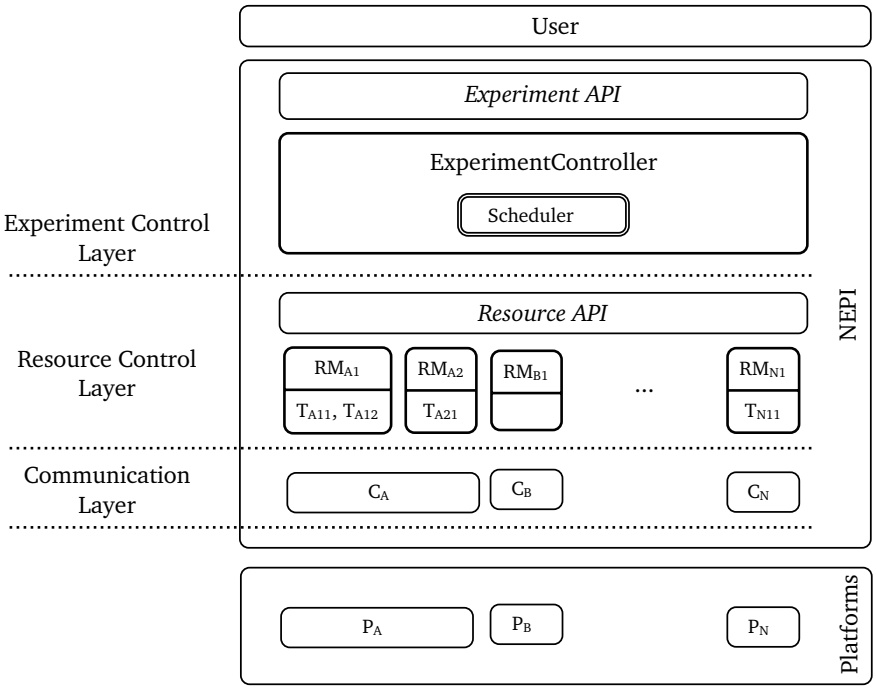


Figure 3.1 – Architecture of the NEPI framework. The NEPI architecture is divided into three layers: *Experiment Control*, *Resource Control*, and *Communication*. The layers interact vertically to provide experiment automation between the user and the platforms. The components on the different layers implement the GNEM, GNEPI, and GEO components of the generic experiment automation architecture defined in Chapter 2. The ExperimentController (EC), the ResourceManagers (RM), and the Traces (T) implement the GNEM, the Experiment API and Resource API implement the GNEPI, and the Scheduler implements the GOE.

interacts with the EC through the Experiment API in order to design and run experiments. In turn, the EC interacts with the RMs through the Resource API. The Resource API exposes design, deployment, and execution operations, as well as data collection and measurement capabilities. All RMs must implement the same Resource API and provide the underlaying logic to communicate with and control resources in the target platform.

There is no specific API that mediates between the RMs and the platforms. Each RM can use a different communication mechanism to perform actions on the platforms and control resources. The components in the Communication Layer can be thought of as drivers that adapt RMs to the specific communication and control primitives supported by individual platforms.

3.2.1 Generic Network Experiment Model

The *Experiment*, *Resource*, and *Data* entities in the Generic Network Experiment Model are implemented by the *ExperimentController*, *ResourceManager*, and *Trace* classes respectively.

ExperimentController (EC). Users manipulate experiments through the EC. The EC implements the *Experiment API*, which defines a subset of the Generic Network Experiment Programming Interface operations. It exposes experiment design and experiment run methods to the user. The EC contains a collection of ResourceManager instances that model platform resources. The EC is platform independent and relies on the RMs it contains to perform platform specific actions. The EC is also in charge of experiment orchestration and monitoring and contains the *Scheduler* component that implements the orchestration algorithm.

ResourceManager (RM). RM instances represent platform resources and internally follow the Resource state machine presented in Chapter 2. They hold information about the current state of a Resource and the conditions required to pass to the next state. Extending the framework to support a new type of resource requires adding a new RM class that implements the *Resource API* to support platform specific control logic.

Trace. Traces represent data to be collected in association to a RM. Traces can be used to expose any type of data, and to retrieve the data dynamically during the experiment execution.

3.2.1.1 ResourceManager Instances

Resources in NEPI are modeled as instances of ResourceManager (RM) classes. There can be many different RM classes, each capable of managing a different resource or service on a platform. A RM class is associated to a unique string identifier, the resource type.

RM instance
resource type: A::Application
guid: 1
state: NEW
attributes: *command*="ping nepi.inria.fr", *shell*="bash"
traces: stdout, stderr

Figure 3.2 – Representation of a ResourceManager instance that models an application on Platform A. ResourceManager instances have a resource type, a global unique identifier (GUID), a state, a list of attributes, and a list of traces.

Users manipulate RM instances indirectly through the ExperimentController (EC). The EC holds a reference to a ResourceFactory object that implements the Factory design pattern to instantiate RM instances. When a user requests the creation of a RM instance, the EC delegates its creation to the ResourceFactory, passing the corresponding resource type. The EC keeps an internal reference to all instantiated RMs, along with their global

unique identifiers (GUIDs). The GUID is the reference the user employs to request operations on a RM instance.

Figure 3.2 shows the information of the RM instances that is exposed to the users. A RM instance is associated to a list of attributes and a list of traces. The elements in those lists are determined by the resource type. Attributes are used to specify the configuration of a resource and traces to specify the data elements to be enabled during experimentation. RM also have a state whose possible values are: NEW, DEPLOYED, DISCOVERED, RESERVED, PROVISIONED, READY, STARTED, STOPPED, FAILED, and RELEASED. Additionally, RM instances are associated to a list of connections and a list of conditions, i.e., workflows, to keep track of the relations between resources.

3.2.1.2 ResourceManager Class Hierarchy

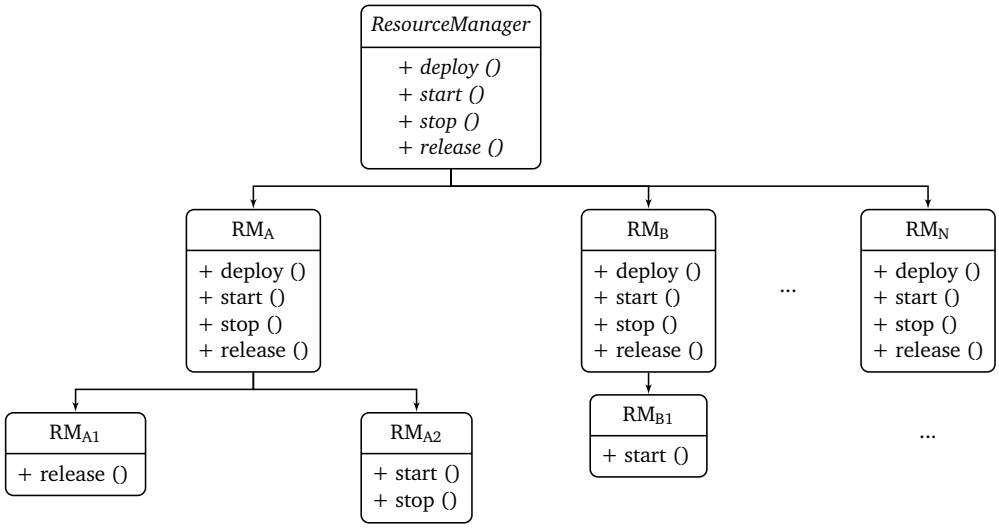


Figure 3.3 – Family of ResourceManager classes extending the base ResourceManager abstract class.

The RM class hierarchy is the structure that enables extending the framework to support resources on arbitrary platforms. All RM classes extend a same parent abstract class, the ResourceManager class, which defines the Resource API. The Resource API exposes as class methods the operations of the Resource Life Cycle that represent the common behavior of all resources, e.g., deploy, start, etc. RM classes must provide implementations for the methods in the Resource API to manage resources on specific platforms.

RM classes can be sub-classed any number of times. A RM class can override the implementation of all or a subset of the Resource API methods defined by its parent class. Figure 3.3 provides an example of a possible class hierarchy to support resources in hypothetic platforms A, B, and C. RM classes RM_A, RM_B, and RM_C extend the abstract ResourceManager class and implement the abstract methods of the Resource API. Fur-

ther derived RM classes, RM_{A1}, RM_{A2}, and RM_{B1}, override only a subset of their parent class methods.

3.2.1.3 Extending the ResourceManager class

The ResourceManager class is the parent class of all RM classes. The source code of the ResourceMananger class can be found in the NEPI project source code at *src/nepi/execution/resource.py*. A template to help NEPI users to add new RM class is available as part of the NEPI documentation[1]. This template is included in Appendix C. The steps required to create a new RM class are the following:

1. Choose a platform identifier and a resource type identifier for the new ResourceManager class. Normally, a resource type identifier should be of the form *<platform>::<resource_type>*.
2. Create a Python module, a file with .py extension, and place it under the directory *src/nepi/resources/<platform>* in the NEPI project sources.
3. In the .py file create a new class that inherits from one of the existing RM classes. A RM class can be a child of the base ResourceManager or any of its children.
4. Declare attributes and traces for the new class. Attributes from the parent class are automatically inherited if the *@clsinit_copy* decorator is used in the class declaration.
5. Implement the relevant methods from the Resource API.

The most important step in the creation of a RM class is the implementation of the Resource API methods. Each public method in the Resource API, e.g., deploy, start, etc, is associated to a private method prefixed with *do_*, e.g. *do_deploy* and *do_start*. Rather than directly overriding the public methods, users must re-implement the private *do_* methods. The private *do_* methods are invoked by the public method after performing validation and synchronization actions. This separation between the public and the private methods of a given operation is intended to allow users to concentrate on the implementation of resource specific logic without having to deal with the framework general logic.

For illustration, Listing 3.1 shows the definition of a Dummy platform composed of four ResourceManagers representing Link, Interface, Node, and Application resources. The *do_deploy* method of the RMs in the Dummy platform only does a sleep, during a time defined by the *wait_time* variable, instead of performing platform specific actions. The Dummy Application RM also sleeps for a given amount of time when the *do_start* method is invoked. Internal dependencies, i.e., operation constraints, are defined between *do_deploy* methods of the RM classes. For instance, the Application and the Interface classes need to wait until the Node they are connected to is in state READY before they can proceed. The Interface additionally needs to wait for the Link to be in

[1] Available in the NEPI online repository at
`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/doc/templates/template_rm.py`.

state READY. If all dependencies are not met when a method is invoked, the method reschedules itself to be invoked again in the near future, with a delay specified by the *self.reschedule_delay* attribute.

```
from nepi.execution.resource import ResourceManager, ResourceState

import time

# global variable for the time to sleep on do_deploy and do_start
wait_time = 0

class Link(ResourceManager):
    _rtype = "dummy::Link"

    def do_deploy(self):
        time.sleep(wait_time)
        super(Link, self).do_deploy()

class Interface(ResourceManager):
    _rtype = "dummy::Interface"

    def do_deploy(self):
        node = self.get_connected(Node.get_rtype())[0]
        link = self.get_connected(Link.get_rtype())[0]

        if node.state < ResourceState.READY or \
                link.state < ResourceState.READY:
            self.ec.schedule(self.reschedule_delay, self.deploy)
        else:
            time.sleep(wait_time)
            super(Interface, self).do_deploy()

class Node(ResourceManager):
    _rtype = "dummy::Node"

    def do_deploy(self):
        time.sleep(wait_time)
        super(Node, self).do_deploy()

class Application(ResourceManager):
    _rtype = "dummy::Application"

    def do_deploy(self):
        node = self.get_connected(Node.get_rtype())[0]

        if node.state < ResourceState.READY:
            self.ec.schedule(self.reschedule_delay, self.deploy)
        else:
            time.sleep(wait_time)
            super(Application, self).do_deploy()

    def do_start(self):
        super(Application, self).do_start()
        time.sleep(wait_time)

        # automatically schedule stop method
        self.ec.schedule("0s", self.stop)
```

Listing 3.1 – Implementation of the ResourceManagers for a Dummy platform.

Listing 3.2 shows a more realistic implementation of a *do_deploy* for a *RMA* class that models a resource in an hypothetical platform. Instances of the *RMA* class can be connected to instances of another class *RMB*. In the example, the *do_deploy* method invokes the *do_discover* and *do_provision* methods, which perform the actual deployment actions in the platform. Before invoking these methods, the *do_deploy* checks that the connected *RMB* instance is in state READY. If this is not the case, it requests the Exper-

imentController to reschedule the *deploy* method. The rescheduling logic is a central part of the orchestration algorithm described in Chapter 2. The orchestration internals implemented in NEPI are described in Subsection 3.2.3.

If the connected *RMB* instance is in state FAILED, the *RMA* instance sets itself to FAILED as well by raising an exception. When *do_* prefixed methods raise an exception, NEPI automatically sets the state of the RM to FAILED.

```
def do_deploy(self):
    other_rm = self.get_connected(RMB.get_rtype())[0]

    if other_rm.state < ResourceState.READY:
        self.ec.schedule(self.reschedule_delay, self.deploy)

    elif other_rm.state == ResourceState.FAILED:
        msg = "Failed to deploy resource"
        self.error(msg)
        raise RuntimeError(msg)

    else:
        self.do_discover()
        self.do_provision()

        super(RMA, self).do_deploy()
```

Listing 3.2 – Implementation of the do_deploy method for an hypothetical RMA class.

3.2.2 Generic Network Experiment Programming Interface

The implementation of the Generic Network Experiment Programming Interface is divided into two APIs: the Experiment API and the Resource API. The Experiment API is the one exposed to the user, whereas the Resource API is internal and is used between the ExperimentController and the ResourceManagers. Further documentation for the Experiment and Resource APIs can be found in Appendix B.

3.2.2.1 Experiment API

Table 3.1 lists the main methods of the Experiment API exposed by ExperimentController class. These methods are invoked by the user from a Python script or from a Python console to design and run experiments.

Method	**Description**
ec = ExperimentController(exp_id)	Creates an ExperimentController instance, assigning a user-defined experiment identifier.
guid = register_resource(resource_type)	Creates a RM instance of a given resource_type and returns a global unique identifier (guid) for the RM.
value = get(guid, attribute_name)	Retrieves the value of an attribute of the RM identified by guid.
set(guid, attribute_name, attribute_value)	Modifies the value of an attribute of the RM identified by guid.
register_connection(guid1, guid2)	Marks the RMs identified by guid1 and guid2 as connected to each other.
register_condition(guid, operation, guids, state, time)	Marks the operation *operation* for the RM identified by guid as due to execution time *time* after all RMs in the guids list reached the state *state*.
enable_trace(guid, trace_name)	Activates collection of data associated to the Trace identified by *trace_name*, on the RM identified by guid.
trace(guid, trace_name)	Retrieves information and data of the active Trace identified by *trace_name*, on the RM identified by guid.
deploy(guids=guids, wait_all_ready=False)	Triggers experiment orchestration. If specified, the optional *guids* list restricts deployment to the RMs in the list. The optional *wait_all_ready* flag forces the start operation on any RM in the *guids* list to be invoked after all the RMs in the list reached the state READY.
val = status()	Returns the current status of the experiment run (either FAILED or OK).
wait_finished(guids)	Sets a barrier to block the main execution thread until all RMs in the *guids* list are in state STOPPED.
shutdown()	Releases all RMs and terminates the experiment.

Table 3.1 – Main Experiment API methods implemented by the ExperimentController class.

3.2.2.2 Resource API

Table 3.2 lists the methods exposed by the ResourceManager class. The methods that are associated to life cycle operations, marked in bold, must be implemented by RM classes in order to provide platform specific behavior. The remaining methods have a common implementation and do not need to be re-implemented by RM classes.

Method	Description
value = get(attribute_name)	Retrieves the value of an attribute.
set(attribute_name, attribute_value)	Modifies the value of an attribute.
register_connection(guid)	Marks the RM as connected with another RM identified by guid.
register_condition(operation, guids, state, time)	Marks the operation *operation* for the RM as due to execution time *time* after all RMs in the *guids* list reached the state *state*.
enable_trace(trace_name)	Activates collection of data associated to the Trace identified by *trace_name*.
trace(trace_name)	Retrieves information and data for the active Trace identified by *trace_name*.
deploy()	Prepares the RM for execution. Usually, invoking the discover, reserve, provision, and configure methods.
discover()	Lists platform resources matching criteria specified by the user.
reserve()	Acquires a time slot to access and control a resource in a target platform.
provision()	Allocates a resource in the target platform.
configure()	Applies initial configuration to the resource in the platform.
start()	Activates the resource in the platform.
stop()	Deactivates the resource in the platform.
state()	Retrieves the state of the RM.
release()	Deallocates a resource in the platform.

Table 3.2 – Resource API methods defined by the base ResourceManager class. Methods marked in bold are those that might be implemented by child classes. The remaining methods should not be implemented by child classes.

3.2.3 Orchestration Engine

Invoking the deploy method on an ExperimentController instance triggers the orchestration of the experiment modeled by the user. The deploy method can be invoked passing a group of RMs as optional argument. If a group of RMs is specified, then the deploy method only takes effect on those RMs, otherwise, all RMs in state NEW are deployed. During deployment, the EC schedules tasks to be executed for the deploy, start, and optionally stop, methods of every RM in the deployment group. These tasks are scheduled for immediate execution, i.e., execution time = current time. The stop method for a RM is scheduled only if a stop condition was specified by the user on that RM.

The deploy method of the EC can receive an option *wait_all_ready* flag, with default value *True*. This flags forces the start method of all RMs in a deployment group to be delayed until all RMs are in state READY[2].

The RMs can terminate normally or be forced to terminate when the user decides to end the experiment by invoking the shutdown method on the EC. The invocation of the shutdown method triggers the EC to schedule, for immediate execution, a task with the release method for every deployed RM.

3.2.3.1 The Scheduler

The Scheduler is the component responsible for experiment orchestration. It implements the orchestration algorithm introduced in Chapter 2. The Scheduler runs on a dedicated thread, waiting for Tasks to be available for execution. A Tasks is composed of a method, i.e., an operation, to execute and a scheduled time, i.e., the execution time. The Scheduler uses two queues: a priority queue and a simple queue associated to a pool of worker threads. When the schedule method of the EC is invoked with a method to schedule, e.g., *deploy*, the method gets pushed into the priority queue as a new Task. Tasks are dequeued from the priority queue by earliest scheduled time and order of arrival. Following the terminology used in Chapter 2, the Tasks are the jobs, the priority queue is the priority list, and the workers are the machines.

The processing thread is responsible for identifying tasks that are ready for execution, i.e., current time greater or equal to scheduled time, and inserting those tasks in the simple queue that feeds the pool of workers. For efficiency, instead of busy waiting in a loop, the Scheduler thread uses signals to be awaken when Tasks need to be checked for execution.

Tasks pushed to the workers queue are executed as soon as a worker is available. When a Task's method is executed by a worker, it can reschedule itself by invoking the *schedule* EC method with a *reschedule_time*, as seen in the *do_deploy* example in Listing 3.2. It is the responsibility of the Resource API method implementations, i.e., deploy, start, stop, release, to decide if the method should be rescheduled or not by evaluating the current state of the associated Resources and its internal logic.

Further details about the methods and classes associated to the Scheduler can be found in Section B.5 in Appendix B. The complete implementation of the Scheduler can be found online in the NEPI repository at:

`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/src/nepi/execution`

3.3 Experiment Life Cycle

This section gives an overview on the usage of the Experiment API, showing how the steps of the experiment life cycle introduced in Chapter 2 are supported by the NEPI framework.

[2]This is similar to the *whenAllInstalled* barrier used in OMF [6].

3.3.1 Experiment Design

Figure 3.4 depicts a dummy experiment scenario composed of two RMs, r_1 and r_2, of resource types *A::RM1* and *A::RM2* respectively. r_1 and r_2 are connected to each other, and a workflow condition is added specifying that r_1 must start 5 seconds after r_2 started.

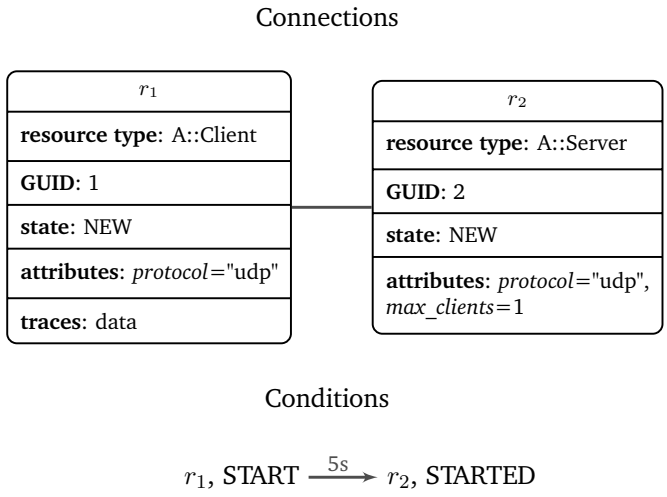


Figure 3.4 – Representation of an Experiment composed of two RMs instances, one modeling a client and the other a server. The client starts 5 seconds after the server.

The scenario in Figure 3.4 is described in Listing 3.3, using the NEPI Experiment API.

```
1  from nepi.execution.resource import ResourceState, ResourceAction
2  from nepi.execution.ec import ExperimentController
3
4  ec = ExperimentController(exp_id="myexp")
5
6  r1 = ec.register_resource("A::Client")
7  ec.set(r1, "protocol", "udp")
8
9  r2 = ec.register_resource("A::Server")
10 ec.set(r2, "protocol", "udp")
11 ec.set(r2, "max_clients", 1)
12
13 ec.register_connection(r1, r2)
14
15 ec.register_condition(r1, ResourceState.START,
16    r2, ResourceAction.STARTED, time="5s")
17
18 ec.enable_trace(r1, "data")
```

Listing 3.3 – Use of experiment design methods to describe the scenario in Figure 3.4.

The *create* operation. The first step in the design of a NEPI experiment is the creation of an ExperimentController instance. The ExperimentController is a Python class that can be imported from the NEPI *execution* module. Creation of the EC is done in line 4 in Listing 3.3.

The *add* operation. Resources are added to the experiment using the *register_resource* method of the EC. This is shown in lines 6 and 9 in Listing 3.3. The *register_resource* method receives a resource type as a string input, and returns an integer global unique identified (guid) that references the newly created RM.

The *set* operation. RMs have a list of attributes, each associated to a string name. As shown in lines 7, 10, and 11 in Listing 3.3, attributes are configured using the *set* method, indicating a RM guid, an attribute name, and the new value.

The *connect* operation. Connections between RMs are described using the *register_connection* method, as shown in line 13 in Listing 3.3. Connections between RMs are symmetrical, so the order in which the RM guids are given to the method is indifferent.

The *condition* operation. User workflows are specified with the *register_condition* method. This method receives a guid, an operation, a list of guids, a resource state, and a time. Line 15 in Listing 3.3 defines a workflow condition between the two RMs that indicates that one RM must start 5s after the other.

The *measure* operation. Traces are activated using the *enable_trace* method, as shown in line 18 in Listing 3.3. Active traces allow to retrieve data during experiment execution. Some traces are active by default.

3.3.2 Experiment Deployment and Execution

Invoking design operations on the ExperimentController does not trigger changes in platform resources or in the experiment execution. Changes in the experiment design take effect only when the deploy method is invoked on the EC. The deploy method can be seen as a commit action that executes design changes in the experiment.

A NEPI experiment is executed from a Python script or console in a *controller host*, which is usually the laptop or desktop of the experimenter. A NEPI experiment is a Python process that holds EC and RM instances. Invoking the EC deploy method instructs these instances to perform actions on platform resources. Listing 3.4 describes the deployment and execution of the experiment modeled in Figure 3.4.

```
1 ec.deploy()
2
3 ec.wait_finished([r2])
4
5 ec.shutdown()
```

Listing 3.4 – Use of experiment deployment and execution methods to run the scenario in Figure 3.4.

The *deploy* operation. The *deploy* method of the EC used in line 1 in Listing 3.4 triggers the orchestration of RMs r_1 and r_2. A list can be passed to the *deploy* method to specify a sub-group of RMs to deploy. If no list is given all RMs in state NEW are deployed.

The *discover*, *reserve*, *provision*, and *configure* operations. When the deploy method of the EC is invoked, it schedules the invocation of the *deploy* method of each RM in the deployment group. The *deploy* method of the RM usually invokes the *discover* and *provision* methods of the RM. The *reserve* and *configure* methods can also be invoked depending on the implementation of the *do_deploy* method of each RM class.

The *start* and *stop* operations. The *start* and *stop* methods of each RM are automatically scheduled by the EC deploy method. The *stop* method is only scheduled automatically if the user defined a stop condition for the RM.

The *wait* operation. The *wait_finished* method in line 3 in Listing 3.4 works as barrier that blocks the execution of the NEPI script until all the RMs given as argument to the wait method have reached the STOPPED state. Other barrier methods are also provided to wait for other RM states, e.g., *wait_deployed*, and *wait_started*. The use of this type of barriers is also found in other network experiment management frameworks and tools [85, 99, 4].

The *shutdown* operation. The *shutdown* method, invoked in line 5 in Listing 3.4, triggers the scheduling of the *release* method for each RM in the EC. The *shutdown* method blocks the execution of the NEPI script until all RMs are released. After the *shutdown* method is invoked it is no longer possible to deploy new RMs or retrieve traces.

The *release* operation. The *release* method of the RMs is part of the Resource API and it is implemented by each RM class independently according to the requirements of the target platform.

3.3.3 Experiment Monitoring and Measuring

Monitoring the state of the experiment and of individual resources, as well as obtaining measurements and configuration information, can be done at any moment during the experiment run. Listing 3.5 shows the use of the monitoring methods of the EC to query the state of RMs, to inspect attributes, and to retrieve data.

```
1  ec.deploy()
2
3  ec.wait_deployed([r1, r2])
4
5  ec.set(r2, "max_clients", 2)
6  print ec.get(r2, "max_clients")
7
8  if ec.state(r1) == ResourceState.STOPPED:
9      print ec.trace(r1, "data")
10
11 ec.shutdown()
12
13 print ec.status()
```

Listing 3.5 – Use of experiment monitoring and measuring operations.

The *get* operation. Information about the configuration of RM instances can be queried using the *get* method of the EC. This method receives a RM guid and an at-

tribute name, and returns the attribute's value, as shown in line 6 in Listing 3.5.

The *state* operation. The state of a RM instance can be queried using the *state* EC method, as shown in line 8 in Listing 3.5. This method receives a guid and returns a numeric state value from the ResourceState enumerate. A string human readable value can be obtained by passing the optional *hr = True* argument to the *state* method.

The *inspect* operation. The *trace* method of the EC receives a RM guid and a trace_name as arguments and returns a stream with data. The *trace* method is used in line 9 in Listing 3.5 to retrieve the content of the "data" trace from the Resource r_1.

The *status* operation. The *status* method of the EC, shown in line 13 in Listing 3.5, is used to query the running state of the EC instance. This method is used internally by the EC to detect whether any fatal failure occurred during the experiment run.

3.4 Support for Platform Uses

One objective of this work is to provide support for diverse ways of combining platforms in a study, namely, single-platform, cross-platform, and multi-platform evaluation. This section discusses how the NEPI framework provides support for these three cases.

3.4.1 Single-Platform Evaluation

Single-platform evaluation is the basic way of conducting a study, where only one platform is involved in the experimentation. The previous sections showed how to model experiments in NEPI using RMs from a same platform. The added value of NEPI with respect to other experiment automation tools is that it supports single-platform evaluation in a variety of platforms, including simulators, emulators, and testbeds.

3.4.2 Cross-Platform Evaluation

Cross-platform evaluation requires interconnecting resources from different platforms to use them at the same time in a single experiment, i.e., hybrid experiment scenario. Having access to a large number and variety of resources and being able to combine those resources in an experiment gives more flexibility to the type of scenarios that can be modeled. Instead of limiting experiments to using resources from a single platform, scenarios can be constructed to combine simulated, emulated, and live resources. In doing this, experimenters can decide the degree of controllability and realism that different parts of the modeled network should have.

In order to run cross-platform experiments, resources in the different platforms must be able to communicate, directly or indirectly. The main challenge of cross-platform evaluation is that most platforms do not support resource interconnection natively. Some platforms can be interconnected through the Internet, either directly or using

tunnels and virtual private networks, others might require more crafty solutions such as adding specific software to plug resources together, or they might even require infrastructure modification to support interconnection. NEPI's ability to support cross-platform evaluation is limited to the cases where platforms can be connected without the need for infrastructure modifications. NEPI does provide a solution to interconnect platforms that are completely isolated.

The strategy offered by NEPI to support cross-platform evaluation consists in creating new RM classes to provide the software artifacts to glue platform resources together. For instance, two private testbeds might be unable to communicate directly through the Internet due to firewall restrictions, but adding a tunnel over the Internet might allow them to communicate transparently without affecting the security of the testbeds. Existing tunneling software can be used or new special purpose tunnels software can be implemented, and then managed using a custom NEPI RM class. Since RM implementations have no restrictions, any lower level communication and synchronization mechanisms are allowed.

Concerning experiment design, RMs on different platforms are interconnected using the same *register_connection* method used for same-platform RMs. Connection can be direct or through additional RM instances, e.g., RMs modeling tunnels, virtual links, pipes. Listing 3.6 shows an hypothetical scenario where two host RMs on platforms A and B are interconnected using a dedicated connector RM that implements traffic exchange mechanism between the two platforms.

```
1  from nepi.execution.ec import ExperimentController
2
3  ec = ExperimentController(exp_id="myexp")
4
5  r1 = ec.register_resource("A::Host")
6  r2 = ec.register_resource("B::Host")
7  connector = ec.register_resource("A::B::Connector")
8
9  ec.register_connection(r1, connector)
10 ec.register_connection(r2, connector)
```

Listing 3.6 – Interconnection of resources from different platforms.

3.4.3 Multi-Platform Evaluation

Multi-platform evaluation allows to obtain complementary results by replicating a same experiment independently on different platforms. It can be used for iterative software development [100], by incrementally adding realism to the development environment used to implement a networking software as it evolves in time, or for result cross-validation, by comparing results from independent platforms for a same experiment scenario.

The design and development of networking software that aims at being deployed on real production systems requires thorough understanding and validation of the software behavior and its interactions with the protocol layers and the network infrastructure. A protocol or application tested initially on one platform can be further studied on other platforms in order to verify that results are not biased by the specificities of the initial platform. Using multiple platforms permits to obtain a better understanding of

the behavior of networking software under different traffic conditions or for different networking technologies.

Production-like development environments are useful to build and test robust networking software, specially in the late development stages. However, realistic live environments are more time consuming and difficult to control and debug than simulated or emulated environments. Iterating between simulated, emulated, and live environments during the development process can help to simplify software debugging and to shorten the development time.

Replicating an experiment on different platforms is seldom simple. It requires mapping the experiment deployment and execution scripts and programs from one platform to the other. To do this it is necessary to master the internals and programming languages used by each platform. Thanks to its uniform model to represent experiments, NEPI eases to some extent the burden of translating experiments from one platform to another. Nevertheless, it does not eliminate the experimenter completely from the design mapping process. The experimenter's judgment is needed to decide what RMs and configurations are equivalent between platforms. As Guruprasad et al. noted, the problem of network testbed mapping is a NP-Hard problem [52]. The problem of experiment description mapping can be thought of as a similar problem.

NEPI attempts to simplify experiment mapping by providing automatic topology generation and experiment translation capabilities as part of the ExperimentController. To use the automatic topology generation capabilities of the EC, users define the mapping functions *add_node* and *add_edge*. Listing 3.7 shows an example of automated topology generation using the ExperimentController. The constructor of the EC can receive two optional topology generation arguments: the *topo_type* and *node_count*. These arguments determine the shape and size of the experiment topology to be generated automatically, i.e., number of RMs and how they are connected. Alternatively, a *topology* argument can be given to the EC, containing an ad-hoc topology shape modeled with a networkx[3] graph.

The functions *add_node* and *add_edge* are implemented by the user. The *add_node* function is invoked once per node in the topology graph and is responsible for adding the RMs to model the nodes in the experiment. The *add_edge* function is invoked once per edge in the topology graph and is responsible for connecting the RMs created by the *add_node* function. By providing different implementation of these functions, the user can customize a NEPI script for multiple platforms.

```
1 from nepi.execution.ec import ExperimentController
2 from nepi.util.netgraph import TopologyType
3
4 ec = ExperimentController(exp_id="linear_topo",
5         topo_type=TopologyType.MESH,
6         node_count=6,
7         add_node_callback=add_node,
8         add_edge_callback=add_edge)
```

Listing 3.7 – Automated topology generation using the *topo_type* and *node_count* arguments of the ExperimentController, and the user-defined *add_node* and *add_edge* functions.

[3]networkx is a Python library to create and manipulate graphs, available at `https://networkx.github.io`.

3.5 Rigorous Experimentation Support

In order to support a rigorous experimentation methodology, NEPI provides result validation, experiment reproducibility, and data archiving functionality.

Result validation is supported in NEPI in different ways, through systematic experimentation, interactive experimentation, multi-platform evaluation, and batch execution. Systematic experimentation consists in making small changes on the experiment description, for instance to sweep a range of values for a scenario parameter. Interactive experimentation allows users to dynamically explore a platform or scenario and directly study their behavior. Multi-platform evaluation permits to corroborate or discard conclusions by contrasting results across platforms. Batch execution is used to automatically re-run a same experiment multiple times in order to collect sufficient data to perform statistical analysis of results. In all cases, it is important to identify and discard invalid experiment runs to prevent invalid data from polluting the results of a study. To discard invalid experiment runs NEPI allows users to define failure policies.

Experiment reproducibility is supported through synthetic description of experiments and automatic orchestration of those descriptions. Several network experiment orchestration tools support synthetic experiment description through the use of well defined experiment modeling languages and orchestration mechanisms [6, 16, 63, 65]. NEPI adapts these principles to span across platforms through its generic experiment model and orchestration mechanisms. Experiments in NEPI can be reproduced by re-running a experiment script or by loading a persistent XML description of an experiment to a EC instance.

NEPI also provides automated data collection and archiving functionality. Data analysis and visualization are not directly incorporated as features of the framework, but users can integrate them into the experiment life cycle by adding their analysis and visualization functions to the NEPI script, using the numpy and matplotlib Python libraries.

This section describes the functionality provided by NEPI to help researchers to enforce a rigorous experimentation procedure.

3.5.1 Interactive Experimentation: Dynamic Deployment

Dynamically deploying RMs in a NEPI experiment permits to modify experiments on real time and to explore platforms or scenarios interactively. Interactive experimentation can help to explore platform features or the parameters of a scenario during the experiment conception stage. It can also help students to get familiar with a platform. Once a scenario is defined, interactive experimentation can be further employed to test and debug an experiment by dynamically reproducing error conditions. Platform administrators can take advantage of dynamic deployment to interactively test platform software and infrastructure, or to monitor the platform and react to problems or abnormal situations.

NEPI supports interactive experimentation through incremental RM deployment and by direct re-configuration of resources using the get and set methods. An experiment

can be dynamically constructed by incrementally deploying and executing groups of RMs, and by dynamically inspecting and retrieving results during run time.

NEPI can be used as a reactive experiment controller capable of adapting scenarios to events in an experiment. For instance, it can be used to deploy a new resource if a failure was detected in an already deployed RM, or to change traffic generation patterns according to the saturation of a channel in an experiment. Listing 3.8 shows a script that uses dynamic deployment to add a new client RM in case the data collected by the first client RM is not correct.

```
ec = ExperimentController(exp_id="dynamic_deployment")

r1 = ec.register_resource("A::Client")
ec.set(r1, "protocol", "udp")
ec.enable_trace(r1, "data")

r2 = ec.register_resource("A::Server")
ec.set(r2, "protocol", "udp")
ec.set(r2, "max_clients", 1)

ec.register_connection(r1, r2)

ec.deploy()

ec.wait_started([r1])

if ec.trace(r1, "data") == "bad data":
    r3 = ec.register_resource("A::Client")
    ec.set(r3, "protocol", "udp")
    ec.register_connection(r3, r2)

    ec.deploy([r3])
```

Listing 3.8 – Interactive experimentation example.

Interactive experimentation can be a very powerful feature, however support for it in NEPI has limitations. Not all platforms allow interactivity, and some, in particular simulators, might not be capable of modifying an experiment after it has started or even to collect traces dynamically during runtime.

3.5.2 Batch Experiment Execution: The ExperimentRunner

Network measurements are subject to variability due to the underlying stochastic processes that affect network behavior, e.g., queuing delays in the network, varying traffic patterns, etc. One way of neutralizing the variability is to use a fully controlled platform, such as simulators, that provides perfect repeatability. But using fully controlled environments is not always possible, nor desirable because it can eliminate realism in the experimentation. Another alternative is to use statistical methods to analyse results and estimate a metric of interest. This requires collecting many measurement samples by independently re-running a same experiment several times.

A same scenario can be reproduced many times by re-executing a NEPI script. However, to efficiently conduct a network study it is best to re-run an experiment the minimum number of times needed to obtain statistically meaningful results. NEPI provides the *ExperimentRunner* (ER) to supervise the re-run of an experiment scenario. Given an experiment description, the ER can automatically re-run an experiment until a stop condition defined by the user occurs.

```
from nepi.execution.runner import ExperimentRunner

rnr = ExperimentRunner()

runs = rnr.run(ec, min_runs=10, max_runs=300,
       compute_metric_callback=metric_function,
       evaluate_convergence_callback=convergence_function,
       wait_guids=rms_list)
```

Listing 3.9 – ExperimentRunner usage example.

Listing 3.9 shows the instantiation and use of an ER to re-run an experiment. The ER receives an EC instance with an experiment description defined by the user. It also receives parameters for the minimum and maximum number of times to re-run the experiment. The two functions *compute_metric_callback* and *evaluate_convergence_callback* are defined by the user. These functions are used to decide when to interrupt the experiment re-run. The ER can receive a list of RM GUIDS, i.e., *wait_guids*, to customize the behavior of each run. If a *wait_guids* list is provided, the ER will wait until all RMs in the list are in state STOPPED before proceeding to the next run. The ExperimentRunner is documented in Appendix B.3.

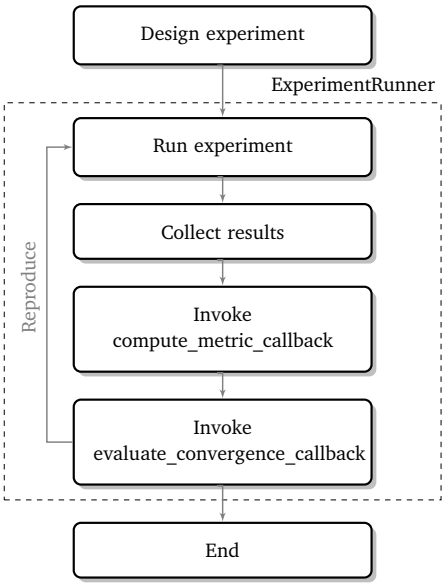


Figure 3.5 – Execution flow of the ExperimentRunner.

Figure 3.5 shows the execution flow used by the ER to re-run an experiment. The user first designs the experiment by constructing an EC object and passes it to the ER. The ER clones the original EC and runs the experiment. When the run finishes, results are collected and the ER invokes the *compute_metric_callback* function to calculate a metric based on the collected results. The metric computed by the function is internally stored by the ER. After each run, the ER passes all previously computed metrics to the *evaluate_convergence_callback* function, which uses the information to determine if the sequence of re-runs should continue or not. The *evaluate_convergence_callback* also takes into account the minimum and maximum number of runs specified by the user.

The data collected from each run is archived in the host that runs the NEPI script.

3.5.3 Error Detection and Failure Policies: The FailureManager

Early error detection is important in experiment automation to discard invalid results and to avoid wasting time on faulty experiment runs. However, some errors or unexpected behaviors might occur without affecting the outcome of an experiment. For example, in an experiment where a group of hosts randomly generate traffic the experiment results might be valid even if one of the hosts failed to send traffic. Discarding experiments with non relevant failures might result in longer times to conduct a study. For this reason it is necessary to detect when resources in an experiment are not behaving as expected, but also to define how the experiment should react to different failures.

To address this, NEPI allows to define failure strategies through a FailureManager object. The FailureManager object implements the failure policies and keeps track of the experiment's failure state. Users can implement a FailureManager and pass it to the ExperimentController upon construction.

When a critical error occurs in a RM, the RM is responsible of invoking the *inform_failure* method on the EC to declare the failure. Invoking this method forces the FailureManager to evaluate the global state of the experiment and decide whether the experiment should be aborted by setting a abort flag in the EC. The behavior of the default FailureManager is to abort the experiment when a critical RM reports a failure, but this behavior can be modified by the experimenter by providing a new FailureManager that implements a different failure policy and passing it to the EC. User-defined failure policies are also supported by other experiment automation tools [4].

Critical RMs are those whose failure can not be ignored, whereas non-critical RMs are those that can fail without affecting the outcome of the experiment. Users can mark RMs as critical or non-critical by setting the *critical* attribute. This attribute takes a boolean true or false value, and it is inherited by all RM classes from the base ResourceManager class. By default all RMs are critical unless the user configures them as non-critical. Listing 3.10 shows the configuration of two RMs, one as critical and the other non critical.

```
1 ec.set(r2, "critical", False)
2 ec.set(r1, "critical", True) # default value
```

Listing 3.10 – Usage of the 'critical' attribute to define critical RMs.

3.5.4 Experiment Re-use: Saving, loading, and plotting

NEPI provides methods to persist a scenario description in human readable form to a local file. The save method of the EC can be used to persist the experiment description in XML format. The load and plot methods can be used to recreate an EC from a previously saved XML experiment description and to visualize the experiment topology. Listing 3.11 shows the use of the save, load, and plot methods. In line 1, an existing EC object is persisted in human readable XML format to a local file, using the save method. The XML description is saved to a file and can be kept as experiment documentation. In line

2, the load method is used to re-create an EC with the previously persisted experiment description. In lines 6 and 7 the plot method is used to visualize the experiment topology and to persist the plot using different formats.

```
1 filename = ec.save()
2 ec2 = ExperimentController.load(filepath = filename)
3
4 from nepi.util.plotter import PFormats
5
6 filename = ec.plot()
7 filename = ec.plot(format = PFormats.DOT)
```

Listing 3.11 – Experiment Saving, Loading, and Plotting methods.

3.5.5 Data Archiving: The Collector

An important part of experimentation is generating and collecting results. NEPI exposes a simple interface to store and organize experiment results through the Collector RM class. An instance of a Collector RM is used like any other RM instance. A Collector is responsible for automatically downloading selected traces upon experiment completion, and archiving the collected data in the machine where the NEPI script is executed.

Trace collection is automatically triggered when the shutdown method is invoked on the EC. The data retrieved from each trace associated to a Collector is individually stored in a separate file in an archive directory. By default the following path is used for archiving traces: *NEPI_HOME/experiment_id/run_id/sub_dir/guid.trace_name*. The *NEPI_HOME* directory is usually a *.nepi* directory inside the user's home directory. The *experiment_id* is the identifier given to the EC by the user upon creation. The *run_id* is the timestamp generated by NEPI to identify an experiment run. The *sub_dir* is an optional subdirectory given by the user. The *guid* is the identifier of the RM that generated the collected data.

```
r1 = ec.register_resource("A::Client")
r3 = ec.register_resource("A::Client")

collector = ec.register_resource("Collector")
ec.set(collector, "traceName", "data")
ec.set(collector, "subDir", "clients")
ec.set(collector, "rename", "data.log")
ec.register_connection(collector, r1)
ec.register_connection(collector, r3)
```

Listing 3.12 – Usage of the Collector RM to automate collection and archiving of the 'data' trace of RM orbjects of resource type "A::Client".

Listing 3.12 shows the creation and configuration of a Collector RM. A Collector RM must be associated to a single trace name, but it can be connected to many RM instances. In the example, a Collector RM is connected to two RM instances of type *A::Client*, and configured to archive their *data* traces in the "*clients*" folder.

Chapter 4

Framework Evaluation: Extensibility

The NEPI framework models and controls platform resources through ResourceManager (RM) objects, which are instances of ResourceManager classes. The ability of NEPI to manage arbitrary resources, on potentially any platform, is based on the extensibility of the ResourceManager class hierarchy. The ResourceManager class hierarchy is rooted on the abstract *ResourceManager class* described in Chapter 3, which can be specialized to manage arbitrary resources. This class exposes the Resource API and serves as a template to add platform specific implementations for the API methods, e.g., deploy, start, stop.

This chapter provides an overview of the platforms that are currently supported in NEPI, and shows how the proposed architecture, based on the ResourceManager class hierarchy, can be extended to manage resources in different types of platforms, including simulators, emulators, and testbeds.

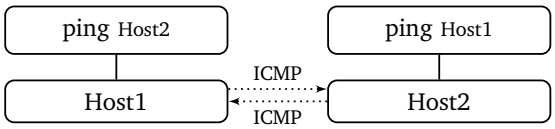


Figure 4.1 – Experiment scenario: Two nodes exchange ICMP messages to measure peer wise round trip time (RTT) and packet loss, using the Ping application.

Figure 4.1 depicts a scenario that consists on measuring the peer wise round trip time (RTT) and packet loss between two nodes by exchanging ICMP messages. This scenario is used along the chapter to show how a same experiment can be modeled in NEPI using resources from different platforms, producing simulated, emulated, and live versions of the experiment.

An analysis of the cost of extending NEPI to support new platforms and resources is provided at the end of the chapter.

4.1 Live Experimentation

This section presents different platforms for live experimentation currently supported in NEPI. NEPI provides ResourceManagers to model and run network experiments on a variety of live platforms, including Linux/SSH testbeds, PlanetLab testbeds, and OMF testbeds.

4.1.1 Linux/SSH Testbeds

Linux/SSH testbeds provide Linux hosts to conduct experiments. Users interact with these hosts by executing commands via SSH. In NEPI, Linux hosts and other associated resources are modeled and controlled by the Linux ResourceManager family. The RMs in this family use a custom SSH client that executes Bash commands in the hosts. The SSH clients authenticate using an SSH account with key authentication, and are able to execute automatically generated commands over SSH in batch without user interaction.

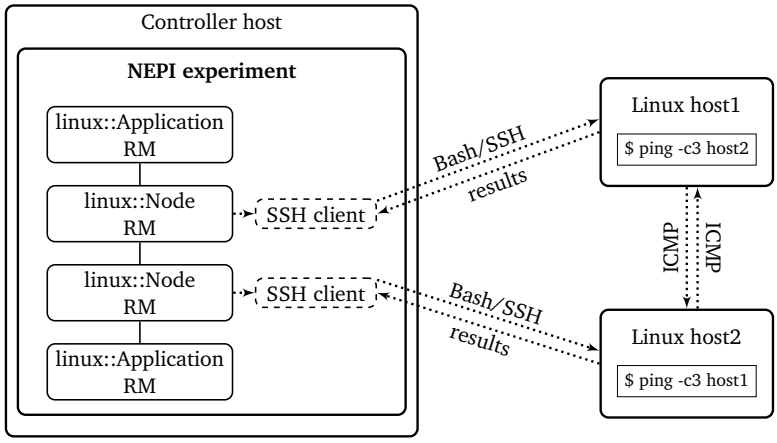


Figure 4.2 – Live experiment scenario using Linux hosts. *linux::Node* and *linux::Applications* RMs are used in NEPI to model and execute an experiment where two Linux hosts exchange ICMP messages. NEPI manages the remote hosts and retrieves results using the NEPI SSH clients.

Figure 4.2 shows the experiment scenario from Figure 4.1 implemented using Linux ResourceManagers. Several Linux ResourceManager classes are available in NEPI to model experiments using Linux resources. The two Linux ResourceManager classes used in the scenario in Figure 4.2 are the *linux::Node* and *linux::Application*. A *linux::Node* represents a Linux host and is associated to a SSH client instance. The SSH account credentials for the client are provided by the user through the RM attributes. A *linux::Application* represents a program, script, or command to be executed in a Linux host using a Bash command. Additionally, many other Linux ResourceManager classes are implemented in NEPI to model other Linux resources, such as TUN/TAP virtual devices, tunnels, and specific network applications such as tcpdump or traceroute.

```
from nepi.execution.ec import ExperimentController

def add_host(ec, hostname, username, ssh_key):
    host = ec.register_resource("linux::Node")
    ec.set(host, "hostname", hostname)
    ec.set(host, "username", username)
    ec.set(host, "identity", ssh_key)
    ec.set(host, "cleanExperiment", True)
    ec.set(host, "cleanProcesses", True)

    return host

def add_ping(ec, host, peer_hostname):
    ping = ec.register_resource("linux::Application")
    ec.set(ping, "depends", "gcc make")
    ec.set(ping, "sources", "http://www.skbuff.net/iputils/iputils-s20101006.tar.bz2")
    ec.set(ping, "build", "tar xvjf ${SRC}/iputils-s20101006.tar.bz2 && cd iputils-s20101006 &&
     make ping")
    ec.set(ping, "install", "cp ping ${BIN}")
    ec.set(ping, "command", "${BIN}/ping -c3 %s" % peer_hostname)
    ec.set(ping, "sudo", True)

    ec.register_connection(host, ping)

    return ping

hostname1 = "host1"
hostname2 = "host2"
username = "myuser"
ssh_key = "~/.ssh/id_rsa"

ec = ExperimentController(exp_id="linux_scenario")

host1 = add_host(ec, hostname1, username, ssh_key)
ping1 = add_ping(ec, host1, hostname2)

host2 = add_host(ec, hostname2, username, ssh_key)
ping2 = add_ping(ec, host2, hostname1)

ec.deploy()

ec.wait_finished([ping1, ping2])

print ec.trace(ping1, "stdout")
print ec.trace(ping2, "stdout")

ec.shutdown()
```

Listing 4.1 – NEPI script to run a live experiment that measures peer wise RTT and packet loss between two Linux hosts, using the ping application.

Listing 4.1 shows a NEPI script to run the scenario in Figure 4.2. Two functions are defined at the begining of the script, one to add a *linux::Node* RM and another to add a *linux::Application* RM. *linux::Node* RMs are configured with the *hostname* of a host and the SSH credentials to access that host. The *cleanExperiment* and *cleanProcesses* attributes are used to ensure a clean host state before experiment run, including the removal of result files from previous runs and the termination of applications and system processes.

linux::Application RMs expose attributes to configure and execute applications in Linux hosts. As shown in function *add_ping*, these RMs allows to specify package dependencies, sources, build and install scripts, as well as the command to start the application. The sources can be files uploaded from the experimenter's machine, i.e., the controller host, or downloaded from a URL, as in the case of the *iputils* sources in the

example. Wildcards, such as ${SRC} and ${BIN}, can be used to indicate paths in the remote host.

Upon deployment, the *do_deploy* method of the *linux::Node* RM class creates a directory structure on the remote Linux host to store application sources and experiment results. The root of this directory structure is located at the *.nepi* folder in the user's home directory on the remote host. Application sources, and other files that might be re-used in different experiments are stored under the *nepi-usr* subdirectory. Examples of files that can be re-used in different experiments are application binaries and application input files, such as media files.

Results, and other files that are specific to one experiment, are stored under the *nepi-exp* subdirectory. An independent directory is created per experiment, resource, and run, at the path *.nepi/nepi-exp/<exp-id>/<guid>/<run-id>*. The *exp-id* is the experiment identifier used when contructing the *ExperimentController* instance, the *guid* is the GUID associated to an RM, and the *run-id* is a timestamp that identifies an experiment run within an experiment identifier. Results generated during experiment execution are stored as files in a Linux host, and their content can be retrieved dynamically during the execution using the *trace* method of the EC.

The advantage of storing results as files in the remote host is that results can be retrieved on demand as many times as necessary, preferably at the end of the experiment to prevent the extra traffic generated by the download of results from interfering with the running experiment.

The *linux::Node* ResourceManager does not implement any particular host discovery or reservation mechanism, so users need to make sure that hosts are known and accessible before the experiment starts. However, it is possible to extend the *linux::Node* RM to automate host resource discovery and provisioning. This can be done by creating a new derived RM class that implements the *do_discover* and *do_provision* methods to interact with specific resource provisioning services, such as the Grid5000 OAR [38] service, or the PlanetLab MyPLC [40] service.

Linux ResourceManager classes can be found in the NEPI project source code at *src/nepi/resources/linux*.

4.1.2 PlanetLab Testbeds

Following the idea of building upon a common ResourceManager class hierarchy to support new resource types or even testbeds, a *planetlab::Node* RM class was created extending the *linux::Node* RM class. The *planetlab::Node* RM re-implements the *do_discover* and *do_provision* methods of the *linux::Node* RM to add discovery and provisioning support for PlanetLab hosts using the MyPLC PLanetLab service. MyPLC exposes an API that can be queried using XMLRPC calls to retrieve a list of hosts matching specific criteria, and to provision virtual machines on those hosts. To support discovery and provisioning through MyPLC, a custom MyPLC client was added to NEPI. The *planelab::Node* RM class uses the MyPLC client to issue XMLRPC calls to the MyPLC service during the discovery and provisioning steps in order to provision hosts for an experiment. Since PlanetLab hosts are Linux hosts, the *planetlab::Node* RM class re-uses the SSH client of

the *linux::Node* to execute Bash commands on the hosts over SSH, after host provisioning.

Figure 4.3 shows the experiment scenario from Figure 4.1 using *planetlab::Node* RMs. Applications are modeled using the same *linux::Application* RMs used with Linux hosts.

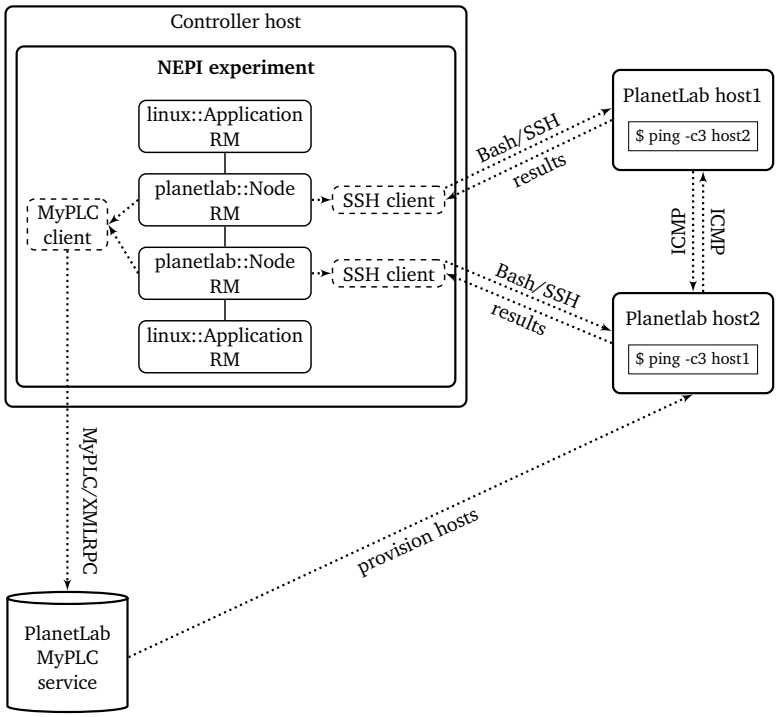


Figure 4.3 – Live experiment scenario using PlanetLab hosts. *planetlab::Node* and *linux::Application* RMs are used to model and execute an experiment where two PlanetLab hosts exchange ICMP messages. NEPI uses a MyPLC client to discover and provision hosts in PlanetLab during deployment, and an SSH client, inherithed from *linux::Node*, to execute commands during experiment execution.

The NEPI script shown in Listing 4.1, using Linux hosts, can be adapted to use PlanetLab hosts by modifying the *add_host* function. Listing 4.2 shows an example function to configure a *planetlab::Node* RM with attributes to set the credentials for the MyPLC service and additional filters to select PlanetLab hosts.

```
1  def add_host(ec, username, ssh_key, pl_user, pl_password):
2      host = ec.register_resource("planetlab::Node")
3      ec.set(host, "username", username)
4      ec.set(host, "identity", ssh_key)
5      ec.set(host, "cleanExperiment", True)
6      ec.set(host, "cleanProcesses", True)
7
8      # MyPLC credentials
9      ec.set(host, "pluser", pl_user)
10     ec.set(host, "plpassword", pl_password)
11
12     # filters
13     ec.set(host, "country", "France")
14     ec.set(host, "operatingSystem", "f12")
```

```
    ec.set(host, "architecture", "x86_64")
    ec.set(host, "minBandwidth", 512)
    ec.set(host, "minCpu", 50)

    return host
```

Listing 4.2 – Example of configuration of a planetlab::Node using filters to specify resource selection criteria.

Upon deployment, NEPI takes care of discovering and provisioning PlanetLab hosts that match the requirements of the user. Listing 4.3 shows a NEPI script adapted to provision PlanetLab hosts and to obtain their hostnames. In this example the applications are deployed only after the hosts are provisioned and their hostnames are known.

```
ec = ExperimentController(exp_id="planetlab_scenario")

host1 = add_host(ec, username, ssh_key, pl_user, pl_password)
host2 = add_host(ec, username, ssh_key, pl_user, pl_password)

ec.deploy([host1, host2])

ec.wait_deployed([host1, host2])

hostname1 = ec.get(host1, "hostname")
hostname2 = ec.get(host2, "hostname")

ping1 = add_ping(ec, host1, hostname2)
ping2 = add_ping(ec, host2, hostname1)

ec.deploy([ping1, ping2])

ec.wait_finished([ping1, ping2])

print ec.trace(ping1, "stdout")
print ec.trace(ping2, "stdout")

ec.shutdown()
```

Listing 4.3 – NEPI script to provision two PlanetLab hosts using the MyPLC service, and execute the ping application on each host.

Malfunctioning PlanetLab hosts can be blacklisted by the experimenter by adding the hostname to the *.nepi/plblacklist.txt* file in the home folder of the host running the NEPI script. PlanetLab ResourceManager classes can be found in the NEPI project source code at *src/nepi/resources/planetlab*.

4.1.3 OMF Testbeds

OMF testbeds supported in NEPI are private wireless networks whose hosts can be managed by sending messages to an XMPP server. The XMPP messages sent to the server contain OMF protocol instructions, which the XMPP server delivers to the testbed hosts using a *publish/subscribe* mechanism. To communicate with the XMPP server and control experiments in OMF testbeds, NEPI implements an XMPP client based on the Python SleekXMPP library[1]. This client is used by the *omf::Node* RMs that model OMF hosts to send OMF instructions using the XMPP protocol.

Figure 4.4 shows the experiment scenario from Figure 4.1 using OMF ResourceManagers.

[1]The Python SleekXMPP library is available at `https://pypi.python.org/pypi/sleekxmpp`.

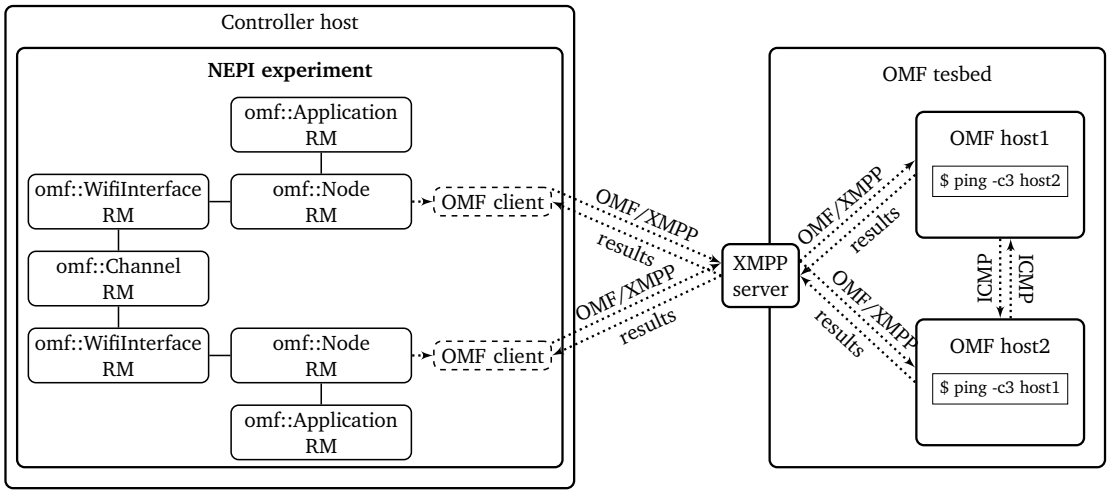


Figure 4.4 – Live experiment scenario using OMF hosts. *omf::Node* and *omf::Application* RMs are used in NEPI to model and execute an experiment where two OMF hosts exchange ICMP messages over a wireless medium. NEPI manages the OMF hosts and retrieves results by sending OMF instructions over XMPP.

```
from nepi.execution.ec import ExperimentController
from nepi.execution.resource import ResourceAction, ResourceState

def add_host(ec, hostname, username, password):
    host = ec.register_resource("omf::Node")
    ec.set(host, "hostname", hostname)
    ec.set(host, "xmppUser", username)
    ec.set(host, "xmppPassword", password)
    ec.set(host, "xmppServer", "nitlab.inf.uth.gr")
    ec.set(host, "xmppPort", "5222")

    return host

def add_iface(ec, host, ip, prefixlen):
    iface = ec.register_resource("omf::WifiInterface")
    ec.set(iface, "name", "wlan0")
    ec.set(iface, "mode", "adhoc")
    ec.set(iface, "hw_mode", "g")
    ec.set(iface, "essid", "ping")
    ec.set(iface, "ip", "%s/%s" % (ip, prefixlen))

    ec.register_connection(host, iface)

    return iface

def add_channel(ec, chan_number, username, password, ifaces):
    channel = ec.register_resource("omf::Channel")
    ec.set(channel, "channel", chan_number)
    ec.set(channel, "xmppUser", username)
    ec.set(channel, "xmppPassword", password)
    ec.set(channel, "xmppServer", "nitlab.inf.uth.gr")
    ec.set(channel, "xmppPort", "5222")

    for iface in ifaces:
        ec.register_connection(channel, iface)

    return channel

def add_ping(ec, host, peer_hostname):
    ping = ec.register_resource("omf::Application")
```

```
41      ec.set(ping, "appid", "Ping#1")
42      ec.set(ping, "command", "/bin/ping -c3 %s" % peer_hostname)
43
44      ec.register_connection(ping, host)
45
46      ec.register_condition(ping, ResourceAction.STOP, ping,
47          ResourceState.STARTED , "30s")
48
49      return ping
50
51  hostname1 = "hostname1"
52  hostname2 = "hostname2"
53  username = "myuser"
54  password = "password"
55  chan_number = "channel"
56
57  ec = ExperimentController(exp_id="omf_scenario")
58
59  host1 = add_host(ec, hostname1, username, password)
60  iface1 = add_iface(ec, host1, "192.168.0.1", "24")
61
62  host2 = add_host(ec, hostname2, username, password)
63  iface2 = add_iface(ec, host2, "192.168.0.2", "24")
64
65  add_channel(ec, chan_number, username, password, [iface1, iface2])
66
67  ping1 = add_ping(ec, host1, hostname2)
68  ping2 = add_ping(ec, host2, hostname1)
69
70  ec.deploy()
71
72  ec.wait_finished([ping1, ping2])
73
74  print ec.trace(ping1, "stdout")
75  print ec.trace(ping2, "stdout")
76
77  ec.shutdown()
```

Listing 4.4 – NEPI script to run a live experiment that measures peer wise RTT and packet loss between two OMF hosts, using the ping application.

Listing 4.4 shows a NEPI script to run the scenario from Figure 4.4. OMF hosts are modeled with *omf::Node* RMs and OMF applications are modeled with *omf::Application* RMs. Unlike the previous examples, in the OMF case, it is necessary to configure the wireless interfaces in the nodes and the wireless channel connecting them. Credentials to authenticate with the XMPP server in the testbed, as well as other resource configuration information, such as the channel number and the mode for the wireless interfaces, are provided as RM attributes. OMF hosts and channels must be manually reserved by the experimenter before the NEPI script can be executed. There is so far no unique OMF standard for host reservation, and different OMF testbeds usually implement their own resource reservation mechanism[2]. It would be possible to extend the *omf::Node* RM to incorporate automatic host discovery, reservation, and provisioning support as it was done for the PlanetLab testbed.

OMF ResourceManager classes can be found in the NEPI project source code at *src/nepi/resources/omf*.

[2]In particular, a proposal for a resource discovery, reservation, and provisioning framework for OMF can be found at `http://omf.mytestbed.net/projects/omf6/wiki/BrokerDesign`.

4.2 Simulation

The ResourceManager model proposed in NEPI can be used to automate simulated experiments as well as live experiments. Currently, the only simulation platform supported in NEPI is the ns-3 simulator, however other simulators can also be supported by implementing the corresponding ResourceManager class family.

4.2.1 The ns-3 Simulator

The traditional usage model of the ns-3 simulator consists in defining a network simulation by writing C++ programs that use models provided by ns-3 libraries. The resulting simulation program is then executed in a host without interruption until its termination. Simulations constructed in this way are fully defined before execution and no changes to the simulated scenario can occur during run time. This traditional approach assumes that simulations are self-contained, stand-alone experiments, that do not interact with the experimenter or other external systems.

Nevertheless, ns-3 also supports an extended usage model that allows simulations to interact with the experimenter and with external systems. For instance, ns-3 is able to exchange traffic with live networks using special network device models, such as the TapBridge device or the FdNetDevice[3]. These special ns-3 devices are capable of interfacing with TAP/TUN virtual devices or raw sockets in the host that runs the simulation in order to transparently send and receive network traffic.

ns-3 also incorporates Python bindings to model and run ns-3 simulations[4] from Python scripts instead of from C++ programs. Writing scripts not only simplifies running simulations for non-expert programmers, it also permits to dynamically change a simulation at run time, thanks to the Python interpreter's support for interactive use. The ability to interactively define and start a simulation from a scripting environment, as opposed to the non-interactive pre-compiled binary execution alternative, makes it simpler to control ns-3 simulations from an external orchestration engine such as NEPI. An external orchestration engine can use the ns-3 Python bindings to dynamically define an ns-3 simulation scenario, by adding and interconnecting ns-3 objects on the fly, and then start the simulation when all the parts of the experiment, including any external live resources, are ready.

NEPI leverages the features of the ns-3 simulator to provide automated orchestration of simulations using the same experiment model as for live platforms. In doing so, it also simplifies deployment of ns-3 simulations in multi-host scenarios [101]. Taking advantage of the extended simulation capabilities provided by ns-3, such as interconnecting simulated and live networks, or distributing simulations over many live hosts, requires advanced system administration skills and considerable manual work. This adds a non-negligible cost to the experimentation. NEPI addresses this added cost by acting as an

[3]Documentation on special ns-3 network devices is available at `http://www.nsnam.org/docs/models/html/emulation-overview.html`.

[4]Documentation on Python bindings for ns-3 is available at `http://www.nsnam.org/wiki/Python_bindings`.

abstraction layer for the design and execution of complex ns-3 experiments, relieving experimenters from complex manual tasks.

Adding support for ns-3 simulations in NEPI consisted in two parts: the creation of a family of ns-3 ResourceManagers to model and control ns-3 resources, such as ns-3 nodes, devices, and applications, and the implementation of an ns-3 client/server architecture to control remote simulation instances interactively.

Figure 4.5 shows the experiment scenario from Figure 4.1 using ns-3 ResourceManagers. A NEPI experiment involving ns-3 resources is modeled in a similar way than a live experiment. A particularity of the description of ns-3 experiments in NEPI is that a RM representing an ns-3 simulation instance must be associated to a physical host RM. This host can be the localhost, or a remote host as shown in the example. Simulations are executed as independent processes that behave as servers and can be controlled from a custom ns-3 client. To control the simulation instance, the client sends custom messages with instructions to a socket attached to the server. The ns-3 server architecture is designed to receive instructions interactively, which means that an ns-3 simulation can be incrementally described and modified.

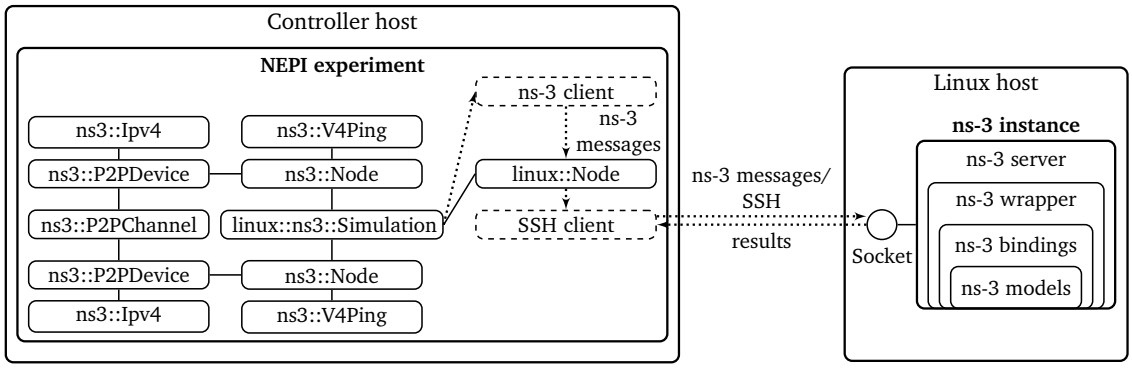


Figure 4.5 – Simulated experiment scenario using ns-3 nodes. *ns3::Node* and *ns3::V4Ping* RMs are used to model and execute an experiment where two ns-3 nodes exchange ICMP messages over a simulated point to point channel. An ns-3 simulation instance runs as a server in a remote Linux host, and NEPI sends custom messages over SSH to the ns-3 server to control the simulation.

4.2.1.1 The Interactive ns-3 Server

The ns-3 server design is based on a layered architecture as shown in Figure 4.6. There are four well defined and fairly independent layers: the ns-3 models layer, the ns-3 Python bindings layer, the ns-3 wrapper layer, and the ns-3 server layer. The two inner layers consist of libraries provided by ns-3 and deal with exposing ns-3 C++ models as Python objects, whereas the two outter layers deal with exposing ns-3 simulations as interactive services that can be managed remotely.

The ns-3 server architecture could be used independently of NEPI. For this, a stand alone ns-3 client would need to be implemented that is able to receive instructions directly from the user and send them to the server. The instructions the NEPI ns-3 client

sends to the server are automatically generated by the ns-3 ResourceManagers based on the scenario modeled by the experimenter. The ns-3 RMs also take care of compiling and installing ns-3 binaries in the Linux host and launching the ns-3 server.

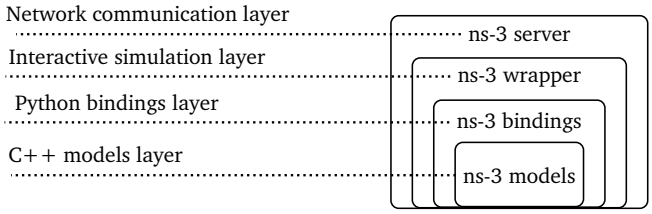


Figure 4.6 – The ns-3 server layered architecture. The server architecture is divided into four layers: the ns-3 models layer contains the C++ objects used in the ns-3 simulation, the Python binding layer contains the bindings to expose ns-3 objects as Python objects, the ns-3 wrapper layer provides the logic to make ns-3 simulations interactive, and the ns-3 server layer implements the communication mechanisms to receive instructions from a client and send back data.

The ns-3 Server Layer

The ns-3 server layer implements the communication mechanism to exchange messages and data with an ns-3 client. The current implementation of the ns-3 server[5] layer waits for incoming messages from an ns-3 client on a local Unix socket. The ns-3 client[6] connects to the Unix socket, either from a local process or from a remote host, and sends messages with instructions to control the simulation.

The ns-3 client communicates with the ns-3 server using a custom protocol that includes the messages in Table 4.1. These messages can be invoked independently one from another. The *CREATE*, *FACTORY*, *GET*, *SET*, and *INVOKE* messages are used to define and configure a simulation. The *START*, *STOP*, and *SHUTDOWN* methods are used to control the simulation execution. The *FLUSH* method is used to flush the standard error and standard output streams of the simulation process, so that data that is buffered in memory can be persisted to a file. Data generated by an ns-3 simulation is stored on local files in the Linux host. These files can be retrieved as a streams, i.e., traces, by the ns-3 client and archived in the host that runs the NEPI script.

The only ns-3 client currently implemented in NEPI connects to the Unix socket on a remote Linux host using SSH to send instructions to the ns-3 server. Other ns-3 client and ns-3 server layer implementations could be provided to support different communication mechanisms, without affecting the inner layers of the server architecture. For instance, a new ns-3 client and ns-3 server layer could be implemented to support the publish/subscribe communication scheme of OMF testbeds. An OMF ns-3 client would send the messages in Table 4.1 encapsulated in OMF protocol messages over XMPP. An

[5]The ns-3 server code is available at
`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/src/nepi/resources/ns3/ns3server.py`.

[6]The ns-3 client code to connect from a Linux host to an ns-3 server is available at
`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/src/nepi/resources/linux/ns3/ns3client.py`.

Message	Description
CREATE	Instantiate an ns-3 object using a constructor, without configuring the object.
FACTORY	Instantiate an ns-3 object using the ns-3 factory system, pre-setting the configuration of the object.
GET	Get the value of an ns-3 object attribute.
SET	Set the value of an ns-3 object attribute.
INVOKE	Invoke a method on an ns-3 object.
START	Start the ns-3 simulation.
STOP	Stop the ns-3 simulation.
SHUTDOWN	Stop the ns-3 server.
FLUSH	Flush standard error and standard output streams for the ns-3 server process.

Table 4.1 – Custom protocol messages used by the ns-3 client and server.

OMF-compatible ns-3 server layer would listen for messages from the XMPP server and send data back to the OMF ns-3 client using the publish/subscribe scheme. OMF ResourceManagers, used to model and control ns-3 simulations in OMF hosts from a NEPI script, would use the OMF ns-3 client to communicate with the ns-3 server.

The ns-3 Wrapper Layer

An ns-3 server instance internally holds an ns-3 wrapper instance. When a message arrives to the server with an instruction from a ns-3 client, the server passes the instruction to the wrapper. The wrapper is responsible for performing the actions needed to execute the instruction and returning the output of the instruction to the server. The server then sends back a reply to the client with the result of the instruction.

The ns-3 wrapper layer is in charge of implementing interactive control of ns-3 simulations. The ns-3 wrapper uses two threads, one to run the simulation and the other to execute instructions received from the client[7]. The wrapper exposes an API to the ns-3 server with methods that correspond to the ns-3 protocol messages from Table 4.1.

The ability of the ns-3 wrapper to interactively modify an ns-3 simulation is based on the introspection capabilities provided by Python. Python introspection allows to dynamically execute methods on ns-3 Python objects using string function names and arguments. Using Python introspection, ns-3 messages sent by the ns-3 client, which are composed of text fields, can be transformed into executable instructions.

Listing 4.5 shows how Python introspection can be used to construct ns-3 objects and invoke methods on them using string method names. ns-3 Python modules for network and application models are imported in lines 1 and 2. In line 4, the Python *getattr* method is used to obtain the constructor of an ns-3 Node object from its string

[7]The ns-3 wrapper code is available at
`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/src/nepi/resources/ns3/ns3wrapper.py`.

name. Line 5 invokes the constructor method and creates the ns-3 Node object. The same procedure is repeated in lines 7 and 8 to instantiate an ns-3 V4Ping object. In line 10, the *getattr* method is used again, providing a string method name to retrieve the *AddApplication* method of the ns-3 Node object. Finally, line 11 invokes the *AddApplication* method on the ns-3 Node passing the V4Ping object as argument.

```
1  import ns.network
2  import ns.applications
3
4  node_constructor = getattr(ns.network, "Node")
5  node = node_constructor()
6
7  app_constructor = getattr(ns.applications, "V4Ping")
8  app = app_constructor()
9
10 method = getattr(node, "AddApplication")
11 method(app)
```

Listing 4.5 – Usage of Python introspection to construct ns-3 objects and invoke methods from strings method names.

The ns-3 wrapper keeps references to all ns-3 objects instantiated in a simulation. To allow remote manipulation of these objects from the ns-3 client, the wrapper assigns a unique identifier to each object upon creation and returns it to the client. The client can use the identifier to invoke methods on the objects later on.

Since debugging dynamically generated distributed simulations is difficult, the ns-3 wrapper supports a debug mode. When configured in debug mode, the ns-3 wrapper records all received instructions to a Python script, serializing the instructions in the same order they are received by the wrapper. The script can then be executed as a regular Python script to replay the recorded experiment in a non-interactive way. The serialization of the experiment makes it easier to inspect the execution step by step using pdb, i.e., Python debug library. However, this does not allow to capture actions that happened outside the scope of the wrapper, such as the creation of virtual TAP devices or Linux applications in a host. For this reason, the debug mode of the ns-3 wrapper is only useful to debug the part of an experiment that is seen by a single ns-3 wrapper.

4.2.1.2 The ns-3 ResourceManagers

There is roughly one ns-3 ResourceManager class available in NEPI for each model supported in ns-3. For example, the ns3::Node model, ns3::V4Ping model, ns3::Ipv4L3Protocol model, all have individual RM classes. This detailed modeling granularity responds to the choice of exposing to the NEPI user the same experiment design granularity provided by each platform. The generation of most ns-3 ResourceManager classes in NEPI can be automated by running a Python script[8] that uses introspection to obtain information about ns-3 models and create the NEPI code for the RM classes. Except for a few base ns-3 ResourceManagers, which were implemented manually, the majority of the ns-3 RMs provided by NEPI were automatically generated. This is possible because most

[8]The script to generate ns-3 ResourceManagers is available at
`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/src/nepi/resources/ns3/resource_manager_generator.py`.

ns-3 models belong to a few families of models, e.g., applications, protocols, devices, that share a similar behavior and differ externally only in their configuration attributes.

Of all the ns-3 RMs in NEPI, the central one is the RM to model ns-3 simulation instances. Since a simulation is executed on a live host, the implementation of a simulation RM is tied to a particular host type. Currently, NEPI only provides a ResourceManager class to model ns-3 instances running on Linux hosts. This RM class corresponds to the resource type *linux::ns3::Simulation*. A *linux::ns3::Simulation* RM extends the *linux::Application* RM class with additional functionally, such as compiling and installing ns-3 on the target Linux hosts and communicating with an ns-3 server instance using an ns-3 client. Like the *linux::Application* RMs, a *linux::ns3::Simulation* RM must be connected to a *linux::Node* RM that models the live host where the simulation instance will run.

The example in Listing 4.6 shows a NEPI script that models the ping scenario from Figure 4.1 using ns-3 Nodes. A protocol stack including ARP, IP, ICMP, UDP, and TCP protocols is added to the ns-3 Nodes. ns-3 Nodes are connected through a point to point link. ICMP traffic is generated using the ns-3 V4Ping model. The structure of this script is fairly similar to the NEPI scripts used to model the ping experiment scenario on live platforms.

```
from nepi.execution.ec import ExperimentController

def add_linux_host(ec, hostname, username, ssh_key):
    host = ec.register_resource("linux::Node")
    ec.set(host, "hostname", hostname)
    ec.set(host, "username", username)
    ec.set(host, "identity", ssh_key)
    ec.set(host, "cleanExperiment", True)
    ec.set(host, "cleanProcesses", True)

    return host

def add_node(ec, simu):
    node = ec.register_resource("ns3::Node")
    ec.register_connection(node, simu)

    arp = ec.register_resource("ns3::ArpL3Protocol")
    icmp = ec.register_resource("ns3::Icmpv4L4Protocol")
    ipv4 = ec.register_resource("ns3::Ipv4L3Protocol")
    udp = ec.register_resource("ns3::UdpL4Protocol")
    tcp = ec.register_resource("ns3::TcpL4Protocol")

    ec.register_connection(node, arp)
    ec.register_connection(node, icmp)
    ec.register_connection(node, ipv4)
    ec.register_connection(node, udp)
    ec.register_connection(node, tcp)

    return node

def add_iface(ec, node, ip, prefixlen):
    iface = ec.register_resource("ns3::PointToPointNetDevice")
    ec.set(iface, "ip", ip)
    ec.set(iface, "prefix", prefixlen)

    ec.register_connection(node, iface)

    queue = ec.register_resource("ns3::DropTailQueue")
    ec.register_connection(iface, queue)

    return iface

```

```
43 def add_channel(ec, ifaces):
44     channel = ec.register_resource("ns3::PointToPointChannel")
45 
46     for iface in ifaces:
47         ec.register_connection(channel, iface)
48 
49     return channel
50 
51 def add_ping(ec, node, peer_ip):
52     ping = ec.register_resource("ns3::V4Ping")
53     ec.set (ping, "Remote", peer_ip)
54     ec.set (ping, "Interval", "1s")
55     ec.set (ping, "Verbose", True)
56     ec.set (ping, "StartTime", "0s")
57     ec.set (ping, "StopTime", "20s")
58 
59     ec.register_connection(ping, node)
60 
61     return ping
62 
63 hostname = "hostname"
64 username = "myuser"
65 ssh_key = "~/.ssh/id_rsa"
66 
67 ec = ExperimentController(exp_id="ns3_scenario")
68 
69 host = add_linux_host(ec, hostname, username, ssh_key)
70 
71 simu = ec.register_resource("linux::ns3::Simulation")
72 ec.register_connection(simu, host)
73 
74 node1 = add_node(ec, simu)
75 iface1 = add_iface(ec, node1, "192.168.0.1", "24")
76 
77 node2 = add_node(ec, simu)
78 iface2 = add_iface(ec, node2, "192.168.0.2", "24")
79 
80 channel = add_channel(ec, [iface1, iface2])
81 
82 ping1 = add_ping(ec, node1, "192.168.0.2")
83 ping2 = add_ping(ec, node2, "192.168.0.1")
84 
85 ec.deploy()
86 
87 ec.wait_finished([ping1, ping2])
88 
89 print ec.trace(simu, "stdout")
90 
91 ec.shutdown()
```

Listing 4.6 – NEPI script to model a simulated experiment using ns-3. The experiment measures peer wise RTT and packet loss between two ns-3 Nodes using a ping application ns-3 model.

To configure the ns-3 wrapper to run on debug mode, experimenters can use the *enableDump* attribute of the *linux::ns3::Simulation* RMs, as shown in Listing 4.7.

```
1 simu = ec.register_resource("linux::ns3::Simulation")
2 ec.set(simu, "enableDump", True)
```

Listing 4.7 – The *enableDump* attribute in a *linux::ns3::Simulation* RM is used to set the ns-3 wrapper to debug mode.

4.3 Emulation

NEPI currently supports emulation using the Direct Code Execution (DCE) [15] emulation extension for the ns-3 simulator, and the NetNS [102] lightweight emulation

platform based on Linux namespace virtualization.

4.3.1 DCE

Direct Code Execution (DCE) adds to the ns-3 simulator the ability to execute real application binaries, the same binaries that would be executed in live Linux hosts. DCE turns ns-3 into a software emulator capable of running real software inside a synthetic ns-3 network.

Adding support for application emulation in NEPI using DCE required adding a new ResourceManager class, the *linux::ns3::dce::Application* RM. This RM class is used to model the execution of Linux application binaries inside an ns-3 simulated network using DCE. The *linux::ns3::Simulation* RM was also modified to automate installation of DCE libraries in the Linux host used to run the DCE/ns-3 emulation.

Modeling an emulation experiment in NEPI using DCE is almost the same as modeling an ns-3 experiment. Figure 4.7 shows the experiment scenario from Figure 4.1 using ns-3 RMs to model the network and *linux::ns3::dce::Application* RMs to model the ping applications.

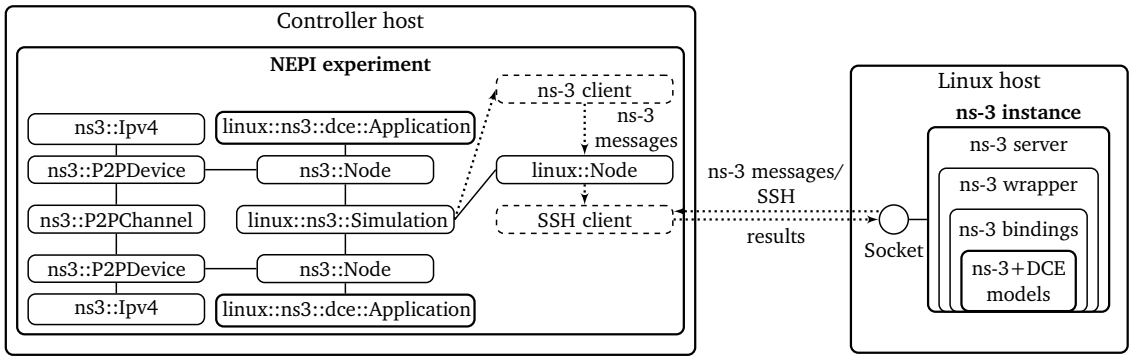


Figure 4.7 – Emulated experiment scenario using ns-3 nodes and DCE applications. *ns3::Node* and *linux::ns3::dce::Applications* RMs are used to model and execute an experiment where two ns-3 nodes exchange ICMP messages over a simulated point to point channel using the real Linux ping application. The simulation instance runs as a server in a remote Linux host and NEPI sends custom messages over SSH to give instructions to the remote ns-3 server.

```
from nepi.execution.ec import ExperimentController

def add_linux_host(ec, hostname, username, ssh_key):
    host = ec.register_resource("linux::Node")
    ec.set(host, "hostname", hostname)
    ec.set(host, "username", username)
    ec.set(host, "identity", ssh_key)
    ec.set(host, "cleanExperiment", True)
    ec.set(host, "cleanProcesses", True)

    return host

def add_node(ec, simu):
    node = ec.register_resource("ns3::Node")
    ec.register_connection(node, simu)

```

```
17      arp = ec.register_resource("ns3::ArpL3Protocol")
18      icmp = ec.register_resource("ns3::Icmpv4L4Protocol")
19      ipv4 = ec.register_resource("ns3::Ipv4L3Protocol")
20      udp = ec.register_resource("ns3::UdpL4Protocol")
21      tcp = ec.register_resource("ns3::TcpL4Protocol")
22
23      ec.register_connection(node, arp)
24      ec.register_connection(node, icmp)
25      ec.register_connection(node, ipv4)
26      ec.register_connection(node, udp)
27      ec.register_connection(node, tcp)
28
29      return node
30
31  def add_iface(ec, node, ip, prefixlen):
32      iface = ec.register_resource("ns3::PointToPointNetDevice")
33      ec.set(iface, "ip", ip)
34      ec.set(iface, "prefix", prefixlen)
35
36      ec.register_connection(node, iface)
37
38      queue = ec.register_resource("ns3::DropTailQueue")
39      ec.register_connection(iface, queue)
40
41      return iface
42
43  def add_channel(ec, ifaces):
44      channel = ec.register_resource("ns3::PointToPointChannel")
45
46      for iface in ifaces:
47          ec.register_connection(channel, iface)
48
49      return channel
50
51  def add_ping(ec, node, peer_ip):
52      ping = ec.register_resource("linux::ns3::dce::Application")
53      ec.set (ping, "sources",
54          "http://www.skbuff.net/iputils/iputils-s20101006.tar.bz2")
55      ec.set (ping, "build", "tar xvjf ${SRC}/iputils-s20101006.tar.bz2 && "
56          "cd iputils-s20101006/ && "
57          "sed -i 's/CFLAGS=/CFLAGS+=/g' Makefile && "
58          "make CFLAGS=-fPIC LDFLAGS='-pie -rdynamic' ping && "
59          "cp ping ${BIN_DCE} && cd - ")
60      ec.set (ping, "binary", "ping")
61      ec.set (ping, "stackSize", 1<<20)
62      ec.set (ping, "arguments", "-c 10;-s 1000;%s" % peer_ip)
63      ec.set (ping, "StartTime", "1s")
64      ec.set (ping, "StopTime", "20s")
65
66      ec.register_connection(ping, node)
67
68      return ping
69
70  hostname = "hostname"
71  username = "myuser"
72  ssh_key = "~/.ssh/id_rsa"
73
74  ec = ExperimentController(exp_id="dce_scenario")
75
76  host = add_linux_host(ec, hostname, username, ssh_key)
77
78  simu = ec.register_resource("linux::ns3::Simulation")
79  ec.register_connection(simu, host)
80
81  node1 = add_node(ec, simu)
82  iface1 = add_iface(ec, node1, "192.168.0.1", "24")
83
84  node2 = add_node(ec, simu)
85  iface2 = add_iface(ec, node2, "192.168.0.2", "24")
86
87  channel = add_channel(ec, [iface1, iface2])
88
89  ping1 = add_ping(ec, node1, "192.168.0.2")
90  ping2 = add_ping(ec, node2, "192.168.0.1")
```

```
91
92 ec.deploy()
93
94 ec.wait_finished([ping1, ping2])
95
96 print ec.trace(ping1, "stdout")
97 print ec.trace(ping2, "stdout")
98
99 ec.shutdown()
```

Listing 4.8 – NEPI script to run an emulated experiment that measures peer wise RTT and packet loss between two ns-3 Nodes, using the Linux ping application executed inside the simulation with DCE.

The NEPI script to run the ping experiment scenario from Figure 4.1 using DCE applications is almost the same as the one shown in Listing 4.6. The main differences are in the *add_ping* function and in the way traces are retrieved. When using DCE applications, the ping traces are retrieved from the standard output of each ping application, as it is done with live *linux::Application* RMs, instead of from the output of the simulation RM.

Listing 4.8 shows a NEPI script where the *add_ping* function, in lines 52 to 68, uses a *linux::ns3::dce::Application* RM instead of the *ns3::V4Ping* RM to model the ping application. The additional attributes exposed by the *linux::ns3::dce::Application* RM, i.e., *sources*, *build*, *binary*, and *arguments*, are used to provide all the information needed to automate compilation, installation, and execution of the ping application binaries for DCE.

4.3.2 NetNS

NetNS is a lightweight emulation library written in Python that uses Linux namespace virtualization [103] to isolate the network stack seen by different processes in the same live host. NEPI provides support for emulation using the NetNS emulator.

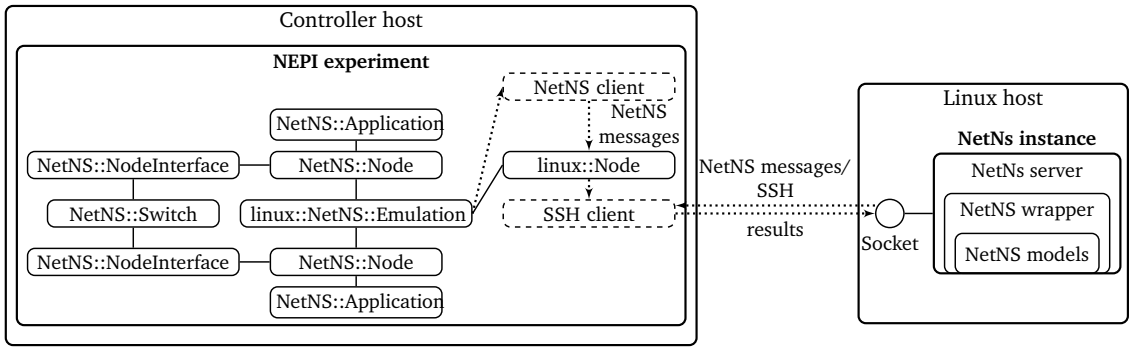


Figure 4.8 – Emulated experiment scenario using NetNS nodes. *NetNS::Node* and *NetNS::Application* RMs are used to model and execute an experiment where two NetNS nodes exchange ICMP messages over an emulated Ethernet switch. The emulation instance runs as a server in a remote Linux host and NEPI sends custom messages over SSH to give instructions to the remote NetNS server.

System processes virtualized with NetNS appear to each other, and to the host, as if they had an independent network stack and network devices. NetNS processes can

be interconnected by virtual devices, bridges, and pipes to construct virtual Ethernet networks where Linux applications can be executed seamlessly. NetNS emulations can be modified interactively, allowing to add nodes and execute new applications at runtime. To support interactive NetNS emulations, the client/server architecture used for ns-3 was recycled for NetNS. Figure 4.8 shows the experiment scenario from Figure 4.1 using NetNS ResourceManagers and a NetNS server.

The NetNS server architecture has only three layers instead of the four layers used by the ns-3 server. NetNS already provides models to describe emulated networks as Python objects, so there is no need for a Python bindings layer. The messages used in the communication between the NetNS client and server was adapted to the characteristics of NetNS, as defined in Table 4.2.

Message	Description
CREATE	Instantiate a NetNS object.
INVOKE	Invoke a method on a NetNS object.
GET	Get the value of a NetNS object attribute.
SET	Set the value of a NetNS object attribute.
FLUSH	Flush standard error and standard output of the NetNS server process.
SHUTDOWN	Stop the NetNS server.

Table 4.2 – Custom protocol messages used by the NetNS client and server.

Listing 4.9 shows a NEPI script using NetNS ResourceManagers to run the ping experiment scenario. Once again, the script structure is fairly similar to the previous live and simulated cases. The *linux::NetNS::Emulation* RM models a NetNS emulation instance running in the localhost, i.e., the Controller host. Two *NetNS::Node* RMs are connected through an emulated Ethernet switch modeled using a *NetNS::Switch* RM. The *NetNS::Application* RMs are used to execute the Linux ping application.

```
1  from nepi.execution.ec import ExperimentController
2
3  def add_linux_host(ec, hostname):
4      host = ec.register_resource("linux::Node")
5      ec.set(host, "hostname", hostname)
6      ec.set(host, "cleanExperiment", True)
7      ec.set(host, "cleanProcesses", True)
8
9      return host
10
11 def add_node(ec, emu):
12     node = ec.register_resource("netns::Node")
13
14     ec.register_connection(node, emu)
15
16     return node
17
18 def add_iface(ec, node, ip, prefixlen):
19     iface = ec.register_resource("netns::NodeInterface")
20
21     ec.register_connection(iface, node)
22
23     ip4 = ec.register_resource("netns::IPv4Address")
24     ec.set(ip4, "ip", ip)
25     ec.set(ip4, "prefix", prefixlen)
```

```
26
27      ec.register_connection(ip4, iface)
28
29      return iface
30
31  def add_channel(ec, ifaces):
32      channel = ec.register_resource("netns::Switch")
33
34      for iface in ifaces:
35          ec.register_connection(channel, iface)
36
37      return channel
38
39  def add_ping(ec, node, peer_ip):
40      ping = ec.register_resource("netns::Application")
41      ec.set(ping, "command", "ping -c3 %s" % peer_ip)
42
43      ec.register_connection(ping, node)
44
45      return ping
46
47  ec = ExperimentController(exp_id="netns_scenario")
48
49  host = add_linux_host(ec, "localhost")
50
51  emu = ec.register_resource("linux::netns::Emulation")
52  ec.register_connection(emu, host)
53
54  node1 = add_node(ec, emu)
55  iface1 = add_iface(ec, node1, "192.168.0.1", "24")
56
57  node2 = add_node(ec, emu)
58  iface2 = add_iface(ec, node2, "192.168.0.2", "24")
59
60  channel = add_channel(ec, [iface1, iface2])
61
62  ping1 = add_ping(ec, node1, "192.168.0.2")
63  ping2 = add_ping(ec, node2, "192.168.0.1")
64
65  ec.deploy()
66
67  ec.wait_finished([ping1, ping2])
68
69  print ec.trace(ping1, "stdout")
70  print ec.trace(ping2, "stdout")
71
72  ec.shutdown()
```

Listing 4.9 – NEPI script to run an emulated experiment that measures peer wise RTT and packet loss between two NetNS Nodes, using the Linux ping application.

The extensions done to support the NetNS platform in NEPI show how the ns-3 server architecture can be recycled to provide interactive control for other software-based network experimentation platforms other than ns-3. Recycling the server architecture for NetNS and implementing the NetNS ResourceManager family took around two weeks. This, compared to the two months that took supporting the ns-3 platform in NEPI, provides another example of how extensibility in NEPI is made easier by a framework design that encourages code re-use.

Implementation of NetNS RMs is still ongoing, so NetNS is not yet fully functional in NEPI. Basic NetNS experiments involving node, application, interface, and switch RMs are nevertheless already supported.

4.4 The Cost of Extending the Framework

This chapter presented examples of different platforms supported in NEPI to show that it is possible to extend the NEPI model to run experiments in simulators, emulators, and testbeds. Nevertheless, adding new ResourceManagers to extend NEPI to new platforms has a development cost. This cost can be estimated by studying how much time was invested in the implementation of existing ResourceManagers.

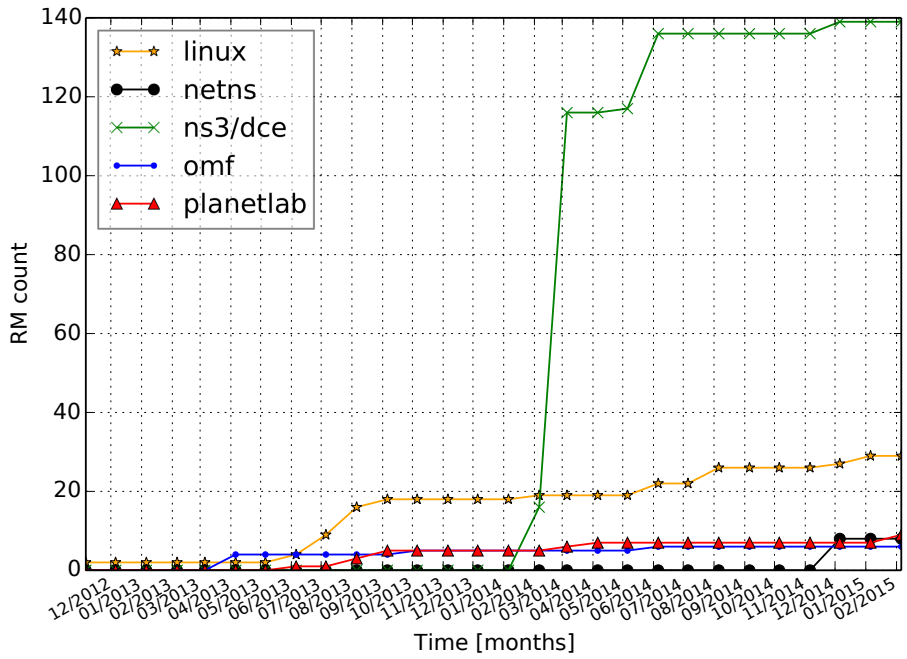


Figure 4.9 – Evolution of the number of ResourceManager classes in time per supported platform.

Figure 4.9 plots data obtained from the commit logs of the NEPI project repository[9] showing the cumulated number of ResourceManagers per platform per month. The development of NEPI 3.0, the NEPI version presented in this thesis, started from scratch at the end of 2012, lasting for 28 months and producing an estimated of 38K Source Lines of Code (SLOC)[10].

The first four months were used to implement the core components of NEPI, including the ExperimentController, the base ResourceManager class, and the experiment orchestration logic. During this time, some early platform ResourceManagers were also implemented for the Linux platform. The Linux platform was the first one to be supported, followed by the OMF and PlanetLab platforms, then the ns-3 platform, and finally the NetNS platform.

[9]Repository available online at `http://nepi.inria.fr/code/nepi/file/43ae08ad10a3`.

[10]Lines of code calculated using the *sloccount* tool, available at `http://www.dwheeler.com/sloccount`.

The Linux platform is the one that received most effort in person months (PM), mostly due to the fact that it was the first one to be supported. For this reason it absorbed most of the cost of developing and testing the general functionality of the framework as well as the specific platform functionality.

The *jump* in the number of ns-3 RM classes is caused by the use of a script to automate their generation. As explained previously, it is possible to dynamically inspect ns-3 models and, using a ResourceManager class template, generate the code for the ns-3 RM classes. All ns-3 ResourceManagers extend a base ns-3 ResourceManager, which in turn extends the base NEPI ResourceManager class.

Adding a new platform in NEPI usually required more effort at the beginning of the development, due to the need to implement the platform communication and the base resource control logic, e.g., SSH client, OMF client. However, once this initial effort was done, adding new ResourceManagers was faster thanks to the possibility of incrementally adding new functionality by extending already existing ResourceManagers. In this sense, the experience developing NEPI confirmed that the choice of a ResourceManager class hierarchy as the basis for model extensibility contributed to facilitate code re-use and to incrementally reduce the time needed to support new resources.

Chapter 5

Framework Evaluation: Efficiency

This chapter presents a benchmark to study how varying different parameters of the orchestration algorithm affects its performance in terms of time, memory, and CPU required to run an experiment.

The benchmark was designed to evaluate the following questions:

- How well the algorithm scales as the number of ResourceManagers used in an experiment increases?
- What is the impact of the reschedule delay in the orchestration algorithm?
- What is the optimal number of threads to parallelize job execution during orchestration?

The benchmark scenario, depicted in Figure 5.1, consists of a varying number of nodes connected to a same network. Each node is associated to a varying number of applications that send ICMP traffic to a randomly chosen node in the network. Each node is connected to the network through a network interface.

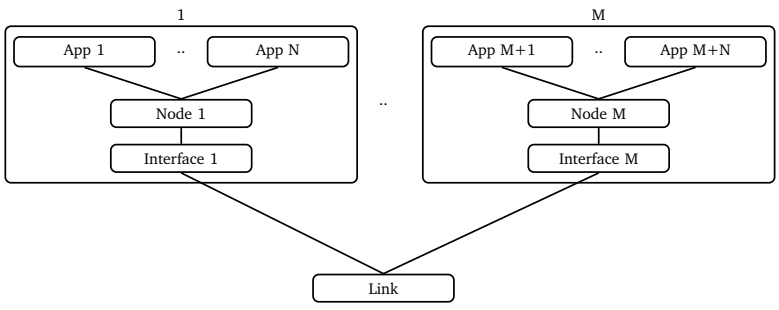


Figure 5.1 – Benchmark scenario composed of a varying number of nodes connected to a single link through a network interface. Each node executes the same number of applications.

The benchmark scenario is first evaluated using ResourceManagers from the Dummy platform defined in Listing 3.1 on Chapter 3. The Dummy ResourceManagers do not

perform any real operations on a platform. Instead, when invoked, the deploy and start methods of the Dummy RMs wait for a configurable amount of time, i.e., the *wait time*. The purpose of using a Dummy platform instead of a real platform is to have better control over the duration of the RM methods invoked during orchestration, avoiding the varying delays experienced by ResourceManagers interacting with resources on real platforms.

The same benchmark scenario is then re-evaluated using the ns-3 and PlanetLab platforms in order to understand how the orchestration algorithm is affected by the behavior of ResourceManagers interacting with real platforms. The choice of the ns-3 and PlanetLab platforms for the benchmark is motivated by their opposite controllability and realism features. ns-3 is a perfectly controllable platform whereas PlanetLab provides a realistic but completely uncontrolled Internet environment.

For each platform, a NEPI script describing the benchmark scenario is evaluated multiple times using different number of nodes, applications, and interface RMs, and varying the number of worker threads used by the orchestration engine and the reschedule delay Δt used to reschedule methods[1].

The three benchmarks were run in a 64-bit Linux machine with Fedora 17, 4 cores, and 8GB of memory, using Python 2.7. Measurements were made of the total duration of each run, as well as of the memory and CPU usage, these last using the *psutil*[2] library, version 1.0.1.

The scripts to reproduce the benchmarks, as well as the collected data and the scripts to generate the plots presented in this chapter, can be found online at:

`http://nepi.inria.fr/code/nepi_benchmark/file/tip/efficiency`

5.1 Dummy Platform Benchmark

Listing 5.1 shows the NEPI script used to run the Dummy platform benchmark, using the Dummy ResourceManagers defined in Listing 3.1 on Chapter 3. In this script, Dummy applications are connected to Dummy nodes, and Dummy nodes are connected to one another through Dummy interfaces and a Dummy link. No traffic is exchanged between nodes in the experiment since Dummy RMs model fake resources.

```
1  from nepi.execution.ec import ExperimentController
2
3  nodes = list()
4  apps = list()
5  ifaces = list()
6
7  ec = ExperimentController("dummy_benchmark")
8
9  for i in xrange(node_count):
10     node = ec.register_resource("dummy::Node")
11     nodes.append(node)
12
13     iface = ec.register_resource("dummy::Interface")
```

[1] The reschedule delay Δt is the time used to postpone a job by the orchestration algorithm defined in Section 2.3.3.2 on Chapter 2.

[2] The *psutil* Python library is available at `https://pypi.python.org/pypi/psutil`.

```
14      ec.register_connection(node, iface)
15      ifaces.append(iface)
16
17      for i in xrange(app_count):
18          app = ec.register_resource("dummy::Application")
19          ec.register_connection(node, app)
20          apps.append(app)
21
22  link = ec.register_resource("dummy::Link")
23
24  for iface in ifaces:
25      ec.register_connection(link, iface)
```

Listing 5.1 – NEPI script used in the Dummy platform benchmark.

The number of worker threads used by the orchestration engine can be configured using the *NEPI_NTHREADS* environment variable as shown in Listing 5.2. This variable must be configured before the instantiation of the ExperimentController, else the default value of 50 threads is used. The reschedule delay is configured by setting the *reschedule_delay* attribute of the ResourceManager class, as shown in line 5.

When the orchestration engine invokes the deploy method on a Dummy RM or the start method on a Dummy application, the worker thread that executes the method sleeps for an amount of time defined by the global variable *wait_time*. The *wait_time* global variable is configured as shown in line 9, in the example in Listing 5.2, to control the duration of the deploy and start methods.

```
1  # set number of threads used by the orchestration engine to execute jobs
2  os.environ["NEPI_NTHREADS"] = str(thread_count)
3
4  # set reschedule delay
5  ResourceManager._reschedule_delay = "0s"
6
7  # set the time to wait in the do_deploy and do_start methods
8  global wait_time
9  wait_time = "1s"
```

Listing 5.2 – Code to set the number of threads, the rechedule delay, and the time to wait for the Dummy benchmark.

To conduct the Dummy benchmark, the script in Listing 5.1 was evaluated for all combinations of 1, 10, 100, and 1000 nodes with 1, 10, and 50 applications, using 1, 10, 50, and 100 threads, and for reschedule delays of 0 and 0.5 seconds and wait time of 0 and 1 seconds. This produces a total of 192 cases. Each case was reproduced 15 times, producing a total of 2880 runs.

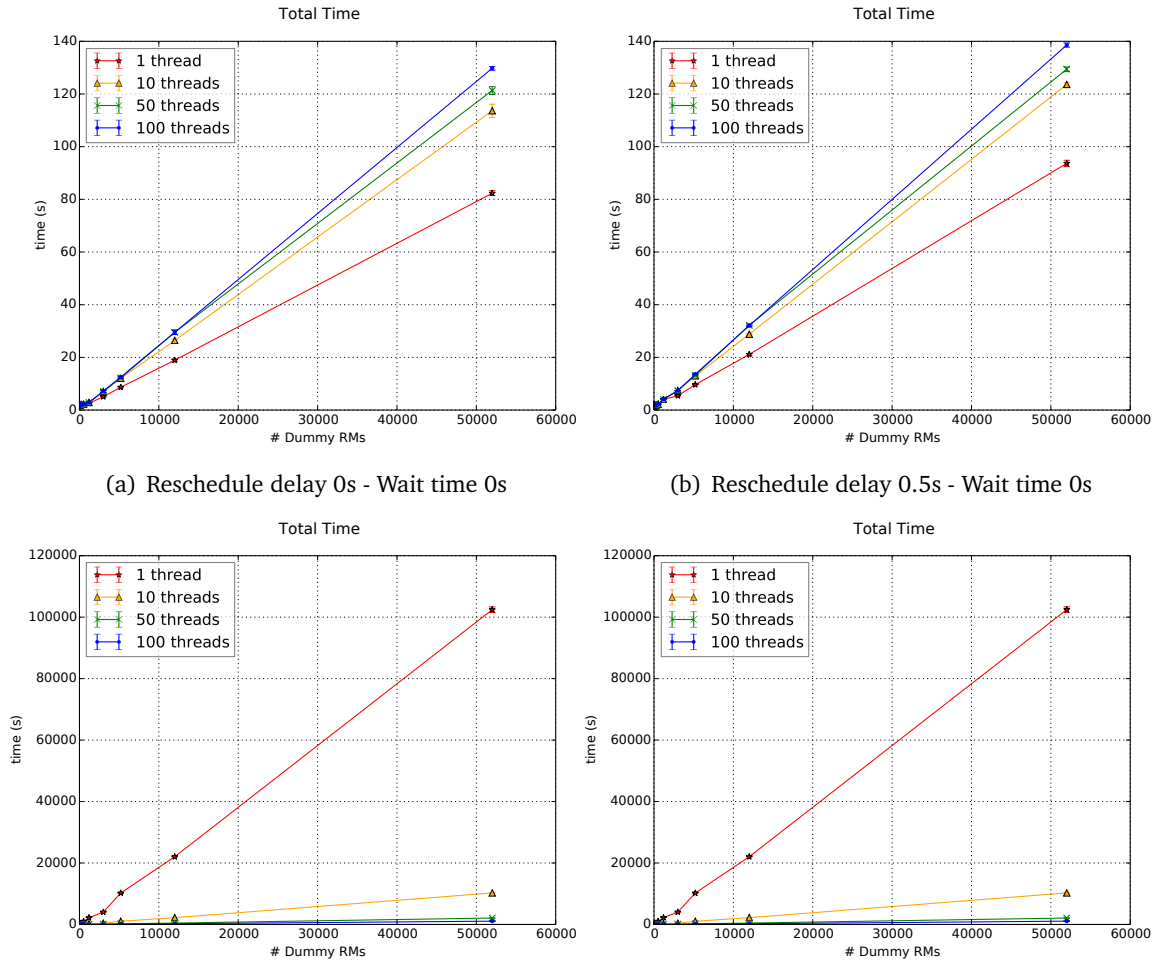
5.1.1 Time Performance

The plots in Figure 5.2 show the mean total time required to run the benchmark script for different numbers of ResourceManagers and threads. Each plot corresponds to one of the four combinations of reschedule delay and wait time. Each data point in the plots shows the mean run time for a combination of ResourceManagers count, thread count, reschedule delay, and wait time. The error bars show the standard deviation from the mean run time for each data point, over a sample of 15 runs. The number of ResourceManagers is calculated as:

$$node\_count * 2 + app\_count * node\_count + 1$$

Nodes are counted twice since each node is associated to one interface, and the link accounts for an additional ResourceManager.

Time performance for the Dummy platform benchmark



(a) Reschedule delay 0s - Wait time 0s

(b) Reschedule delay 0.5s - Wait time 0s

(c) Reschedule delay 0s - Wait time 1s

(d) Reschedule delay 0.5s - Wait time 1s

Figure 5.2 – Mean total time to run the Dummy platform benchmark script for combinations of ResourceManager number, threads number, reschedule delay, and wait time. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs. In most cases the error bars are too small to be noticeable.

An important observation that can be made is that the Dummy benchmark script runs in linear time. As the number of ResourceManagers increases, the time needed to complete a run corresponds to a value given by a linear function. For all plotted data,

the correlation coefficient of fitting the measured data to a linear model is 0.999, which indicates an almost perfect fit[3].

Another observation is that there is only a small reduction in the experiment run time when setting the reschedule delay to 0 instead of 0.5 seconds. This seems to indicate that, at least for the Dummy platform and for reschedule delays of 0 and 0.5 seconds, the reschedule delay has no significant impact on the performance of the orchestration algorithm.

Regarding the wait time, for a wait time of 0 seconds using more threads results in a worse performance, whereas for a wait time of 1 second the opposite occurs. This could be expected due to the way Python 2.7 handles access to the interpreter. Python synchronises access to the interpreter using a Global Interpreter Lock (GIL)[4] that can be held by only one thread at the time. Switching between threads introduces an overhead and it can have a negative impact on the performance of CPU intensive multi-threaded programs. When the wait time is 0 seconds, RM methods spend almost no time executing and most of the orchestration time is spent assigning new jobs to worker threads. In this case, frequent thread switching adds an overhead to the experiment duration. On the contrary, when the wait time is 1 second most of the experiment duration is spent in sleep operations, and the thread switching overhead is mitigated by parallelizing the execution of RM methods. Python switches threads during wait operations, such as sleeping or waiting on a socket, so RM methods that are CPU intensive will not take as much advantage of the use of multiple threads.

The time needed to run the Dummy script with a wait time of 0 seconds for the worst case, for 50000 ResourceManagers and using 100 threads, is less than 2.5 minutes. This time is in fact the orchestration overhead since only the NEPI Scheduler is doing any processing. When the operation delay is 1 second, and most of the run time is spent by the RMs sleeping, the best performance for 50000 RMs occurs with 100 threads, with an experiment duration of around 18 minutes.

The estimated total time spent on deploy and start operations, for a wait time of 1 second, can be computed as:

$$deploy\_time = node\_count * 2 + app\_count * node\_count + 1$$
$$start\_time = app\_count * node\_count$$

For 1000 nodes and 50 applications the estimated experiment run time is 102001 seconds, around 28 hours. This correctly approximates the mean experiment run time observed in the benchmark for the worse case scenario, using 52001 ResourceManagers with a wait time of 1 second and using 1 thread. The benchmark produces a mean run time of 102448 seconds for this case, around 0.45% more than the estimated experiment run time. Even for the worse case, the overhead introduced by the orchestration algorithm is relatively small.

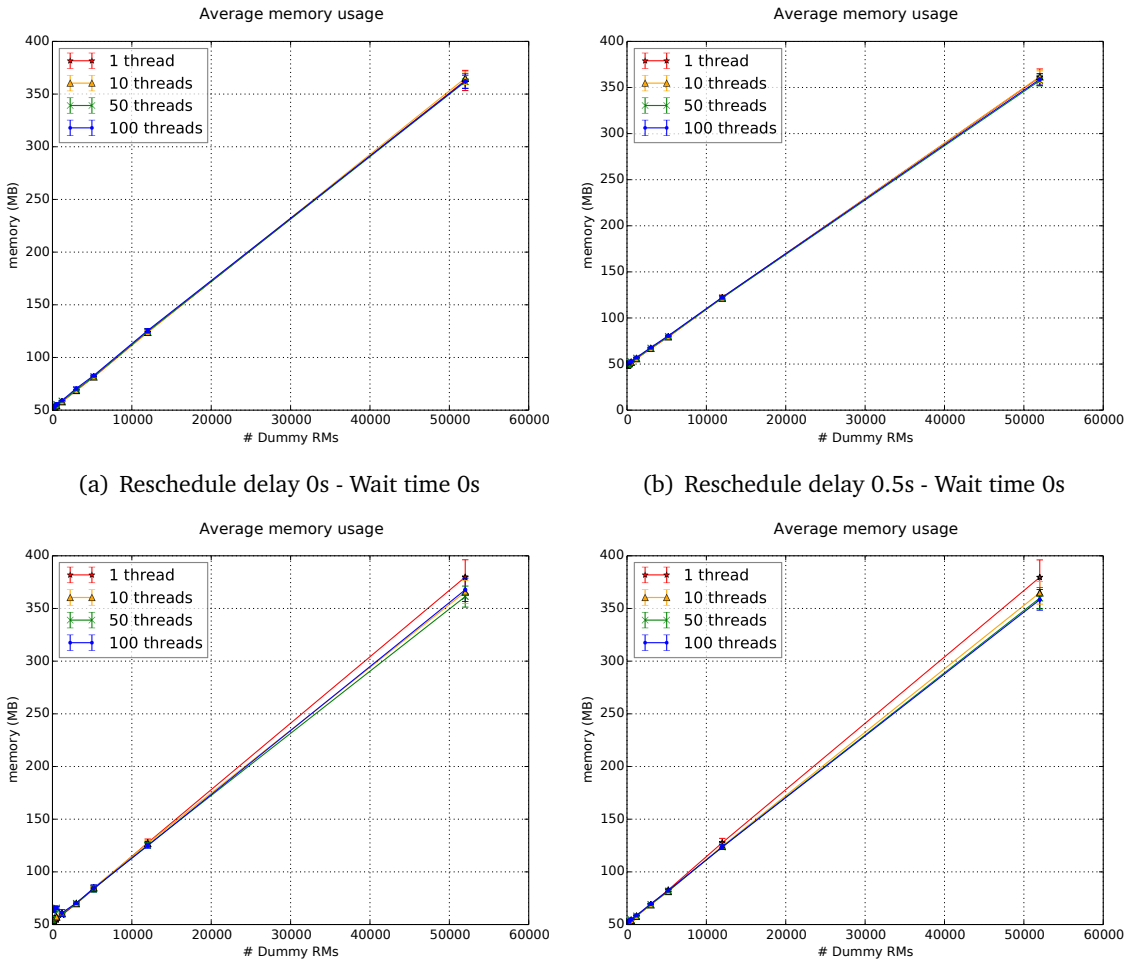
[3]The correlation coefficients were computed using the *stats.linregress* function from the *scipy* Python library.

[4]GIL internals presented by Dave Beazley at PyCon'2010 available at `https://www.youtube.com/watch?v=Obt-vMVdM8s`.

Finally, the plots show that the observed run times do not significantly deviate from the mean values. This suggests that the orchestration algorithm does not introduce variability in the experiment duration, which is an important property to support reproducible experimentation. Since jobs are initially scheduled in a deterministic order for a same NEPI script, if a platform supports repeatability, using a single thread should guarantee that jobs are executed in the same order for every run. When multiple threads are used in the orchestration, the job execution order could vary from one run to another.

5.1.2 Memory Performance

Memory performance for the Dummy platform benchmark



(a) Reschedule delay 0s - Wait time 0s

(b) Reschedule delay 0.5s - Wait time 0s

(c) Reschedule delay 0s - Wait time 1s

(d) Reschedule delay 0.5s - Wait time 1s

Figure 5.3 – Mean average memory used during the run of the Dummy platform benchmark script for different combinations of ResourceManager number, threads number, reschedule delay, and wait time. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

The plots in Figure 5.3 show the mean average memory used by the benchmark script for different number of ResourceManagers and threads. Each plot corresponds to one of the four combinations of reschedule delay and wait time.

Memory usage was measured with the *psutil* library at a 0.5 second interval sampling frequency. The average of these measurements for each run was used as the sample values to compute the mean memory usage for each data point. The error bars show the standard deviation from the mean memory usage for each data point, over a sample of 15 runs.

The plots show that the mean memory used by the Dummy benchmark script is linear with respect to the number of ResourceManagers. The correlation coefficients obtained from fitting the data of each curve to a linear model is 0.999 in all cases, which corresponds to an almost perfect fit.

There is no significant difference between the results obtained for reschedule delays of 0 or 0.5 seconds, suggesting that the reschedule delay does not significantly impact the memory consumed by the NEPI script. For a large number of RMs, using a wait time of 1 second results in a slightly increased memory consumption when fewer threads are used. This might be caused by artifacts introduced by sampling the memory at a low frequency. Since some threads might be switching or inactive at a given instant, the total memory used might appear slightly lower.

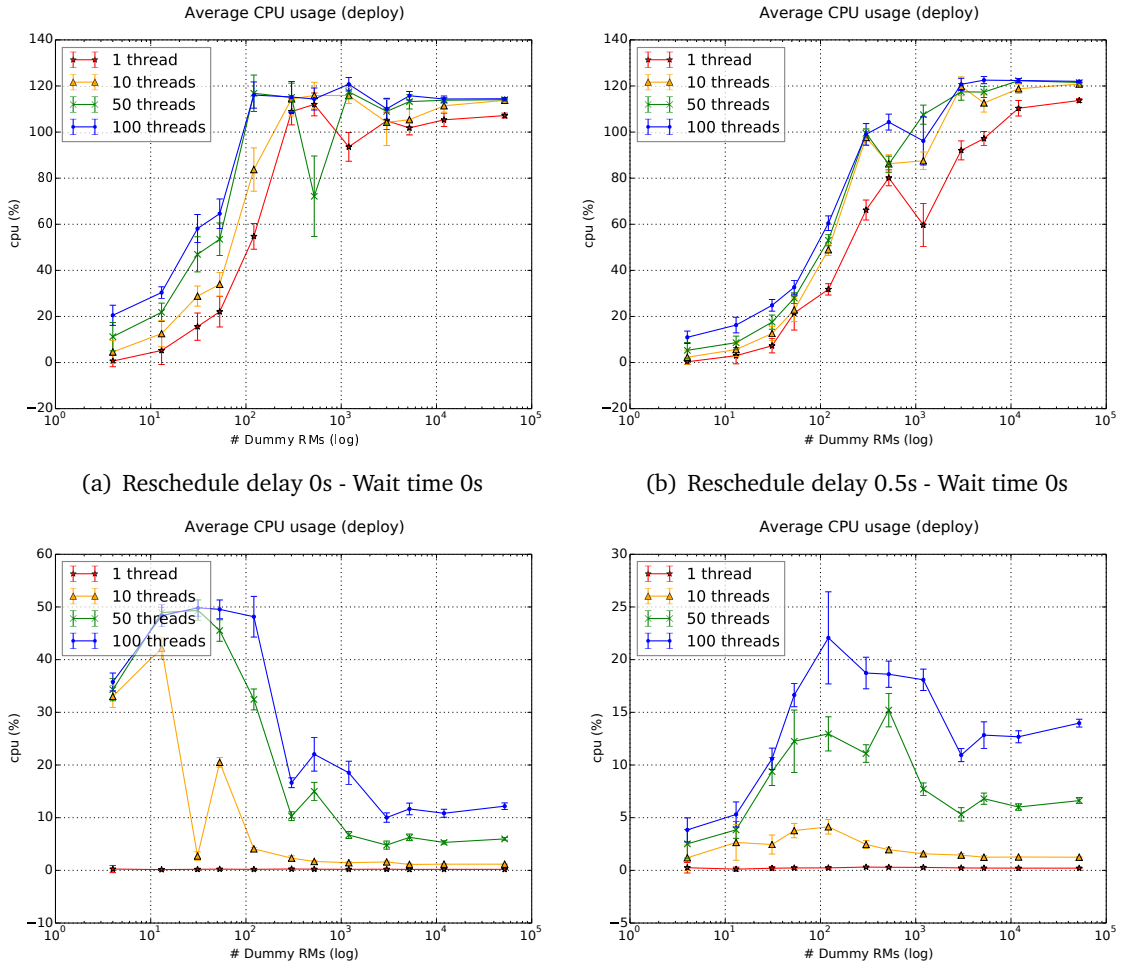
The amount of memory used to run the Dummy script with a large number of RMs is considerable. For close to 50000 RMs more than 350MB of memory are used. Python has a considerably higher memory consumption than other non-interpreted programming languages, for instance C. Almost everything in Python, even basic types, is an object, so Python scripts require more memory than C programs. Also, Python is interpreted instead of compiled and object types are determined dynamically at run time. This requires additional data to be included in all objects to support the dynamic type resolution. A solution to reduce the memory required by a NEPI script would be to re-implement the core parts of NEPI, including the ResourceManager base class, in C.

5.1.3 CPU Performance

Figures 5.4, 5.5, and 5.6 show the mean average CPU used by the Dummy benchmark script during the deploy, execution, and release steps of the experiment run, respectively. The deploy step is assumed to begin when the *deploy* method of the ExperimentController is invoked and to end when all RMs are in state READY. The execution step begins when all RMs are in state READY and end with the invocation of the *shutdown* method of the ExperimentController. Finally, the release step begins with the invocation of the *shutdown* method and ends when all RMs reach the state RELEASED.

CPU usage was measured with the *psutil* library at a 0.5 second interval sampling frequency. The average of these measurements for each run was used as the sample values to compute the mean CPU usage for each data point. The error bars show the standard deviation from the mean CPU usage for each data point, over a sample of 15 runs.

CPU performance for the Dummy platform benchmark during deployment



(a) Reschedule delay 0s - Wait time 0s

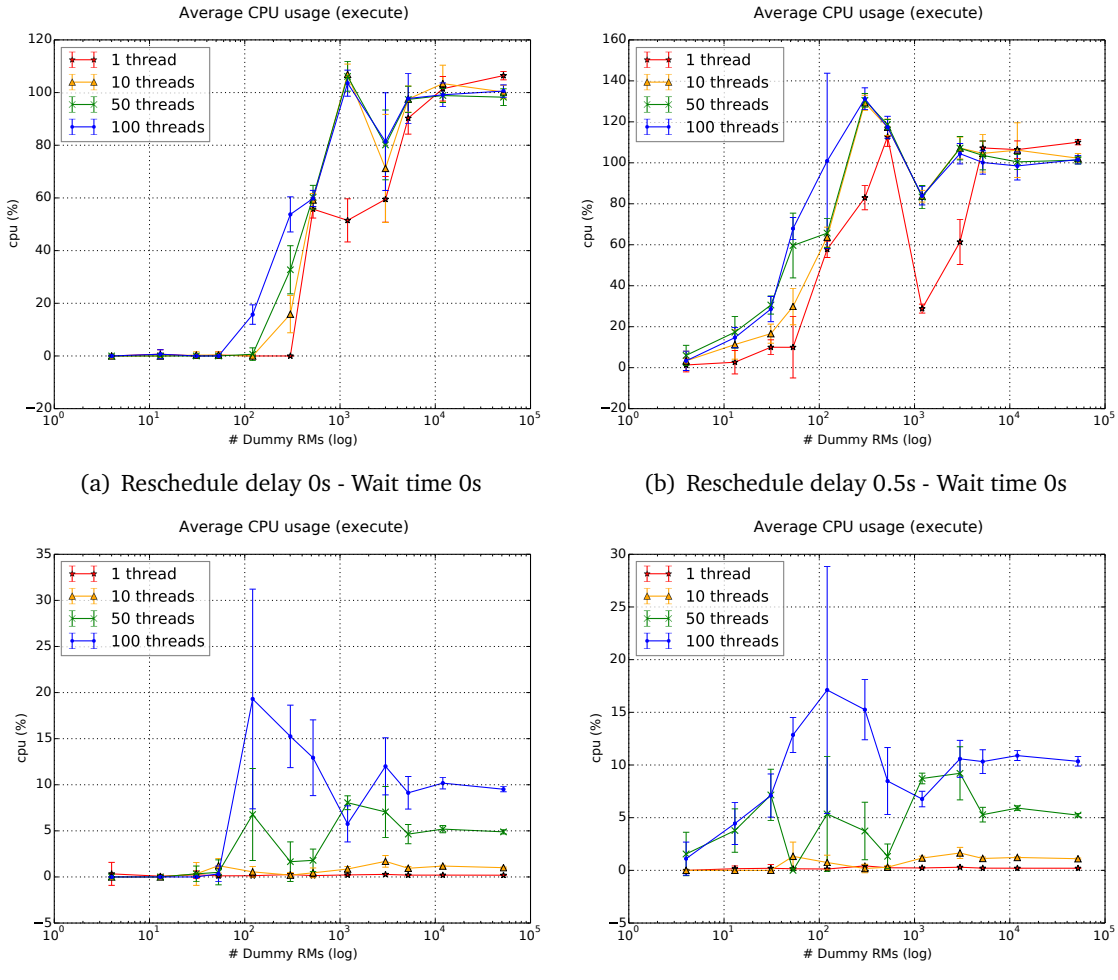
(b) Reschedule delay 0.5s - Wait time 0s

(c) Reschedule delay 0s - Wait time 1s

(d) Reschedule delay 0.5s - Wait time 1s

Figure 5.4 – Mean average CPU used during the deploy step of the Dummy script for different combinations of ResourceManager number, threads number, reschedule delay, and wait time. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

CPU performance for the Dummy platform benchmark during execution



(a) Reschedule delay 0s - Wait time 0s

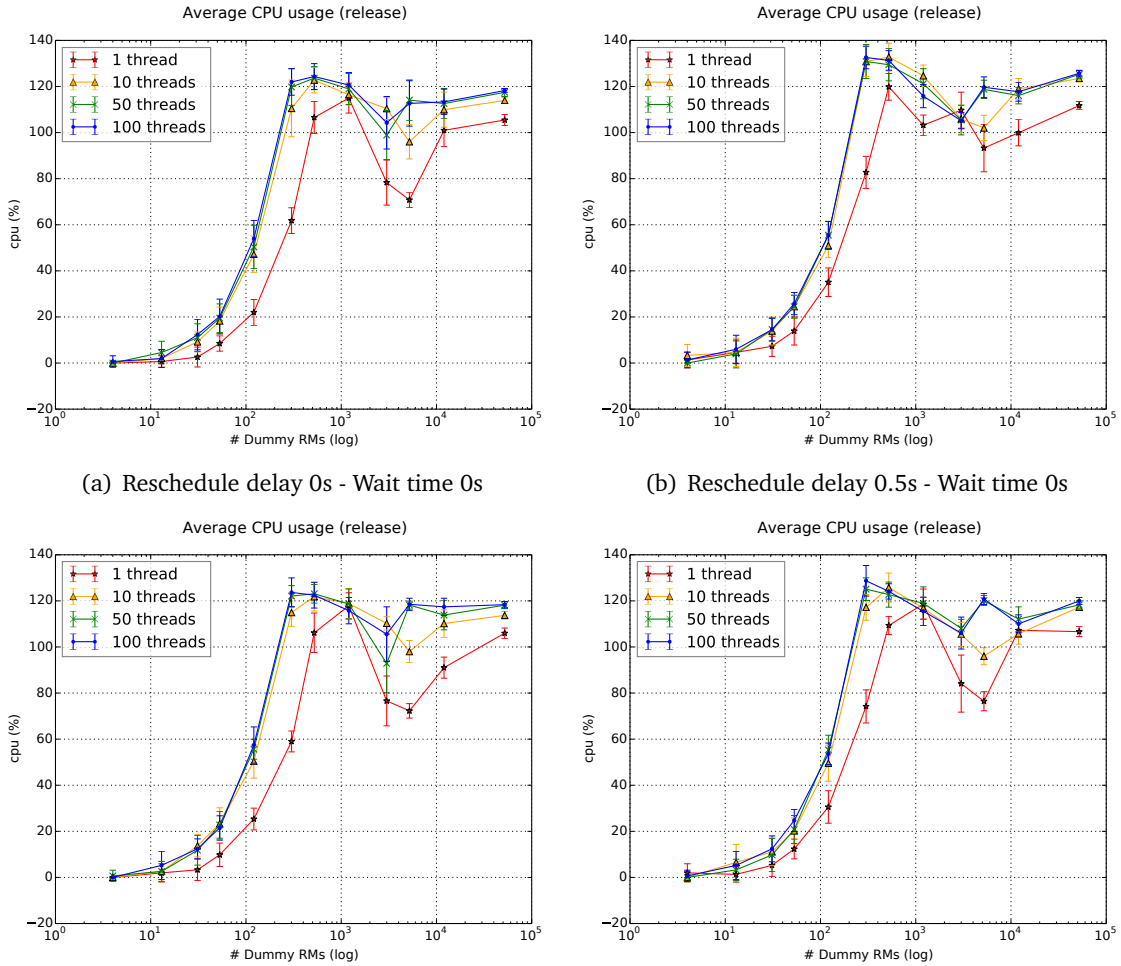
(b) Reschedule delay 0.5s - Wait time 0s

(c) Reschedule delay 0s - Wait time 1s

(d) Reschedule delay 0.5s - Wait time 1s

Figure 5.5 – Mean average CPU used during the execution step of the Dummy script for different combinations of ResourceManager number, threads number, reschedule delay, and wait time. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

CPU performance for the Dummy platform benchmark during release



(a) Reschedule delay 0s - Wait time 0s

(b) Reschedule delay 0.5s - Wait time 0s

(c) Reschedule delay 0s - Wait time 1s

(d) Reschedule delay 0.5s - Wait time 1s

Figure 5.6 – Mean average CPU used during the release step of the Dummy script for different combinations of ResourceManager number, threads number, reschedule delay, and wait time. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

In contrast with the time and memory usage, which did not vary across runs, CPU usage is highly variable. However, distinct CPU usage patterns can be recognised for the deploy, execute, and release steps. Deploy and release tend to be more CPU intensive, in particular for a wait time of 0 seconds. CPU consumption shows a peak around 100 ResourceManagers and goes down again around 1000 ResourceManagers. The experiment seems to hit its maximum CPU utilisation at 100 RMs, reaching a job processing saturation at that point.

Increasing the number of threads increases CPU load, and in all cases a wait time of 0 seconds produces a CPU consumption of over 100% for more than 100 ResourceManagers. A wait time of 1 second produces a maximum CPU consumption below 50% for

the deploy and execution steps. The reschedule delay does not show a clear impact on the CPU utilization.

5.2 ns-3 Platform Benchmark

In order to compare the results obtained using the *ideal* Dummy platform with results from real platforms, the benchmark was evaluated using ns-3. ns-3 simulations are highly controllable and repeatable. They can be executed as self-contained experiments that do not depend on external events.

Listing 5.3 shows the script used to evaluate the benchmark with ns-3. In the script, an ns-3 simulation is configured to run in the *localhost*, and a variable number of ns-3 nodes are connected to the simulated Csma network. The same number of ns-3 *V4Ping* applications are added to each node. Each *V4Ping* application sends ICMP traffic to a randomly chosen node in the simulation.

The script was evaluated for all combinations of 1, 10, 100, and 1000 nodes with 1, 10, and 50 applications, using 1, 10, 50, and 100 threads, and reschedule delays of 0 and 0.5 seconds. This produces a total of 96 cases. Each case was reproduced 15 times, producing a total of 1440 runs. The wait time, i.e., the time spent to execute an operation on a platform resource, can not be fixed for RMs of real platforms.

```
import ipaddr

from nepi.execution.ec import ExperimentController

ec = ExperimentController("ns3_benchmark")

host = ec.register_resource("linux::Node")
ec.set(host, "hostname", "localhost")

simu = ec.register_resource("linux::ns3::Simulation")
ec.register_connection(simu, host)

nodes = list()
apps = list()
ifaces = list()
ips = dict()

segment = "10.0.0.0/16"
net = ipaddr.IPv4Network(segment)
ip_itr = net.iterhosts()

for i in xrange(node_count):
    node = ec.register_resource("ns3::Node")
    ec.set(node, "enableStack", True)
    ec.register_connection(node, simu)

    nodes.append(node)

    ip = ip_itr.next().exploded

    iface = ec.register_resource("ns3::CsmaNetDevice")
    ec.set(iface, "ip", ip)
    ec.set(iface, "prefix", prefix)
    ec.register_connection(node, iface)

    queue = ec.register_resource("ns3::DropTailQueue")
    ec.register_connection(iface, queue)

    ifaces.append(iface)

```

```
41      ips[node] = ip
42
43  for nid in nodes:
44      # choose a random node to ping or ping itself if there
45      # is only one node available
46      for j in xrange(app_count):
47          remote_ip = ips[nid]
48
49          if len(nodes) > 1:
50              choices = ips.values()
51              choices.remove(remote_ip)
52              remote_ip = random.choice(choices)
53
54          app = ec.register_resource("ns3::V4Ping")
55          ec.set (app, "Remote", remote_ip)
56          ec.set (app, "Verbose", True)
57          ec.set (app, "Interval", "1s")
58          ec.set (app, "StartTime", "0s")
59          ec.set (app, "StopTime", "20s")
60          ec.register_connection(app, node)
61          apps.append(app)
62
63  channel = ec.register_resource("ns3::CsmaChannel")
64  ec.set(channel, "Delay", "0s")
65
66  for iface in ifaces:
67      ec.register_connection(channel, iface)
```

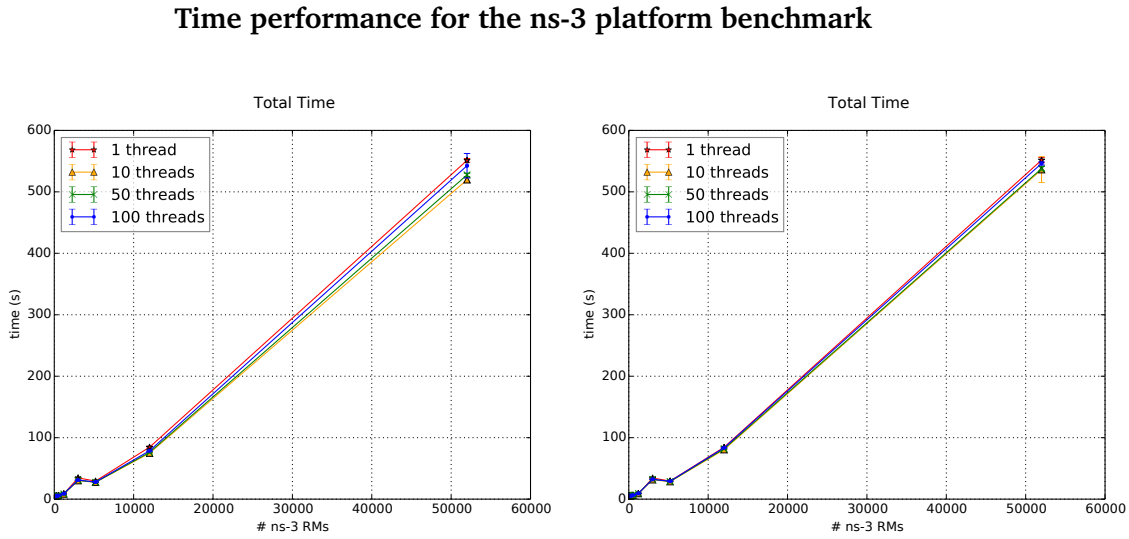
Listing 5.3 – NEPI script used to conduct the ns-3 platform benchmark.

5.2.1 Time Performance

The plots in Figure 5.7 show the mean total time required to run the benchmark script for different numbers of ResourceManagers and threads, using reschedule delays of 0 and 0.5 seconds. Each data point in the plots shows the mean run time for a combination of ResourceManagers count, thread count, and reschedule delay. The error bars show the standard deviation from the mean run time for each data point, over a sample of 15 runs. The number of ResourceManagers is calculated in the same way as for the Dummy benchmark.

Like the Dummy script, the ns-3 script runs in linear time. The data obtained for the run duration as a function of the number of ResourceManagers matches a linear model with correlation coefficient of 0.995.

The mean experiment duration for 50000 RMs for all thread counts is around 9 minutes. This is the same order of magnitude than the maximum mean duration obtained for the Dummy benchmark when using a wait time of 0 seconds. The number of threads used has little impact on the mean experiment duration. This can be explained taking into account the way the ns-3 ResourceManagers are implemented. Since the ns-3 server that controls the ns-3 simulation instance is not thread safe, all communication between the ns-3 RMs and the server is synchronized using locks. This has the effect of serializing the jobs executed by the orchestration engine, resulting in a similar performance regardless of the number of threads used. A small divergence in performance, depending on the number of threads, appears for a large number of RMs when using a reschedule delay of 0 seconds.



(a) Reschedule delay 0s

(b) Reschedule delay 0.5s

Figure 5.7 – Mean total time to run the ns-3 platform benchmark script for combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

5.2.2 Memory Performance

Figure 5.8 shows the mean average memory used by the benchmark script for different number of ResourceManagers and threads, and values of reschedule delays of 0 and 0.5 seconds.

Memory usage was measured with the *psutil* library at a 0.5 second interval sampling frequency. The average of these measurements for each run was used as the sample values to compute the mean memory usage for each data point. The error bars show the standard deviation from the mean memory usage for each data point, over a sample of 15 runs.

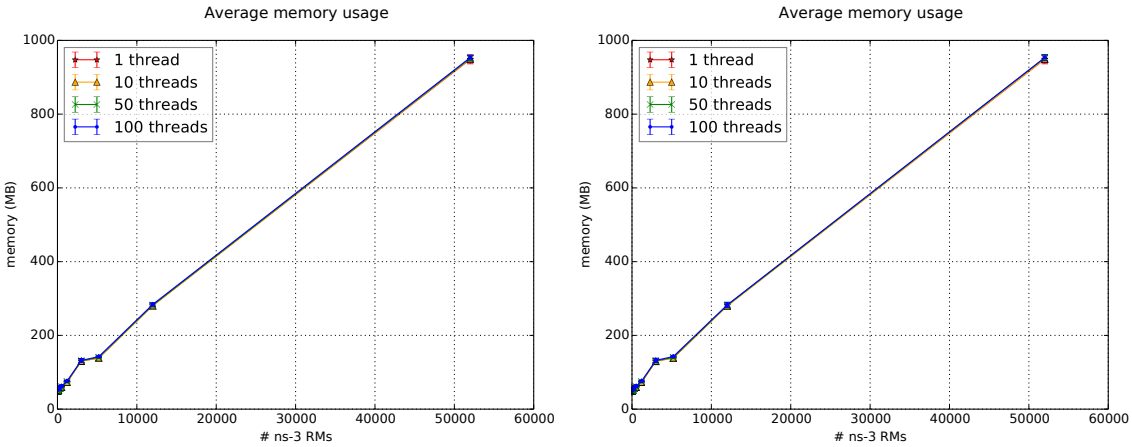
The results show that the mean memory used by the ns-3 script is linear with respect to the number of ResourceManagers. The correlation coefficients obtained from fitting the data of each curve to a linear model is 0.999 in all cases, which corresponds to an almost perfect fit.

Like in the previous cases, there is no significant difference between a reschedule delay of 0 or 0.5 seconds, suggesting that the reschedule delay does not impact the memory consumed by the NEPI script.

The amount of memory required to run the ns-3 benchmark script does not present variations across runs for a same number of ResourceManagers and threads. According to the results obtained, the number of threads does not show either an impact the memory used by the script. Nevertheless, the memory consumption of the ns-3 benchmark script is considerably larger than for the Dummy script, i.e., close to 950MB for ns-3 against 350MB for the Dummy platform, for around 50000 ResourceManagers.

As mentioned previously, Python is expensive in terms of memory consumption com-

Memory performance for the ns-3 platform benchmark



(a) Reschedule delay 0s

(b) Reschedule delay 0.5s

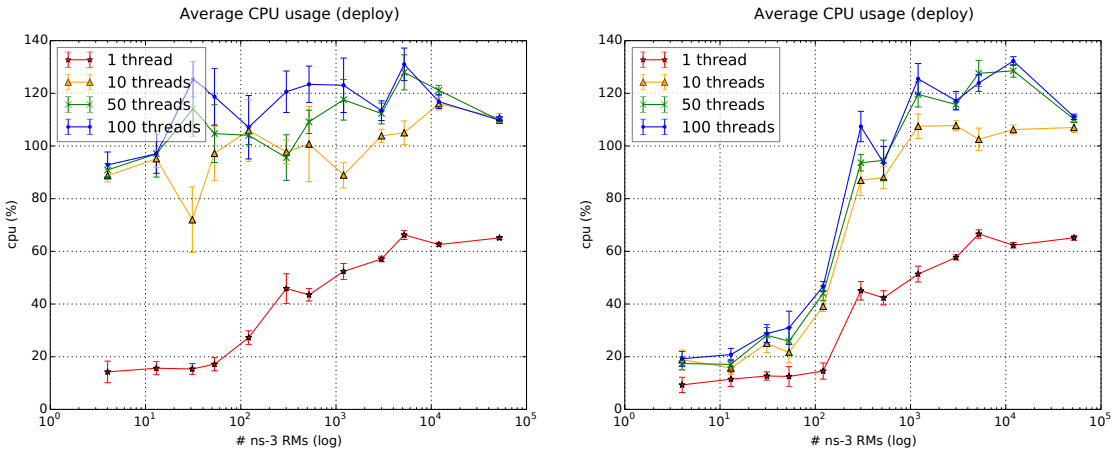
Figure 5.8 – Mean average memory used during the run of the ns-3 benchmark script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

pared to other programming languages such as C/C++. A Dummy ResourceManager, having almost no attributes, takes up less memory than an ns-3 ResourceManager. Comparing the results from the ns-3 and the Dummy benchmarks, the ns-3 script requires over two times more memory than the Dummy script for 50000 ResourceManagers.

5.2.3 CPU Performance

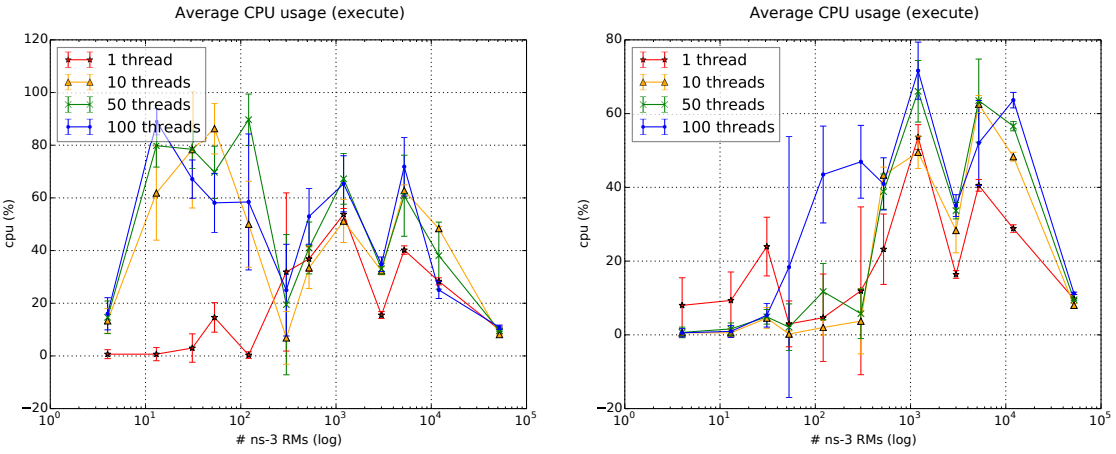
Figures 5.9, 5.10, and 5.11 show the mean average CPU used by the ns-3 benchmark script during the deploy, execution, and release steps of the script run, respectively.

As before, the CPU usage was measured with the *psutil* library at a 0.5 seconds interval sampling frequency. The average of these measurements for each run was used as the sample values to compute the mean CPU usage for each data point. The error bars show the standard deviation from the mean CPU usage for each data point, over a sample of 15 runs.



(a) Reschedule delay 0s (b) Reschedule delay 0.5s

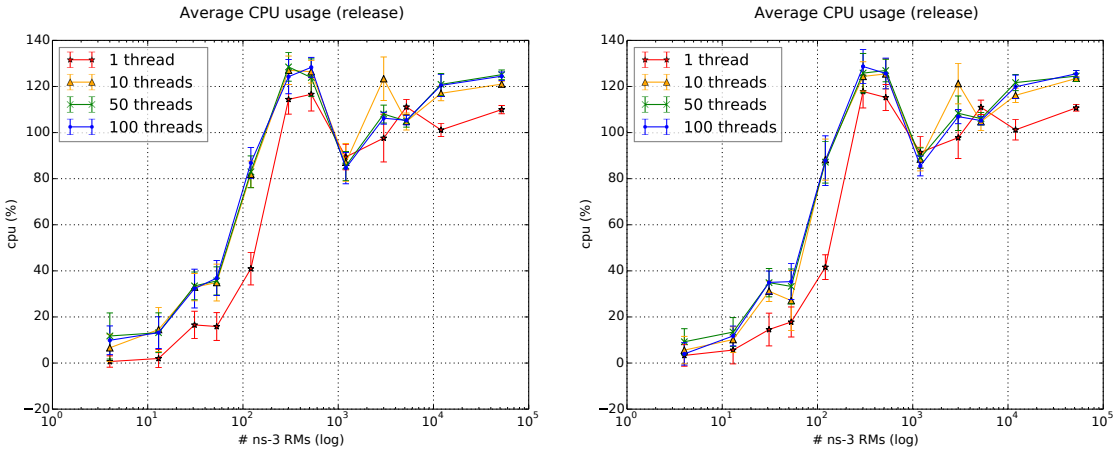
Figure 5.9 – Mean average CPU used during the deploy step of the ns-3 script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.



(a) Reschedule delay 0s (b) Reschedule delay 0.5s

Figure 5.10 – Mean average CPU used during the execution step of the ns-3 script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

CPU performance for the ns-3 platform benchmark during release



(a) Reschedule delay 0s

(b) Reschedule delay 0.5s

Figure 5.11 – Mean average CPU used during the release step of the ns-3 script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

As expected, again the CPU usage shows considerable variability across runs, but also different run steps present distinct usage patterns. Using 1 thread is consistently more efficient, particularly during deploy, than using 10, 50, or 100 threads. Taking into account that using more than one thread does not significantly shorten the duration of experiments, or decrease memory consumption, this benchmark shows that for ns-3 NEPI scripts it is better to use a single thread.

5.3 PlanetLab Platform Benchmark

To complement the results obtained with the Dummy and the ns-3 benchmarks, a similar benchmark scenario was evaluated using the PlanetLab platform. A live platform, such as PlanetLab, permits to evaluate how variations in the job durations, due to changing conditions in the environment, affect the performance of the orchestration algorithm. NEPI controls resources in PlanetLab testbed by issuing SSH commands over the Internet. Communication delays over the Internet can vary randomly depending on uncontrolled conditions. Variations in the time needed to execute SSH commands on a remote host will directly affect the duration of the jobs executed by the NEPI scheduler.

Listing 5.4 shows the script used to evaluate the benchmark using PlanetLab ResourceManagers. In this script *planetlab::Node* RMs are connected to *linux::Application* RMs. A list of preselected PlanetLab hosts is read from the *planetlab_hosts.txt* file. The hosts are preselected so that the RTT from the Controller host running the NEPI script to each PlanetLab host is under 4 seconds. The application RMs execute the Linux ping command to send ICMP traffic to a randomly selected host from the list of hosts. In the PlanetLab script it is not necessary to configure the network interfaces or the link. Plan-

etLab hosts are directly connected through the Internet and users do not have privileges to configure the network interfaces.

The PlanetLab script was evaluated for all combinations of 1, 10, and 50 nodes with 1, 10, and 50 applications, using 1, 10, 50, and 100 threads, and reschedule delays of 0 and 0.5 seconds. This produces a total of 72 cases. Each case was reproduced 15 times, producing a total of 1080 runs. While there are technically several hundreds of hosts available in PlanetLab, a lot of them are unresponsive or faulty for long periods of time. This makes it difficult to find a large number of hosts that are responsive uninterruptedly for long periods. For these reasons the number of nodes used in the PlanetLab benchmark was limited to 50.

```
hostnames = []
with open("planetlab_hosts.txt", "r") as f:
    for line in f:
        line = line.strip()
        if not line:
            continue

        hostnames.append(line)

ec = ExperimentController("planetlab_bench")

nodes = []
apps = []

for i in xrange(node_count):
    hostname = hostnames[i]
    node = ec.register_resource("planetlab::Node")
    ec.set(node, "hostname", hostname)
    ec.set(node, "username","my_user")
    ec.set(node, "identity", "~/.ssh/id_rsa")

    nodes.append(node)

i = 0
for nid in nodes:
    # choose a random node to ping or ping itself if there
    # is only one node available
    for j in xrange(app_count):
        hostname = hostnames[i]

        if len(nodes) > 1:
            choices = hostnames[:]
            choices.remove(hostname)
            hostname = random.choice(choices)

        app = ec.register_resource("linux::Application")
        ec.set(app, "command", "ping -c3 %s" % hostname)
        ec.register_connection(app, node)
        apps.append(app)

    i+=1
```

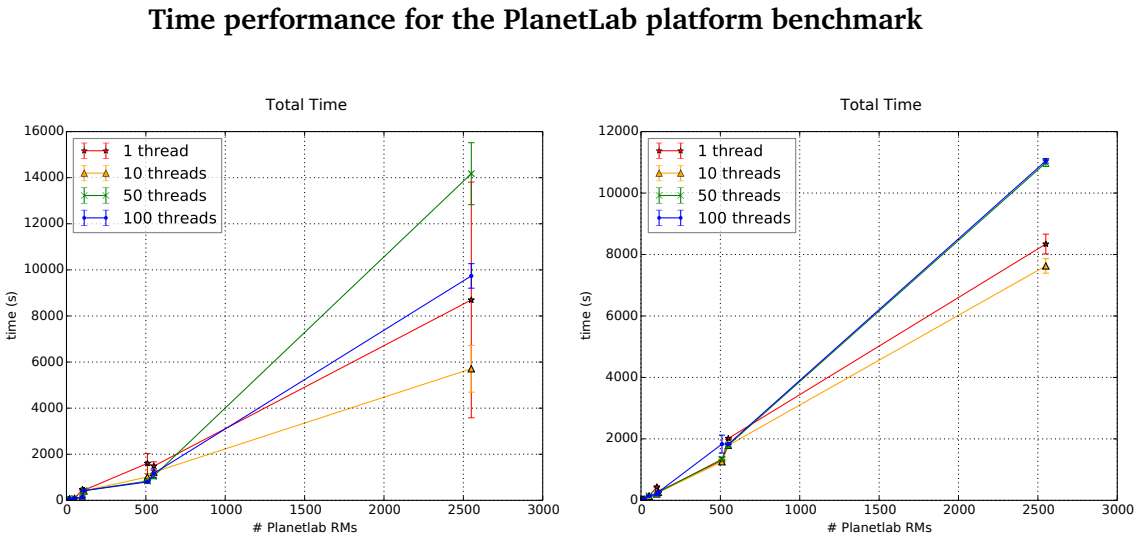
Listing 5.4 – NEPI script used in the PlanetLab platform benchmark.

5.3.1 Time Performance

The plots in Figure 5.12 show the mean total time required to run the benchmark script for different number of ResourceManagers and threads, and for reschedule delays of 0 and 0.5 seconds. Each data point in the plots shows the mean run time for a combination of ResourceManagers count, thread count, and reschedule delay. The error bars show

the standard deviation from the mean run time for each data point, over a sample of 15 runs. The number of ResourceManagers is calculated as:

$$node\_count + node\_count * app\_count$$



(a) Reschedule delay 0s

(b) Reschedule delay 0.5s

Figure 5.12 – Mean total time to run the PlanetLab platform benchmark script for combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

Since PlanetLab is a testbed that does not support reproducible experimentation [56], the results obtained in this benchmark must be analyzed with this factor in mind. The plots show a much more important variation of the run duration between runs compared to the Dummy and ns-3 cases. This variation increases with an increase in the number of ResourceManagers. This is to be expected due to unstable nature of the response times in the Internet, which adds variability to the time needed to complete jobs.

Running the PlanetLab script for 2500 ResourceManagers, i.e., around 50 nodes with 50 applications each, takes around 4 hours in the worst case, when using a reschedule delay of 0 seconds and 50 threads.

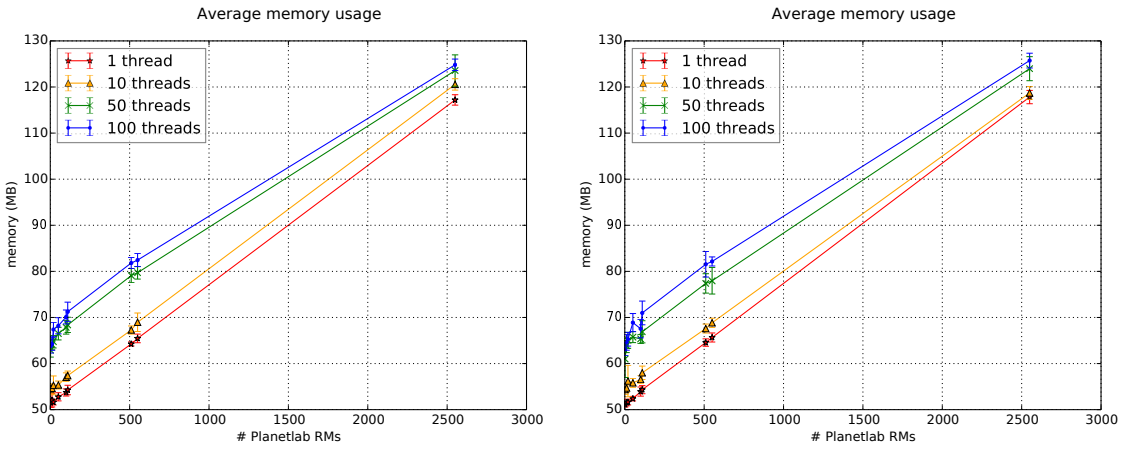
The use of fewer threads seem to consistently reduce the run time, with better results when using a reschedule delay of 0.5 seconds. However, due to the very variable and unpredictable nature of the PlanetLab environment, different results could be obtained if the benchmark was executed at a different time or using different hosts. As an example, the round trip time of a packet between the same two hosts on the Internet can experience abrupt and unpredictable variations during the day. These variations can have a strong impact on the duration of the experiment mean time and standard deviation.

5.3.2 Memory Performance

Figure 5.13 shows the mean average memory used by the benchmark script for different number of ResourceManagers and threads, and values of reschedule delays of 0 and 0.5 seconds.

Memory usage was measured with the *psutil* library at a 0.5 second interval sampling frequency. The average of these measurements for each run was used as the sample values to compute the mean memory usage for each data point. The error bars show the standard deviation from the mean memory usage for each data point, over a sample of 15 runs.

Memory performance for the PlanetLab platform benchmark



(a) Reschedule delay 0s

(b) Reschedule delay 0.5s

Figure 5.13 – Mean average memory used during the run of the PlanetLab benchmark script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

The results show that the mean memory used by the PlanetLab script has a greater variability than in the Dummy and ns-3 cases. The NEPI SSH client continuously creates and destroys subprocess objects to execute SSH commands. This can introduce short time variations in the amount of memory used by NEPI that can might be reflected by a low memory sampling frequency. The number of subprocess objects created at a given time is equal to the number of active threads, so the additional memory consumed by the script will be proportional to the number of threads used. This explains the constant offset in the memory usage curves for different number of threads.

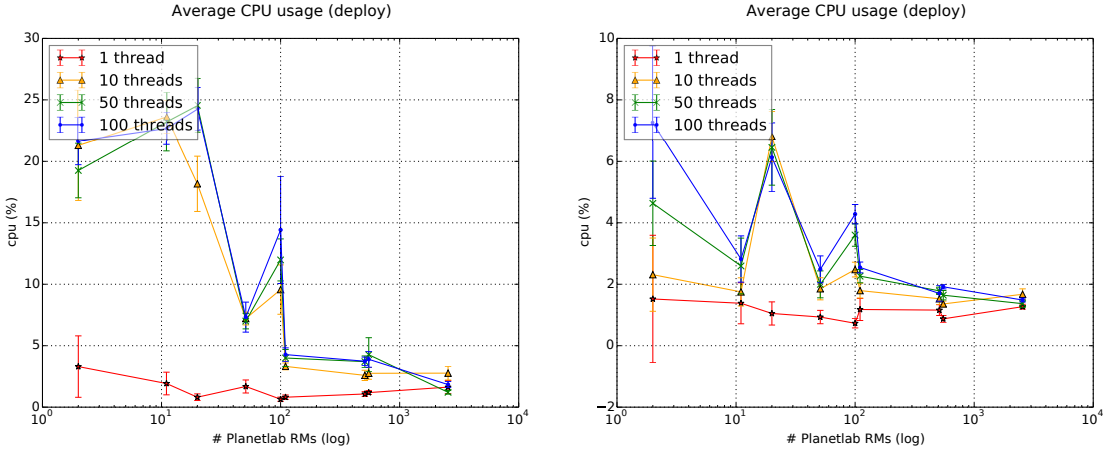
The maximum memory used by the PlanetLab script for 2550 ResourceManagers is around 125MB. In comparison, for the Dummy and ns-3 benchmarks scripts, the maximum memory used for 3000 ResourceManagers was less than 75MB and 135MB, respectively. Like in the previous cases, there is no significant difference between using a reschedule delay of 0 or 0.5 seconds. Also, the use of fewer threads shows a reduced memory consumption.

5.3.3 CPU Performance

Figures 5.14, 5.15, and 5.16 show the mean average CPU used by the PlanetLab benchmark script during the deploy, execution, and release steps of the script run, respectively.

The CPU usage was measured with the *psutil* library at a 0.5 seconds interval sampling frequency. The average of these measurements for each run was used as the sample values to compute the mean CPU usage for each data point. The error bars show the standard deviation from the mean CPU usage for each data point, over a sample of 15 runs.

CPU performance for the PlanetLab platform benchmark during deployment

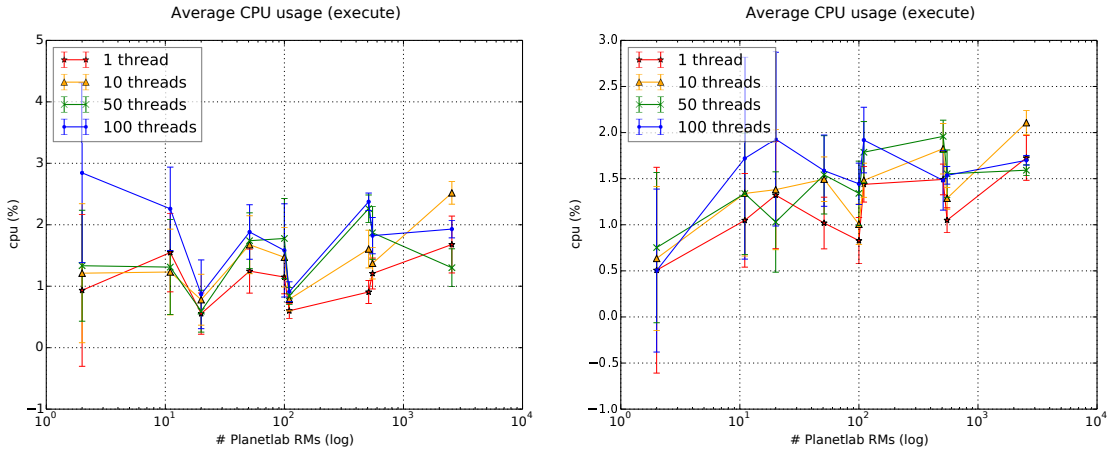


(a) Reschedule delay 0s

(b) Reschedule delay 0.5s

Figure 5.14 – Mean average CPU used during the deploy step of the PlanetLab script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

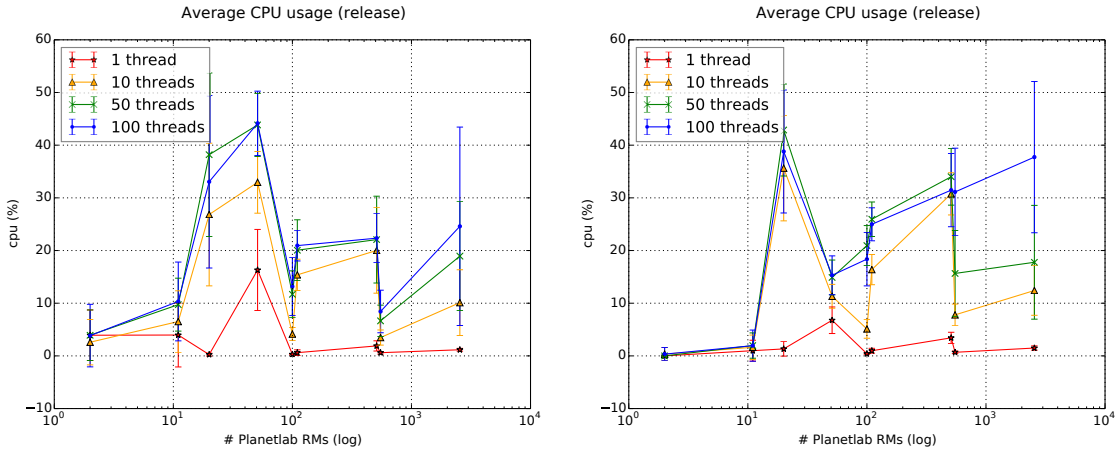
CPU performance for the PlanetLab platform benchmark during execution



(a) Reschedule delay 0s (b) Reschedule delay 0.5s

Figure 5.15 – Mean average CPU used during the execution step of the PlanetLab script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

CPU performance for the PlanetLab platform benchmark during release



(a) Reschedule delay 0s (b) Reschedule delay 0.5s

Figure 5.16 – Mean average CPU used during the release step of the PlanetLab script for different combinations of ResourceManager number, threads number, and reschedule delay. Error bars represent the standard deviation from the mean for each data point over a sample of 15 runs.

Once more the CPU usage shows an important variability across runs, and again different run steps present distinct usage patterns. The CPU usage is much less intensive than in the Dummy and ns-3 cases, being always under 50%, and under 3% during the execution step. The reason of this is that the PlanetLab RMs spend most of their

times waiting for SSH replies, so the experiment execution is I/O bound instead of CPU bound. Using 1 thread is again consistently more efficient than using more threads.

5.4 The Cost of Generic Orchestration

Algorithms that attempt to be generic might penalize performance for not being optimized for specific cases. The proposed orchestration algorithm is generic in the sense that it is capable to deal with arbitrary orchestration tasks on arbitrary experimentation platforms. Despite this, it can scale to handle a large number of resources.

The results presented in this chapter indicate that the algorithm is capable of resolving orchestration in linear time, and with linear memory consumption, with respect to the number of resources in an experiment. Whereas this was shown for the Dummy and the ns-3 benchmarks, the results for the PlanetLab benchmark are less conclusive. Extending the evaluation of this last benchmark to more cases would provide better insight on the behavior of the orchestration algorithm for PlanetLab.

The orchestration algorithm is also able to support reproducible experimentation, since it is capable of reproducing the same orchestration steps across experiment runs. However, this does not guarantee experiment repeatability, which is a property of each platform. Platforms like PlanetLab that introduce unpredictable events do not allow repeatability, or even reproducibility. In contrast, controllable synthetic platforms like ns-3 can support perfect repeatability.

The results presented in this chapter also show that for the evaluated platforms, often a better performance is obtained by using fewer worker threads for the experiment orchestration. This is mostly due to the poor performance of Python for multi-threaded programs.

It would be possible to improve the performance of NEPI by re-implementing the Scheduler in C in order to by-pass the Python GIL and avoid thread management overhead. Re-implementing other core parts of NEPI in C, such as parts of the base ResourceManager class, could also contribute to decrease memory consumption. Users would still be able to write their ResourceManager classes in Python, and in this way keep the benefits of Python scripting, i.e., shorter development times and less programming expertise required.

Chapter 6

Framework Evaluation: Flexibility

Previous chapters showed how the experiment automation approach proposed in this thesis can be effectively extended to support experimentation on diverse platforms, including simulators, emulators, and testbeds. However, another important objective of this work is to provide flexibility to study arbitrary scenarios for any networking research domain, and even for uses other than experimentation.

This chapter focuses on showing, through concrete examples, that the proposed approach can support a variety of uses including platform management, education, and experimentation. The examples provided cover diverse networking domains and address for the following use cases:

- Platform Maintenance
- Teaching Support
- Federated Experimentation
- Cross-Platform Experimentation
- Multi-Platform Experimentation

6.1 Platform Maintenance: Automating Testing and Administration Tasks

The engineers that develop and maintain platforms, as well the users, often need to run tests, maintenance scripts, or benchmarks to characterise the behavior of the platform. Scripts for testing and maintenance might have different characteristics and requirements than scripts to run experiments. Experiment automation tools that are designed specifically for experimentation might not be adapted for testing and maintenance tasks. From this point of view, the generic approach to model experiments proposed in NEPI is an advantage. It permits to construct flexible scenarios that do not impose limitations on how resources are used or managed. An example of this is the use of NEPI to perform maintenance tasks on testbeds, or to prepare the hosts before running an experiment.

The polymorphic nature of NEPI can be illustrated using the OMF testbed. The OMF framework is designed to orchestrate experiments by exchanging messages using a publish/subscribe protocol, such as XMPP, between a publish/subscribe server and the OMF hosts in the testbed. These messages contain OMF instructions to configure hosts and launch applications, and generally manage resources during experimentation. In order to orchestrate an OMF experiment, an experiment automation tool would send messages with OMF instructions to the publish/subscribe server in the testbed to be delivered to the hosts.

Before running an experiment using an OMF testbed, users must perform some setup tasks such as copying an operating system image to the hosts. Additionally, users usually verify that the necessary software for the experiment is correctly installed and that the network interfaces are up and running. OMF provides scripts that can be manually executed to load and save disk images. To perform additional setup operations, users can manually log into the OMF hosts using SSH to directly execute commands, install drivers and libraries, or copy additional files needed in the experiment.

In Chapter 4, an example was provided showing the use of NEPI to run experiments in OMF testbeds using OMF ResourceManagers that implement the OMF publish/subscribe communication mechanism. However, since NEPI does not bind a host, or any other resource, to a specific RM class, OMF hosts can be modeled both using OMF RMs and Linux RMs. Users can take advantage of this to write a NEPI script that uses Linux RMs to setup and configure OMF hosts, using SSH, before running the experiment using OMF RMs.

As an example, Listing 6.1 shows a NEPI script that uses Linux RMs to perform setup tasks on OMF hosts in the NitLab [36] OMF testbed. The script copies a disk image to the hosts using the *omf load* script provided by OMF, and then reboots the hosts using the *omf tell* script provided by the NitLab testbed. These commands are executed in the gateway host of the testbed, the same that runs the XMPP server, using SSH and not XMPP. After waiting for the hosts to reboot, the script executes commands to load a wireless driver and restart the *omf rc* service that listens for incoming XMPP messages, on each OMF host. Executing SSH commands on the OMF hosts requires proxying the commands through the testbed's gateway server. This is necessary because OMF hosts in NitLab are not directly accessible from the outside of the network. NEPI does this automatically, using the gateway information exposed by the Linux node RM.

```
1  from nepi.execution.ec import ExperimentController
2  from nepi.execution.resource import ResourceAction, ResourceState
3
4  hosts = ["host1", "host2"]
5  apps = []
6
7  ec = ExperimentController(exp_id="nitos_bootstrap")
8
9  # Testbed gateway host
10 gw_node = ec.register_resource("linux::Node")
11 ec.set(gw_node, "username", "myuser")
12 ec.set(gw_node, "identity", "~/.ssh/id_rsa")
13 ec.set(gw_node, "gateway", "nitlab.inf.uth.gr")
14 ec.set(gw_node, "cleanExperiment", True)
15
16 # Load image to OMF nodes in the experiment
17 load_cmd = "omf load -i nepi_OMF6_VLC_baseline_grid.ndz -t %s" % hosts
```

```
load_app = ec.register_resource("linux::Application")
ec.set(load_app, "command", load_cmd)
ec.register_connection(load_app, gw_node)

# Turn on the OMF nodes for the experiment
reboot_cmd = "omf tell -a on -t %s" % hosts
reboot_app = ec.register_resource("linux::Application")
ec.set(reboot_app, "command", reboot_cmd)
ec.register_connection(reboot_app, gw_node)

ec.register_condition(reboot_app, ResourceAction.START, load_app,
            ResourceState.STOPPED, time="20s")

hosts = hosts.split(",")

# Restart Resource Controller and load ath5k driver on OMF nodes
for hostname in hosts:
    host = hostname.split(".")[-1]
    node = ec.register_resource("linux::Node")
    ec.set(node, "hostname", host)
    ec.set(node, "username", "root")
    ec.set(node, "gatewayUser", "myuser")
    ec.set(node, "identity", "~/.ssh/id_rsa")
    ec.set(node, "gateway", "nitlab.inf.uth.gr")
    ec.set(node, "cleanExperiment", True)
    ec.register_condition(node, ResourceAction.DEPLOY, reboot_app,
            ResourceState.STOPPED, time="30s")

    app = ec.register_resource("linux::Application")
    ec.set(app, "command", "modprobe ath5k && ip a | grep wlan0 && service omf_rc restart")
    ec.register_connection(app, node)

    apps.append(app)

ec.deploy(wait_all_ready=False)

ec.wait_finished(apps)

ec.shutdown()
```

Listing 6.1 – NEPI script to execute setup commands on OMF hosts in the NitLab testbed using Linux/SSH ResourceManagers.

The complete script is available in the NEPI repository at:

`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/examples/omf/nitos_testbed_bootstrap.py`

6.2 Teaching Support: Helping Students and Teachers in the Classroom

Conducting network experiments often requires advanced technical skills that students might not have yet acquired. Whether knowledge in programming languages or skills in systems administration are required, students can take advantage of tools that automate experimentation to focus on learning course concepts without being held back by technical difficulties.

NEPI facilitates a top down approach to learning about networking, where students can be initially abstracted from details, to dig deeper once they have mastered the basic concepts. With NEPI, students are able to run experiments in different platforms without knowing the details about how hosts are accessed and controlled. However, they do

need to understand network concepts and the experiment modeling granularity of the platforms. NEPI describes experiments from a high-level point of view, but exposing a maximum of configuration information about the resources. The Python scripting language, used to run experiments in NEPI, is well suited for fast prototyping and allows to easily get the hands on the experimentation.

To facilitate the work of students, teachers can provide example scripts, or even extend NEPI with new ResourceManagers, to simplify experimentation tasks for a course. For instance, a teacher could extend the Linux application ResourceManager to perform some specific network measurement using tcpdump, and provide a NEPI script that students can run to collect and then analyze data. Then, the teacher could ask the students to play with the configuration parameters in the script and compare the results. Furthermore, teachers can take advantage of the interactive execution in NEPI to simplify the exploration of a networking technology. Students can use NEPI to incrementally add hosts, launch applications, and make measurements.

In 2014, students of the *Laboratoire d'Informatique Paris 6* used NEPI to investigate denial of service and information disclosure attacks on OpenFlow[1]. To accomplish this, they implemented NEPI scripts to automate the deployment of Open vSwitch overlays on PlanetLab. Having the overlays deployed, they could manually configure OpenFlow controllers, run the attacks, and perform measurements. This is an example of how NEPI can be used to simplify technically challenging tasks, i.e., overlay deployment, to let students focus on the objectives of the course, i.e., learning about OpenFlow and performing attacks and measurements. The results and NEPI scripts for this project can be found at:

`http://nepi.inria.fr/UseCases/DosOpenflow`

Listing 6.2 shows a script to deploy a two-node Open vSwitch overlay in PlanetLab, where one node models a switch and the other a client. A Open vSwitch RM is added to one of the hosts and a port is configured on the switch. On the other host, a TAP device is configured. The hosts are connected using a tunnel, with the port and the TAP as endpoints.

```
1  from nepi.execution.ec import ExperimentController
2
3  def add_host(ec, hostname, username, ssh_key):
4      host = ec.register_resource("planetlab::Node")
5      ec.set(host, "hostname", hostname)
6      ec.set(host, "username", username)
7      ec.set(host, "identity", ssh_key)
8
9      return host
10
11 def add_switch(ec, host, br_name, ip_pref, controller_ip, port):
12     switch = ec.register_resource("planetlab::OVSSwitch")
13     ec.set(switch, "bridge_name", br_name)
14     ec.set(switch, "virtual_ip_pref", ip_pref)
15     ec.set(switch, "controller_ip", controller_ip)
16     ec.set(switch, "controller_port", port)
17     ec.register_connection(switch, host)
18
```

[1]The project subject is available online at `http://www-rp.lip6.fr/~fourmaux/PRes2013.html#sujet8`.

```
19 def add_tap(ec, host, ip, prefixlen, peer_ip):
20     tap = ec.register_resource("planetlab::Tap")
21     ec.set(tap, "ip", ip)
22     ec.set(tap, "prefix", prefixlen)
23     ec.set(tap, "pointopoint", peer_ip)
24 
25     ec.register_connection(tap, host)
26 
27     return tap
28 
29 add_ovs_port(ec, switch, port_name, network):
30     port = ec.register_resource("planetlab::OVSPort")
31     ec.set(port, "port_name", port_name)
32     ec.set(port, "network", network)
33 
34     ec.register_connection(port, switch)
35 
36     return switch
37 
38 hostnames = ["host1", "host2"]
39 
40 ec = ExperimentController(exp_id="ovs_overlay")
41 
42 switch_host = add_host(ec, hostname, username, ssh_key)
43 switch = add_switch(ec, switch_host, "nepi_bridge", "192.169.1.1/24",
44     "1.1.1.1", "6633")
45 port = add_ovs_port(ec, switch, "nepi_port", "192.168.1.0")
46 
47 client_host = add_host(ec, hostname, username, ssh_key)
48 tap = add_tap(ec, client_host, "192.168.1.2", "24", "192.168.1.1")
49 
50 tunnel = ec.register_resource("linux::UdpTunnel")
51 ec.register_connection(port, tunnel)
52 ec.register_connection(tunnel, tap)
```

Listing 6.2 – NEPI script to deploy a two host Open vSwitch overlay on PlanetLab.

Other NEPI examples to describe Open vSwitch overlays in PlanetLab are available online in the NEPI repository at:

`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/examples/openvswitch`

6.3 Federated Experimentation: Supporting Experimentation On Testbed Federations

A part of the development of NEPI was done in the context of two European testbed federation projects: OpenLab [49] and Fed4FIRE [104]. These two closely related projects focus on standardizing experimentation services across testbeds. The main objective of these federation projects is to give access to a larger and more diverse set of resources to users of individual testbeds.

Federations not only give users access to more resources, they also provide common experimentation services and tools to run experiments across testbeds. Testbeds that join a federation are often constrained to adopt some common software to provide minimum required services. Services featured by a federation range from unified authorization, authentication, and resource usage policies, to synchronized resource discovery, reservation, and provisioning mechanisms. These services are paired with experiment orchestration tools capable of providing homogeneous experimental control for all resources in the federation. NEPI is one of such tools.

Federation aside, NEPI can be seen as implementing an exo-federation approach to experimentation. It provides a unified interface to manage experiments across testbeds, like a federation does, but without requiring modifications inside the testbeds. This exo-federation approach, while it imposes zero constraints on testbeds and can be adapted to arbitrary platforms, can not provide the same degree of integration that the full federation approach provides. For instance, NEPI users can use in a same experiment resources from different testbeds, but if the testbeds are not federated with a common account, users must provide authentication credentials for each of them.

OpenLab and Fed4FIRE propose a fully integrated federation approach, where users can access resources on all testbeds using a single user account and credentials. NEPI can be extended to support managing federated resource by implementing the corresponding ResourceManagers.

An example of NEPI usage in Fed4FIRE is the Geo-Cloud study [105]. Geo-Cloud was one of the studies sponsored with the Fed4FIREd open calls. The objective of this study was to evaluate the feasibility of employing cloud computing for real-time on-demand processing of satellite images. PlanetLab was used to obtain a realistic model of the network impairments in a scenario were satellite data was downloaded at specific locations around the globe, i.e., network ground stations. The data retrieved from the satellites traveled across the Internet from the ground stations to a central cloud computing infrastructure for processing. Once processed, the data could be accessed by end users around the world through a web service. NEPI was used to measure latency, loss-rate, and bandwidth from 30 PlanetLab hosts to a central host location, using iperf and ping. The study lasted 6 hours and consisted on establishing connections from one host to the central host with an interval of 1 second.

The Geo-Cloud study relied on NEPI's support for PlanetLab hosts using PlanetLab specific services, instead of the more general federation services. Both OpenLab and Fed4FIRE propose the standardization of experimentation services through the Slice Federation Architecture (SFA) for resource discovery, reservation, and provisioning, and the Federated Resource Control Protocol (FRCP), derived from OMF, for experiment execution and result collection.

In order to support the integrated federation approach proposed by OpenLab and Fed4FIRE, NEPI implements ResourceManagers capable of controlling resources using both SFA and FRCP. An example of the configuration of such ResourceManager is given in Listing 6.3. In this example, credentials are given for both SFA and FRCP separately. This is necessary at the moment because the federation framework is still under development and the authentication for the different services is not yet integrated. However, in the future a single set of credentials could be used per user to authenticate and gain access to all services.

```
node = ec.register_resource("omf::sfa::Node")
ec.set(node, "hostname", hostname)
ec.set(node, "slicename" slicename)
ec.set(node, "sfauser",  sfa_user)
ec.set(node, "sfaPrivateKey", sfa_ssh_key)
ec.set(node, "xmppServer", frcp_server)
ec.set(node, "xmppPassword", frcp_password)
ec.set(node, "xmppUser", frcp_user)
```

```
9 ec.set(node "xmppPort", frcp_port)
```

Listing 6.3 – Definition of a NEPI node using SFA and FRCP credetials.

Since the OMF protocol is an implementation of FRCP compatible with the OpenLab and Fed4FIRE federations, the OMF node ResourceManager was extended to implement a federated node ResourceManager. Additionally, in order to support discovery and provisioning of resources using SFA, a new SFA client was implemented for the federated node RM. The work to support federated resources in NEPI is still ongoing, although partially functional. A first version of a federated node RM was tested on the iMinds OMF testbed. A demo of a NEPI experiment using SFA/FRCP iMinds hosts and PlanetLab hosts is available online at:

`http://nepi.inria.fr/UseCases/VLCCCNStreamingExperiment`

6.4 Cross-Platform Experimentation: Crossing the Boundaries Between Platforms

Combining simulated, emulated, and live resources in a single experiment permits to overcome platform limitations. For instance, the size of the network in an experiment can be scaled beyond the available physical hosts in a testbed by emulating additional hosts, or real traffic coming from a live network can be sent into a simulated network to increase the realism of the simulation. However, most platforms are not integrated with each other by default, and tools and frameworks to automate experimentation often focus on a single platform, or a limited set of platforms.

Being able to use resources from different platforms in a same experiment is one of the initial requirements of the work presented in this thesis. The idea of modeling resources in a uniform way, regardless of the platform they belong to, derives from this requirement. NEPI was designed to allow the integration of resources from heterogeneous platforms in a single experiment. In particular, specific ResourceManagers were implemented in NEPI to integrate ns-3 simulations and DCE emulations with Linux and PlanetLab testbeds. These ResourceManagers can be used to run distributed simulations using multiple physical hosts, or to integrate simulated and live resources into hybrid networks [101].

Figures 6.1 and Figures 6.2 depict two cross-platform experiment scenarios supported in NEPI, mixing ns-3 simulations with live hosts. In the first scenario, a distributed simulation scenario, two ns-3 simulations in different physical hosts are interconnected through a tunnel. In the second scenario, an hybrid experiment scenario, a ns-3 network exchanges traffic with a live host through a file descriptor between an ns-3 FdNetDevice and a virtual TAP device in the live host.

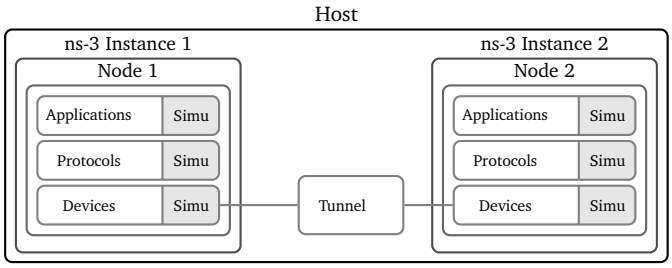


Figure 6.1 – Distributed simulation scenario: ns-3 instances interconnected through a tunnel at the network device level to scale the simulation size.

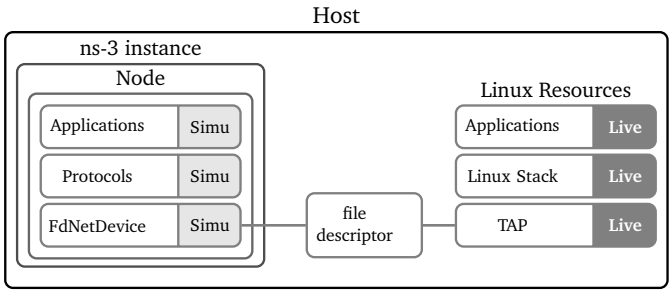


Figure 6.2 – Hybrid experiments scenario: a simulated ns-3 network interconnected to a live host to exchange traffic, using an ns-3 FdNetDevice and a Linux TAP device.

Listing 6.4 shows an example NEPI script for the distributed simulation scenario. In the script, two ns-3 simulations running on different PlanetLab hosts are interconnected using a tunnel. Each simulation consists of only one node and one interface of type *ns3::FdNetDevice*. The *ns3::FdNetDevice* is an ns-3 device model that allows to exchange traffic between a simulated node and the outside world. In this case, the two *ns3::FdNetDevices* are connected using a *planetlab::ns3::FdUdpTunnel* RM provided by NEPI, which establishes a UDP connection between the devices across the Internet.

```
from nepi.execution.ec import ExperimentController

def add_host(ec, hostname, username, ssh_key):
    host = ec.register_resource("planetlab::Node")
    ec.set(host, "hostname", hostname)
    ec.set(host, "username", username)
    ec.set(host, "identity", ssh_key)

    return host

def add_simu(ec, host):
    simu = ec.register_resource("linux::ns3::Simulation")
    ec.set(simu1, "simulatorImplementationType", "ns3::RealtimeSimulatorImpl")
    ec.set(simu1, "checksumEnabled", True)
    ec.register_connection(simu, host)

    return host

def add_node(ec, simu):
    node = ec.register_resource("ns3::Node")
    ec.set(node, "enableStack", True)
    ec.register_connection(node, simu)

```

```
    return node

def add_fd_device(ec, node, ip, prefixlen):
    dev = ec.register_resource("ns3::FdNetDevice")
    ec.set(dev, "ip", ip)
    ec.set(fddev1, "prefix", perfixlen)
    ec.register_connection(node, dev)

    return dev

hostname1 = "host1"
hostname2 = "host2"
username = "myuser"
ssh_key = "~/.ssh/id_rsa"

ec = ExperimentController(exp_id="distributed_simulation")

host1 = add_host(ec, hostname1, username, ssh_key)
simu1 = add_simu(ec, host1)
node1 = add_node(ec, simu1)
fddev1 = add_fd_device(ec, node, "10.0.0.1", "30")

host2 = add_host(ec, hostname2, username, ssh_key)
simu2 = add_simu(ec, host2)
node2 = add_node(ec, simu2)
fddev2 = add_fd_device(ec, node, "10.0.0.2", "30")

tunnel = ec.register_resource("planetlab::ns3::FdUdpTunnel")
ec.register_connection(tunnel, fddev1)
ec.register_connection(tunnel, fddev2)

ping = ec.register_resource("ns3::V4Ping")
ec.set (ping, "Remote", "10.0.0.2")
ec.set (ping, "Interval", "1s")
ec.set (ping, "Verbose", True)
ec.set (ping, "StartTime", "0s")
ec.set (ping, "StopTime", "20s")
ec.register_connection(ping, node)
```

Listing 6.4 – NEPI script to interconnect two remote simulations in different PlanetLab hosts using a tunnel.

Listing 6.5 shows another example NEPI script for the hybrid experiment scenario. In this case, instead of distributing a simulation across hosts, the simulation is transparently connected to the PlanetLab host where it runs, in order to exchange traffic with the real world. In this example, an *ns3::FdNetDevice* in the simulation is connected to a virtual TAP device in the PlanetLab host, using the *planetlab::ns3::TunTapFdLink* virtual link provided by NEPI.

```
from nepi.execution.ec import ExperimentController

def add_host(ec, hostname, username, ssh_key):
    host = ec.register_resource("planetlab::Node")
    ec.set(host, "hostname", hostname)
    ec.set(host, "username", username)
    ec.set(host, "identity", ssh_key)

    return host

def add_simu(ec, host):
    simu = ec.register_resource("linux::ns3::Simulation")
    ec.set(simu1, "simulatorImplementationType", "ns3::RealtimeSimulatorImpl")
    ec.set(simu1, "checksumEnabled", True)
    ec.register_connection(simu, host)

    return host

def add_node(ec, simu):
    node = ec.register_resource("ns3::Node")
```

```
    ec.set(node, "enableStack", True)
    ec.register_connection(node, simu)

    return node

def add_fd_device(ec, node, ip, prefixlen):
    dev = ec.register_resource("ns3::FdNetDevice")
    ec.set(dev, "ip", ip)
    ec.set(fddev1, "prefix", perfixlen)
    ec.register_connection(node, dev)

    return dev

def add_tap(ec, host, ip, prefixlen, peer_ip):
    tap = ec.register_resource("planetlab::Tap")
    ec.set(tap, "ip", ip)
    ec.set(tap, "prefix", prefix_len)
    ec.set(tap, "pointopoint", peer_ip)

    ec.register_connection(host, tap)

    return tap

hostname = "host"
username = "myuser"
ssh_key = "~/.ssh/id_rsa"

ec = ExperimentController(exp_id="hybrid_simulation")

host = add_host(ec, hostname, username, ssh_key)
tap = add_tap(ec, host, "10.0.0.1", "30", "10.0.0.2")

simu = add_simu(ec, host)
node = add_node(ec, simu)
fddev = add_fd_device(ec, node, "10.0.0.2", "30")

link = ec.register_resource("planetlab::ns3::TunTapFdLink")
ec.register_connection(link, tap)
ec.register_connection(link, fddev)

ping = ec.register_resource("ns3::V4Ping")
ec.set (ping, "Remote", "10.0.0.2")
ec.set (ping, "Interval", "1s")
ec.set (ping, "Verbose", True)
ec.set (ping, "StartTime", "0s")
ec.set (ping, "StopTime", "20s")
ec.register_connection(ping, node)

ec.deploy()
```

Listing 6.5 – NEPI script to interconnect a ns-3 simulation with a live host.

An detailed overview of NEPI's support for cross-platform experimentation using ns-3 is to appear in the 2015 workshop on ns-3 (WNS3) [101]. Complete script examples to describe ns-3 distributed and hybrid networks are available online in the NEPI repository at:

`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/examples/ns3/multi_host`

6.5 Multi-Platform Evaluation: Iterative Network Software Development

Evaluating a same networking technology independently on different platforms can help to gain better understanding of a networking technology or to better test and debug the

behavior of networking software. Network applications and protocols might perform differently depending on the characteristics of the network they are evaluated on. For example, a network application might perform better in Ethernet networks than in wireless networks. Evaluating the application in both types of networks can help to improve its performance for all cases.

Chapter 4 showed how NEPI can be used to describe a same scenario in different platforms, including simulators, emulators, and testbeds. Simulators, emulators, and testbeds can be used complementary to support the development process of networking software [100]. Developing software that is intended for deployment in live networks requires the use of adequate development environments that can provide both controllable conditions for debugging and realistic conditions for testing. Networking software developers can iterate between a controllable emulation environment and a more realistic live environment to incrementally add features and validate their software.

NEPI can be used as a networking software development environment to incrementally add new features and to test and validate those features by iterating between different platforms. A NEPI script can be incrementally modified along the life of the software to take into account added features or changes in the software requirements. Also, NEPI can be used for automated testing by running scripts in batch and automatically collecting results, or to interactively debug errors, or stress-test a software implementation. A same experiment description can be re-used with small modifications to scan network parameter ranges, and to systematically study the behavior of a network software on different platforms.

The use of NEPI to combine PlanetLab and ns-3/DCE for iterative software development was demonstrated in ICN 2015 for CCNx [106]. CCNx is an implementation of the Content Centric Networking (CCN) [107, 108] architecture. CCN builds around the idea of assigning routable addresses to pieces of information in a network, know as content, rather than to the physical host location providing the content, as it is proposed by the IP paradigm.

Listings 6.6 and 6.7 show an example where the ccnpeek and ccnpoke CCNx [107] commands are used to retrieve content from a node, using Linux and ns-3/DCE platforms respectively. In both scripts, the same resource abstractions are used to model the CCNx scenario, e.g., CCND, CCNPeek, CCNPoke. Although RMs might vary in the configuration they expose, it is possible to map resource models from one platform to another. The mapping between scenarios on different platforms can be partially automated by using the ExperimentController's automatic topology generation capabilities described in Section 3.4.3.

```
1  from nepi.execution.ec import ExperimentController
2
3  ec = ExperimentController(exp_id="linux_ccnpeek")
4
5  host = ec.register_resource("linux::Node")
6  ec.set(host, "hostname", "host1")
7  ec.set(host, "username", "myuser")
8  ec.set(host, "identity", "~/.ssh/id_rsa")
9
10 ccnd = ec.register_resource("linux::CCND")
11 ec.register_connection(ccnd, host)
12
```

```
peek = ec.register_resource("linux::CCNPeek")
ec.set(peek, "contentName", "ccnx:/chunk0")
ec.register_connection(peek, ccnd)

poke = ec.register_resource("linux::CCNPoke")
ec.set(poke, "contentName", "ccnx:/chunk0")
ec.set(poke, "content", "DATA")
ec.register_connection(poke, ccnd)

ec.register_condition(peek, ResourceAction.START, poke,
    ResourceState.STARTED)

ec.deploy()
```

Listing 6.6 – NEPI script to execute the CCNx's ccnpeek and ccnpoke commands in a Linux host.

```
from nepi.execution.ec import ExperimentController

ec = ExperimentController(exp_id="linux_ccnpeek")

host = ec.register_resource("linux::Node")
ec.set(host, "hostname", "host1")
ec.set(host, "username", "myuser")
ec.set(host, "identity", "~/.ssh/id_rsa")

simu = ec.register_resource("linux::ns3::Simulation")
ec.register_connection(simu, node)

node = ec.register_resource("ns3::Node")
ec.set(node, "enableStack", True)
ec.register_connection(node, simu)

ccnd = ec.register_resource("linux::ns3::dce::CCND")
ec.set (ccnd, "stackSize", 1<<20)
ec.set (ccnd, "StartTime", "1s")
ec.register_connection(ccnd, node)

ccnpoke = ec.register_resource("linux::ns3::dce::CCNPoke")
ec.set (ccnpoke, "contentName", "ccnx:/chunk0")
ec.set (ccnpoke, "content", "DATA")
ec.set (ccnpoke, "stackSize", 1<<20)
ec.set (ccnpoke, "StartTime", "2s")
ec.register_connection(ccnpoke, node)

ccnpeek = ec.register_resource("linux::ns3::dce::CCNPeek")
ec.set (ccnpeek, "contentName", "ccnx:/chunk0")
ec.set (ccnpeek, "stackSize", 1<<20)
ec.set (ccnpeek, "StartTime", "4s")
ec.set (ccnpeek, "StopTime", "20s")
ec.register_connection(ccnpeek, node)

ec.deploy()
```

Listing 6.7 – NEPI script to execute the CCNx's ccnpeek and ccnpoke commands using ns-3/DCE.

More examples of NEPI scripts to model CCNx experiments with ns-3/DCE and Planetlab are available online in the NEPI repository at:

`http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/examples/ccn_emu_live`

6.6 The Cost of Flexibility

NEPI's ability to flexibly model arbitrary scenarios is made possible by the generic approach taken to construct the framework. This flexibility comes at the cost of having to

implement specific ResourceManagers to model the platforms resources used in a scenario. However, once a RM is implemented it can be shared with other users, resulting in a one time effort that can be capitalized by others.

Another consideration, in particular for cross-platform and multi-platform scenarios, is that while NEPI provides a uniform interface to manage experiments on different platforms, users still need to invest time to understand platform basics. NEPI does not completely abstract users from the platforms they use, and while this is positive because it allows experimenters to know and have control over what they are doing, it also means that they are constrained to acquire certain level of knowledge about all platforms in an experiment.

It is difficult to estimate how much time is needed to conduct a study using NEPI, since this depends largely on the characteristics of the scenario and the platforms used, as well as on the experience of the experimenter. Based on the experiments conducted so far with NEPI, the time needed to implement a NEPI script for a scenario, including the time needed to get familiar with NEPI and with the platforms in the experiment, can range from few days to a few weeks if additional RMs need to be implemented. A user familiar with NEPI can implement a script for a scenario in few hours, depending on the complexity of the scenario.

Chapter 7

Conclusions

This thesis presented a generic approach to automate network experiments on arbitrary experimentation platforms and for arbitrary networking research domains. The proposed approach relies on making the experimentation process independent of any platform or networking domain by abstracting the experiment life cycle into generic steps. These generic steps are used as the basis to define a generic automation architecture composed of three elements: a generic network experiment model (GNEM), a generic network experimentation interface (GNEPI), and a generic network orchestration engine (GOE). Through these elements, the architecture provides uniform experiment description and experiment orchestration mechanisms across platforms. Using a same architecture to automate experimentation on different platforms permits to enforce a rigorous experimentation methodology in a platform independent way, and simplifies the use of multiple platforms for result cross validation or to model complex scenarios.

To validate the feasibility of the generic experiment automation architecture, the NEPI framework [90, 91, 92] was re-implemented following the approach proposed in this work. NEPI, the Network Experiment Programming Interface, is a framework that supports automation of network experiments on heterogeneous evaluation platforms, including simulators, emulators, and testbeds.

Tool	Domain	Platform	Backend	Workflows	Interactivity
NEPI 2.0 [90, 91, 92]	Any	Any	Any		
NEPI 3.0	Any	Any+	Any+	✓	✓

Table 7.1 – Comparison between NEPI 2.0 and 3.0 versions. NEPI 3.0 adds workflows support and interactivity, and improves support for arbitrary platforms and backends thanks to a more flexible experiment life cycle.

Table D.2 shows a comparison between the original NEPI 2.0 version, and the new NEPI 3.0 version implemented as part of this thesis. This table completes Table D.1 from Chapter 1, and compares the two NEPI implementations based on their ability to support multiple research domains (Domain), to support different platform types, including simulators, emulators, and testbeds (Platform), to be independent of a preinstalled platform management framework (Backend), to support the specification and

control of runtime experiment behavior (Workflows), and to support interactive experiment execution (Interactive).

NEPI 3.0 models experiments based on three abstractions: Experiment, Resource, and Data, as defined by the generic network experiment model (GNEM). Resources are implemented as Python classes derived from a class hierarchy rooted on the abstract ResourceManager class. The ResourceManager class implements the Resource API defined by generic networking experiment interface (GNEPI).

Experiments are implemented by the ExperimentController class, which exposes to the user the Experiment API, also defined by the GNEPI. A Scheduler component included in the ExperimentController implements the generic orchestration engine (GOE), using an on-line black-box scheduling algorithm. This algorithm is capable of resolving the correct order of execution for the life cycle steps of arbitrary resources in an experiment, taking into account precedence dependencies between resources and workflow dependencies given by the user.

NEPI supports extensibility to arbitrary resources in any platform through the ResourceManager class hierarchy. New resources can be controlled by creating specific ResourceManager classes and implementing the corresponding Resource API methods. NEPI 3.0 currently supports running experiments on Linux testbeds, using SSH key authentication, PlanetLab testbeds [11], OMF testbeds [85], the ns-3 simulator [13], the ns-3 DCE emulation extension [15], and the NetNS emulator.

The efficiency of the algorithm was evaluated with a benchmark replicated on three different platforms: a Dummy platform, the ns-3 simulator, and PlanetLab. In all cases NEPI was capable of orchestrating experiments with a large number of resources, i.e., thousands of resources. NEPI showed to be able to resolve experiment orchestration in linear time with respect to the number of resources, and with linear memory consumption.

The flexibility of the framework to model arbitrary networking scenarios was shown through concrete examples with different networking technologies and for different use cases. These use cases included the use of NEPI for education, platform management, experimentation in testbed federation environments, and cross-platform and multi-platform experimentation.

7.1 Advantages and Limitations

Several advantages of the chosen automation approach were shown through the evaluation of the extensibility, efficiency, and flexibility of the NEPI framework. It was shown that the proposed approach can be effectively used to automate experimentation on simulators, emulators, and testbeds, and combinations of them. It was also shown that the orchestration algorithm implemented in NEPI is capable of handling very different types of resources and can orchestrate experiments involving thousands of resources in linear time. Finally, it was shown that the framework can be used to model diverse scenarios, involving diverse networking technologies, including wired, wireless, Internet,

and content distribution, and that it can also serve for uses other than experimentation, such as education and platform management.

The framework's evaluation also showed some limitations of the implementation. In particular, even if the framework can be extended to support arbitrary resources on any platform, this requires work, and not all experimenters, teachers, or platform owners are willing to invest time to extend the framework. Most experimenters are likely to choose the experimentation alternative that has already proven reasonably effective and that is already familiar to them. Unless experimenters are confronted with a real need, they might prefer to continue writing multiple ad-hoc scripts and to perform manual operations to run experiments, rather than to learn a new framework, even if this saves time in the long run. The advantages of cross-validating results using multiple platforms or to use a framework that enforces a rigorous experimentation methodology might not provide enough motivation to convince experimenters to adopt or even try out the NEPI framework. Perhaps for the above reasons, the growth of a users community around NEPI has been rather slow. More active efforts to publicise the framework, in particular in classrooms, might have a positive impact on the development of the users community.

7.2 Perspectives

This work offers many perspectives of evolution, including improvements to the framework implementation and the orchestration algorithm, and extensions to enhance usability and add new functionality. The rest of this section presents some of the perspectives for future work.

7.2.1 Improvements

The improvements considered in this section focus on further studying the performance of the framework in order to propose optimizations to the orchestration algorithm and the framework implementation.

7.2.1.1 Orchestration Algorithm Optimization

Many paths can be explored to improve the orchestration algorithm proposed in this work. The competitive analysis of the algorithm will permit to compare it to other variants of on-line scheduling algorithms [95]. A comparison with a white-box version of the algorithm, that explicitly codifies dependencies between resource operations, can help to better quantify the benefits and drawbacks of the chosen black-box approach. Further studying how different parameters, e.g., number of threads or reschedule delay, affect the orchestration performance can help to implement better strategies to reduce the experiment duration. Notably, a strategy to investigate consists in dynamically varying the number of threads and reschedule delay based on the instant state of the system, instead of fixing the values of those parameters at the beginning of the experiment.

7.2.1.2 Time and Memory Optimization

Python is a great language for fast prototyping, it also provides an interactive interpreter and a large base of ready-to-use libraries for diverse tasks. However, when it comes to memory consumption and multi-threaded execution, Python is not the most efficient language. As discussed in Chapter 5, the efficiency of the framework could be improved by re-implementing in C some of its core parts. In particular, re-implementing the Scheduler in C would avoid the performance overhead introduced by Python's Global Interpreter Lock (GIL). This is likely to improve the time performance in particular in multi-core machines. Likewise, re-implementing the base ResourceManager class and other core NEPI classes in C would reduce the amount of memory consumed by NEPI scripts, since basic types in C use less memory than in Python. By re-implementing only critical parts of the framework in C, while exposing functionality to the users in Python, both low resource consumption and fast prototyping support could be achieved.

7.2.2 Extensions

Many extensions can be done to this work. This section presents the extensions that appear as natural continuations of existing aspects of the framework.

7.2.2.1 Rigorous Experimentation

Support for rigorous experimentation is one of the main motivations of the work presented in this thesis. The automation framework incorporates basic functionalities to facilitate experiment reproducibility, result validation, and data archiving. Experiment reproducibility is supported by synthetic experiment descriptions and reproducible orchestration mechanisms, result validation is supported through batch experiment execution and result cross-validation using different platforms, and data archiving is supported though automated collection of experiment data to a local repository.

Support for rigorous experimentation can be further enhanced by introducing user defined comparability metrics to compare results across different platforms [5]. This would help to standardize benchmarking metrics for platforms and encourage experimenters to perform benchmarks in order to provide complete information about the characteristics of the environment used in their studies. Additionally, a logbook could be integrated to the framework, association to a Study entity that would group multiple Experiment entities. A digital logbook would help to keep track of additional information that experimenters might want to store or share for a study.

7.2.2.2 Data Processing and Analysis

Data processing is one of the steps of the generic experiment life cycle defined in Chapter 2, however it was not fully developed in the automation framework. Whereas the ExperimentRunner provides some integration for data processing and analysis as part of the experiment re-run workflow, the design assumption made is that experimenters

would use available Python libraries, e.g., numpy and matplotlib, to process and analyse collected data after experiment termination. Further integration of data processing and analysis primitives into the framework could be valuable, in particular for non experienced users who might be unfamiliar with data processing and plotting techniques. Detailed information about the data structure of the traces to be collected in an experiment could be added by the experimenter to the experiment description. The trace description could include information about the data format and about how to parse relevant data fields. This description could be used after data collection to automatically apply data processing operations to filter, aggregate, and transform data from different traces, and plot results according to instructions provided by the experimenter. Existing tools propose similar data processing frameworks [109, 63].

7.2.2.3 Experiment Scenario Translation

Translating an experiment description to match specific resources and configurations is necessary to replicate experiments on different platforms. Replicating experiments is useful to cross-validate results or to iterate between complementary environments, e.g., emulated and live, to develop networking software. Whereas the framework provides high-level abstractions and a uniform interface to simplify mapping experiments between platforms, the experimenter is still a key part of the description translation and must decide the equivalency between platform resources and configurations. As mentioned in Chapter 3, the problem of mapping arbitrary resources from one platform is a difficult problem. However, it would be possible to develop an intelligent system capable of learning from previous mapping decisions made by experimenters in order to automate the mapping of descriptions between different platforms. The system would need to implement a supervised learning algorithm and make the mapping decisions explicit to the experimenters, so they can be manually verified if necessary.

7.2.2.4 User Interface

Graphical user interfaces can be useful for demonstrations and tutorials. For unexperienced users, such as students that are new to networking, graphical user interfaces can also be more intuitive than scripts. Modelling experiments as graphs of resources, like NEPI does, is compatible with a graphical experiment representation. Resources can simply be depicted as boxes exposing a list of attributes and traces, and linked to other boxes. NEPI 2.0 implemented a graphical user interface, in addition to the scripting interface, that allowed users to drag and drop boxes representing resource on a canvas. However, this GUI was not migrated to NEPI 3.0 because it limited the expressiveness of the experiment description. NEPI 2.0 did not allow users to define workflows, and deciding how to graphically describe workflows for large number of resources is not trivial. The scalability of the graphical representation of experiments was also a limitation in the NEPI 2.0 GUI. Graphically adding and interconnecting hundreds of vertices in a graph can be problematic, and it can render the experiment model unclear and hard to configure. A GUI for NEPI 3.0 must take into account these considerations and be usable in experiments involving a large number of resources.

Appendix A

Experiment Life Cycle Examples

This appendix shows, using examples, that the steps of the generic experiment life cycle proposed in Chapter 2 can be mapped to the life cycles of simulators, emulators, testbeds and testbed federations. The platforms studied in this appendix include two simulators, ns-3 [13] and OMNET++ [14], two emulators, DCE [15] and mininet [16], and two testbeds, Planetlab [11] and OMF [85]. Additionally, two testbed federation facilities, Fed4FIRE [8], and GENI [9], are analysed.

These platforms were chosen as representative examples because, apart from being actively used in the networking research community, they are also under active development so it is expected that they will continue to be used in the future.

A.1 Simulation Life Cycle

A.1.1 The NS-3 Simulator

The ns-3 simulator is a discrete event network simulator for Internet systems. It provides realistic models to simulate the behavior of Internet applications and protocol stack, as well as a variety of physical layer technologies, e.g., Ethernet, Wifi, LTE, etc. Thanks to its modular architecture based on fine grained network models, ns-3 can be used to simulate a large variety of networking scenarios with great detail. ns-3 simulations are also capable of generating and exchanging real network traffic with live networks, making it suitable for cross-platform scenarios.

To run ns-3 simulations, experimenters first write a C++ program describing the device, protocol, and application layers of a network by instantiating, configuring, and interconnecting ns-3 objects. The program specifies the start and stop times of the simulation and applications, as well as the data to generate and collect in the simulation. After writing the simulation program, the experimenters use the *waf* [110] build tool to create a binary file to execute the simulation.

A.1.2 OMNET++ Simulation Environment

OMNET++ is a discrete event simulator for communication networks, multiprocessors, and other distributed systems. Instead of providing simulation components, OMNET++ provides the framework to create those components and to compose them into simulations. Simulations are built from simple modules, which can be extended and grouped into compound modules, creating a reusable component hierarchy. Modules communicate with each other by passing messages, either via a connection gate or directly to a destination module. OMNET++ explicitly separates the simulation logic written in C++, i.e., module behavior, from the simulation description defined in network description (NED) files. Simulation modules are re-usable across simulations, while simulation descriptions are specific to a single simulation. OMNET++ can run several simulation instances in parallel using different seeds. The configuration of the simulation instances to run in parallel is specified in INI configuration files.

To define a OMNET++ simulation, an experimenter first creates a project directory for the experiment, then implements simulation components in C++ or re-uses existing ones, then describes the simulation in a NED file, and finally defines the simulation instances in a INI file. The experimenter must create a Makefile to compile and execute the simulation. In OMNET++, simulations can be manually run and terminated from a graphical integrated development environment (IDE).

A.1.3 Compared Life Cycles

Step	NS-3	OMNET++
Platform Setup	Download & compile simulator sources	
Experiment Conception	Define network topology and measurements	
Experiment Design	Write program in C++	Write C++ modules, NED
		INI, and Makefile files
Experiment Deployment	Compile simulation	
Experiment Execution	Run simulation	
Experiment Monitoring	Monitor simulation execution	
Experiment Termination	Predefined	Interrupt simulation execution
Data Collection	Gather data files	
Data Processing	Analyze data & plot results	

Table A.1 – Comparison between operations in ns-3 and OMNET++ life cycles.

Table A.1 compares the operations performed by experimenters to run simulations in ns-3 and OMNET++, grouping them into the steps of the generic experiment life cycle defined in Chapter 2. Differences are found in the experiment design step in the number and type of files experimenters must implement to run a simulation. Also, in the termination step OMNET++ can be terminated by the experimenter from the GUI while an ns-3 simulation will run until its predefined termination time.

A.2 Emulation Life Cycle

A.2.1 The DCE Emulator

Direct Code Execution (DCE) is an application and protocol layer emulator extension for the ns-3 simulator. It adds to ns-3 the ability to execute unmodified Linux applications and the Linux protocol stack inside a simulation. DCE makes it possible to evaluate the same application binaries that can be used in a live network inside a controlled and parametrizable emulation environment. DCE emulations are created by adding special application emulation models to an ns-3 C++ program. The life cycles of DCE and ns-3 are the same, with the exception that applications to be evaluated in DCE must be independently compiled using special compiler flags.

A.2.2 Mininet Emulator

Mininet is a lightweight emulator to model Ethernet networks of hundreds of nodes in a single computer. It uses Linux Container [103] virtualization to create host processes attached to independent network namespaces. Virtualized networks are constructed by interconnecting host processes through virtual Ethernet pairs (veth). Mininet provides both a command line client and a Python API to design network topologies and execute the same applications that can be executed in live networks. One particular feature of Mininet is that it supports *interactivity*. An experimenter can choose between two procedures to run an experiment: writing and running a Python script, or deploying an emulated network and interacting with it in real time through a command line client.

Mininet was specially designed to evaluate software defined networking (SDN) technologies. It incorporates OpenFlow software switches and SDN controllers. It also provides synthetic topology generation for classical topologies, e.g., linear, star, tree, etc. Other topologies can be created by the experimenter using a Python API.

A.2.3 Compared Life Cycles

Step	DCE	Mininet
Platform Setup	Download & compile and install emulator sources	
Experiment Conception	Define network topology and measurements	
Experiment Design	Write program in C++	a) Write emulation script
Experiment Deployment	Compile applications	b) Deploy network using command line
	Compile emulation	
Experiment Execution	Run program	a) Run script
		b) Manually execute commands
Experiment Monitoring	Monitor program	a) Monitor script
Experiment Termination	Predefined	b) Tear down emulated network
Data Collection	Gather data files	
Data Processing	Analyze data & plot results	

Table A.2 – Comparison between operations in DCE and Mininet life cycles.

Table A.2 compares the life cycles of DCE and Mininet emulators. Whereas the operations performed for both emulators can be mapped to the same generic life cycle steps, Mininet presents two alternatives to run an experiment, by a) running a script or by b) manually interacting with the emulated network.

A.3 Live Experiment Life Cycle

A.3.1 PlanetLab Platform

PlanetLab is a research facility composed of hosts distributed around the globe and interconnected through the Internet. PlanetLab hosts can be used simultaneously by different experimenters, in a transparent way and with relative isolation, thanks to the slicing virtualization technique implemented by the PlanetLab platform. A slice represents a virtual view of the PlanetLab testbed, where virtual machines, called slivers, can be allocated on a group of hosts selected by the experimenter. PlanetLab was specifically conceived to perform Internet studies in the wild. There are several PlanetLab instances in use today, including PlanetLab Central in the US and PlanetLab Europe in Europe. Each instance is an independent administrative domain, which means that the hosts in one PlanetLab instance are managed by independent administrators. PlanetLab Central and PlanetLab Europe allow users from one instance to have access to the resources of the other instance.

The architecture of PlanetLab is logically divided into three planes: *Control Plane*, involving resource access and provisioning, *Data Plane*, involving measurements and data collection, and *Experimental Plane*, involving experiment orchestration. PlanetLab provides native control plane services through the MyPLC interface [40]. MyPLC allows to discover available resources and provision them, i.e., create a virtual machine on a host and grant access to the user. Control plane services are also exposed through other non-native interfaces, such as SFA [83] and MySlice [87]. Data plane services, including instrumentation and data collection, are mostly provided by external interfaces such as TopHat [61].

In order to run an experiment, experimenters start by discovering and provisioning available resources that match the requirements of the experiment. Once hosts are provisioned, the experimenter can access them using SSH. PlanetLab does not provide native experimental plane services, e.g., application scheduling and execution, but a number of external tools were created to orchestrate experiments [72, 3]. These tools mostly require writing scripts to configure hosts and launch applications, using a domain specific language (DSL). Alternatively, experimenters can create their own BASH scripts to deploy and run experiments, perform measurements and collect results, or even manually login to each host to launch applications and collect data.

A.3.2 OMF Platform

OMF is a control, measurement, and management framework for network testbeds. Like PlanetLab, OMF is not a single testbed instance but a technology to manage resources

and experiments in a facility. OMF is used by several wireless testbeds, including the Norbit [37] testbed in Australia, the NitLab [36] testbed in Greece, and the w-iLab.t [35] testbed in Belgium. Resources in these testbeds are not isolated and can not be used simultaneously by different experimenters, they must be reserved in advance. There is yet no standardized reservation service for OMF resources. OMF testbeds employ different resource reservation mechanisms: Norbit puts in place an online calendar where experimenters can indicate when they wish to use a set of hosts and channels, NitLab uses the Nitos Scheduler [111] to enforce time slot reservations, and w-iLab.t complements OMF with Emulab [7] reservation mechanism.

OMF provides experiment orchestration and data collection services, including resource configuration, instrumentation, application execution, and result collection. The OMF architecture is divided into three logical planes: the *Management Plane*, for resource provisioning and loading and saving disk images and rebooting hosts, the *Measurement Plane*, for data collection and storage services using the Measurement Collection Server (MCS) and the Measurement Library (OML), and the *Control Plane*, based on a publish/subscribe messaging system for experiment orchestration.

In order to run an experiment in an OMF testbed, experimenters can write an experiment script using the OMF Experiment Description Language (OEDL). The script details the configuration of the resources and the event-based actions that must occur during the experiment run. Before the script can be executed, the experimenter must use a tool to browse and reserve resources for the experiment. Once the reservation is active, the experimenter must copy a previously created disk image to each host. Only then the OEDL script can be executed to automatically start applications and generate measurements. The OEDL script is interpreted by the OMF Experiment Controller (EC), which translates the resource configuration and the events described in the script into messages of the OMF orchestration protocol. These messages are sent to a central publish/subscribe messaging server, to be delivered to OMF Resource Controller (RC) processes managing each host in the testbed. The RC processes then execute the instructions sent by the Experiment Controller script. After the script ends, the experimenter can retrieve a database with measurements or browse the measurements on a dedicated web page.

A.3.3 Compared Life Cycles

Table A.3 compares the experiment life cycles in PlanetLab and OMF testbeds from the perspective of the experimenter. Once again, the operations performed in both cases can be accommodated into the steps of a generic experiment life cycle, but with several small differences. The slice abstraction is not a concept that existed originally in OMF, however it was later adopted to make OMF a more general framework, capable of supporting federation with slice-enabled facilities.

In PlanetLab resource discovery and provisioning are standardized across testbed instances and resource reservation is not necessary thanks to the native PlanetLab resource isolation. Conversely, in OMF resource discovery and reservation are not standard services, and each OMF testbed instance implements these services in an ad-hoc

Step	PlanetLab	OMF
Platform Setup	Account registration	
	Slice creation	
Experiment Conception	Define network topology and measurements	
Experiment Design	Write BASH/DSL scripts	Write OMF script
Experiment Deployment	Resource discovery	
		Resource reservation
	Resource provisioning	
	Resource configuration	
Experiment Execution	Run BASH/DSL scripts	Run OEDL script
	Manually launch applications	
Experiment Monitoring	Monitor scripts execution	
	Monitor host liveliness	
Experiment Termination	Terminate scripts & applications	
Data Collection	Gather remote data files	
Data Processing	Analyze data & plot results	

Table A.3 – Comparison between operations in PlanetLab and OMF life cycles.

way. Resource provisioning in OMF consists in loading a disk image to each host, and OMF provides a command line script for disk image loading. However, some OMF testbed instances, like w-iLab.t, provide their own mechanism to copy disk images and bootstrap hosts. Resource configuration in OMF is done by the experiment orchestration system during execution, whereas in PlanetLab this is left to the experimenter. PlanetLab provides standard services for resource provisioning but not for execution and monitoring, whereas OMF provides standard services for execution and monitoring but not for resource provisioning.

A.4 Federated Experiment Life Cycle

A.4.1 Fed4Fire Federation

Fed4FIRE is a European Integrating Project (IP) in the topic of Future Internet Research and Experiments (FIRE), funded by the Seventh European Union Framework Programme (FP7). Several partner organisations from more than eight countries participate to Fed4FIRE. Its objective is to establish a federation of FIRE facilities by creating the tools and infrastructures needed to provide transparent resource sharing across the physical and administrative boundaries of facilities. FIRE facilities usually target a specific user community and technology within the Future Internet ecosystem. Fed4FIRE aims at fostering Internet innovation by making it easier to conduct cross-facility experiments to model complex Future Internet scenarios that are hard to construct using the resources provided by a single facility.

In the Fed4FIRE project, facility owners adopt a common framework of tools and implement common interfaces, in order to provide uniform services to give access to resources, orchestrate experiments, and gather data across facilities. The federation architecture is divided into four layers: the *Testbed resources*, providing the physical resources of a facility; the *Testbed management framework*, providing facility specific

management software and the federation management interfaces; the *Broker*, providing the services to glue together the interfaces implemented by individual facilities and provide a unified service federated interface; and the *Experimenter* layer, including all the tools and interfaces that the experimenter uses directly to interact with the federation.

A new experimenter who wants to use Fed4FIRE, and is not already a member of a federated facility, needs to register a new federation account through the federation web *Portal*. Then the experimenter can browse available resources on all facilities and reserve the ones that meet the requirements of the experiment. Testbeds in Fed4FIRE expose their resources and reserve or provision resources using the SFA [83] interface. Any client tool that can generate SFA commands can be used to discover, reserve, and provision resources across the federation. Once resources are available, the experimenter can run the experiment and gather results using a federated experiment orchestration tool. Federated experiment orchestration tools must be able to manage resources using Federated Resource Control Protocol (FRCP) [85], which is the standard adopted in Fed4FIRE to control resources. FRCP is implemented in the OMF framework version 6.0.

A.4.2 GENI Federation

The Global Environment for Network Innovations (GENI) project is testbed federation initiative, funded with support of the National Science Foundation (NSF) in the United States. It has for objective to provide a virtual laboratory to conduct computer network experiments on a large scale and with the possibility of integrating a wide variety of networking technologies. Like Fed4FIRE, GENI aims at being a general-purpose facility to foster Future Internet innovation. However, unlike Fed4FIRE, GENI permits different management frameworks, including PlanetLab, OMF, ORCA [84], and ProtoGENI [82], to coexist on separated federation clusters.

Regardless of the management framework adopted, resources can be listed, reserved, and provisioned. GENI SFA Rspecs have been developed as an extension of the original SFA Rspecs to discover, provision, and reserve resources in GENI. Once the resources are obtained, the experimenter uses one of the available orchestration methods, that corresponds to the selected resources, to run experiments. For instance, if the resources are managed using OMF, then the experimenter can use an OMF script, or if the resources are managed using PlanetLab the experimenter can use SSH to run the experiment and collect data.

To execute experiment scripts and to visualize data, GENI experimenters can use the LabWiki [69] tool. LabWiki is a web interface that provides an integrated view of an experiment, showing the experiment scripts and the data collected during experimentation in a single web page. LabWiki was originally developed as a complement for the OMF framework.

Step	Fed4FIRE	GENI
Platform Setup	Account registration	
	Slice creation	
Experiment Conception	Define network topology and measurements	
Experiment Design	FRCP script	FRCP/Bash script
Experiment Deployment	Resource reservation using SFA/Rspecs	
	Resource reservation using SFA/Rspecs	
	Resource provisioning using SFA/Rspecs	
Experiment Execution	FRCP	FRCP or SSH
Experiment Monitoring	FRCP	FRCP or SSH
Experiment Termination	Release resources	
Data Collection	Gather remote data files	
Data Processing	Analyze data & plot results	

Table A.4 – Comparison between operations in Fed4FIRE and GENI life cycles.

A.4.3 Compared Life Cycles

Table A.4 compares the experiment life cycles in Fed4FIRE and GENI federations. The federated life cycle is comparable to an individual testbed life cycle. It must deal with the operations required to handle physical resources, like resource discovery, reservation, and provisioning. The Fed4FIRE and GENI life cycles differ only in the tools used to allocate resources and orchestrate experiments. GENI admits more diversity in the management frameworks and tools used in the federation, whereas Fed4FIRE enforces a unique management framework for the whole federation.

Appendix B

NEPI API Documentation

This Appendix documents the *nepi.execution* Package which implements the Experiment and Resource APIs.

B.1 Modules

- **nepi.execution.ec**: ExperimentController implementation (Experiment API)
- **nepi.execution.resource**: ResourceController implementation (Resource API)
- **nepi.execution.trace**: Trace implementation
- **nepi.execution.runner**: ExperimentRunner implementation
- **nepi.execution.attribute**: Attribute implementation
- **nepi.execution.scheduler**: ExperimentController Scheduler implementation

B.2 Module nepi.execution.trace

B.2.1 Class TraceAttr

A Trace attribute defines information about a Trace that can be queried

B.2.1.1 Class Variables

Name	Description
ALL	**Value:** "all"
STREAM	**Value:** "stream"
PATH	**Value:** "path"
SIZE	**Value:** "size"

B.2.2 Class Trace

object
└ **nepi.execution.trace.Trace**

A Trace represents information about a Resource that can be collected

B.2.2.1 Methods

__init__(*self*, *name*, *help*, *enabled*= False)

:param name: Name of the Trace
:type name: str

:param help: Description of the Trace
:type help: str

:param enabled: Sets activation state of Trace
:type enabled: bool

name(*self*)

Returns the name of the trace

help(*self*)

Returns the help of the trace

enabled(*self*)

Returns the activation state of Trace

B.3 Module nepi.execution.runner

B.3.1 Class ExperimentRunner

object
└ **nepi.execution.runner.ExperimentRunner**

The ExperimentRunner entity is responsible of re-running an experiment described by an ExperimentController multiple time

B.3.1.1 Methods

run(*self*, *ec*, *min_runs*= 1, *max_runs*= -1, *wait_time*= 0, *wait_guids*= [], *compute_metric_callback*= None, *evaluate_convergence_callback*= None)

Run a same experiment independently multiple times, until the evaluate_convergence_callback function returns True

:param ec: Description of experiment to replicate. The runner takes care of deploying the EC, so ec.deploy() must not be invoked directly before or after invoking runner.run()
:type ec: ExperimentController

:param min_runs: Minimum number of times the experiment must be replicated
:type min_runs: int

:param max_runs: Maximum number of times the experiment can be replicated
:type max_runs: int

:param wait_time: Time to wait in seconds on each run between invoking ec.deploy() and ec.release()
:type wait_time: float

:param wait_guids: List of guids wait for finalization on each run. This list is passed to ec.wait_finished()
:type wait_guids: list

:param compute_metric_callback: User defined function invoked after each experiment run to compute a metric. The metric is usually a network measurement obtained from the data collected during experiment execution. The function is invoked passing the ec and the run number as arguments. It must return the value for the computed metric(s) (usually a single numerical value, but it can be several)

metric = compute_metric_callback(ec, run)

:type compute_metric_callback: function

:param evaluate_convergence_callback: User defined function invoked after computing the metric on each run, to evaluate the experiment was run enough times. It takes the list of cumulated metrics produced by the compute_metric_callback up to the current run, and decided whether the metrics have statistically converged to a meaningful value or not. It must return either True or False

stop = evaluate_convergence_callback(ec, run, metrics)

If stop is True, then the runner will exit
:type evaluate_convergence_callback: function

evaluate_normal_convergence(*self*, *ec*, *run*, *metrics*)

Returns True when the confidence interval of the sample mean is less than 5% of the mean value, for a 95% confidence level, assuming normal distribution of the data

run_experiment(*self*, *filepath*, *wait_time*, *wait_guids*)

Run an experiment based on the description stored in filepath

B.4 Module nepi.execution.attribute

B.4.1 Class Types

Allowed types for the Attribute value

B.4.1.1 Class Variables

Name	Description
String	**Value:** "STRING"
Bool	**Value:** "BOOL"
Enumerate	**Value:** "ENUM"
Double	**Value:** "DOUBLE"
Integer	**Value:** "INTEGER"

B.4.2 Class Flags

Flags to characterize the scope of an Attribute

B.4.2.1 Class Variables

Name	Description
NoRead	**Value:** 1
NoWrite	**Value:** 1 << 1
Design	**Value:** 1 << 2
Construct	**Value:** 1 << 3
Credential	**Value:** 1 << 4 | Design
Filter	**Value:** 1 << 5 | Design
Reserved	**Value:** 1 << 6
Global	**Value:** 1 << 7

B.4.3 Class Attribute

object
└ **nepi.execution.attribute.Attribute**

An Attribute exposes a configuration parameter of a resource

B.4.3.1 Methods

__init__(*self*, *name*, *help*, *type*= Types.String, *flags*= None, *default*= None, *allowed*= None, *range*= None, *set_hook*= None)

:param name: Name of the Attribute
:type name: str

:param help: Description of the Attribute
:type help: str

:param type: The type expected for the attribute value. Should be one of Attribute.Types
:type type: str

:param flags: Defines Attribute behavior (i.e. whether it is read-only, read and write, etc). This parameter must take its values from Attribute.Flags. Flags values can be bitwised
:type flags: hex

:param default: Default value for the Attribute
:type default: Depends on the type of Attribute

:param allowed: List of values that the Attribute can take. This parameter is only meaningful for Enumerate type Attributes
:type allowed: list

:param range: (max, min) tuple with range of possible values for Attributes. This parameter is only meaningful for Integer or Double type Attributes
:type range: (int, int) or (float, float)

:param set_hook: Function that will be executed whenever a new value is set for the Attribute :type set_hook: function

name(*self*)

Returns the name of the Attribute

default(*self*)

Returns the default value of the Attribute

type(*self*)

Returns the type of the Attribute

help(*self*)

Returns the description of the Attribute

flags(*self*)

Returns the flags of the Attribute

allowed(*self*)

Returns the set of allowed values for the Attribute

range(*self*)

Returns the range of allowed numerical values for the Attribute

has_flag(*self*, *flag*)

Returns True if the Attribute has the flag 'flag'

:param flag: Flag to be checked
:type flag: Flags

get_value(*self*)

Returns the value of the Attribute

set_value(*self*, *value*)

Configure a new value for the Attribute

is_valid_value(*self*, *value*)

Attribute subclasses will override this method to add adequate validation

has_changed(*self*)

Returns True if the value has changed from the default

B.4.3.2 Class Variables

Name	Description
value	**Value:** property(get_value, set_value)

B.5 Module nepi.execution.scheduler

B.5.1 Class TaskStatus

Execution state of the Task

B.5.1.1 Class Variables

Name	Description
NEW	**Value:** 0
DONE	**Value:** 1
ERROR	**Value:** 2

B.5.2 Class Task

object
└ **nepi.execution.scheduler.Task**

A Task represents an operation to be executed by the ExperimentController scheduler

B.5.2.1 Methods

__init__(*self*, *timestamp*, *callback*)

:param timestamp: Future execution date of the operation
:type timestamp: str

:param callback: A function to invoke in order to execute the operation
:type callback: function

B.5.3 Class HeapScheduler

object
└ **nepi.execution.scheduler.HeapScheduler**

Create a Heap Scheduler
.. note:
This class is thread safe. All calls to C Extensions are made atomic by the GIL in the CPython implementation. heapq.heappush, heapq.heappop, and list access are therefore thread-safe

B.5.3.1 Methods

pending(*self*)

Returns the list of pending task ids

schedule(*self*, *task*)

Add a task to the queue ordered by task.timestamp and arrival order

:param task: task to schedule
:type task: task

remove(*self*, *tid*)

Remove a task form the queue

:param tid: Id of the task to be removed
:type tid: int

next(*self*)

Get the next task in the queue by timestamp and arrival order

Appendix C

ResourceManager Class Template

This appendix provides a template for the creation of new ResourceManager classes. This code is also available online in the NEPI repository at:

http://nepi.inria.fr/code/nepi/file/43ae08ad10a3/doc/templates/template_rm.py

```
1
2  from nepi.execution.attribute import Attribute, Flags, Types
3  from nepi.execution.resource import ResourceManager, clsinit_copy, \
4          ResourceState
5
6  #clsinit_copy is used to inherit attributes from the parent class
7  @clsinit_copy
8  class RMClass(ResourceManager):
9      # Name that will be used in the NEPI script to identify the resource type
10     _rtype = "platform::RMType"
11
12     # User friendly description of the RM
13     _help = "Describes what this RM does"
14
15     # Name of the platform this RM belongs to
16     _platform = "platform"
17
18     # list of valid connection for this RM
19     _authorized_connections = ["platform::AnotherRMType1" , "platform::AnotherRMType2"]
20
21     @classmethod
22     def _register_attributes(cls):
23         """
24         This method is used to register all the attribute of this RM. Check the
25         file src/execution/attribute.py to see all the fields of this class
26         """
27
28         attribute1 = Attribute("nameOfAttribute1",
29                 "Description of Attribute 1",
30                 flags = Flags.Design)
31
32         attribute2 = Attribute("nameOfAttribute2",
33                 "Description of Attribute 2",
34                 flags = Flags.Design)
35
36         cls._register_attribute(attribute1)
37         cls._register_attribute(attribute2)
38
39     def __init__(self, ec, guid):
40         """
41         Declares and initializes variables of the RM that are not Attributes.
42         Attributes represent resource configuration exposed to the user,
```

```
43          other class variables can declared here for RM internal use.
44          """
45
46          super(RMClass, self).__init__(ec, guid)
47
48      def log_message(self, msg):
49          """
50          Prints a log message adding a identification prefix.
51
52          The log message for the RMClass can be redefined here.
53          It can be used to add information about related resources such as
54          the hostname of the node RM associated to an application RM.
55          """
56
57          return " %s guid %d - %s " % (self._rtype, self.guid, msg)
58
59      def valid_connection(self, guid):
60          """
61          Checks whether the RMClass instance can be connected to the
62          other RM corresponding to the given guid.
63          """
64
65          rm = self.ec.get_resource(guid)
66
67          if rm.get_rtype() not in self._authorized_connections:
68              msg = ("Connection between %s %s and %s %s refused: "
69                      "An Application can be connected only to a Node" ) % \
70                  (self.get_rtype(), self._guid, rm.get_rtype(), guid)
71
72              return False
73
74          elif len(self.connections) != 0 :
75              msg = ("Connection between %s %s and %s %s refused: "
76                      "This Application is already connected" ) % \
77                  (self.get_rtype(), self._guid, rm.get_rtype(), guid)
78              self.debug(msg)
79
80              return False
81
82          return True
83
84      def do_discover(self):
85          """
86          Perform actions required to discover resources matching some criteria
87          specified by the user through the configuration of Attributes.
88          """
89
90          super(RMClass, self).do_discover()
91
92      def do_reserve(self):
93          """
94          Perform actions required to reserve resources matching some criteria
95          specified by the user through the configuration of Attributes.
96          """
97
98          super(RMClass, self).do_reserve()
99
100     def do_provision(self):
101         """
102         Perform actions required to provision a resource in the platform,
103         matching the criteria specified by the user.
104         """
105
106         super(RMClass, self).do_provision()
107
108     def do_deploy(self):
109         """
110         Perform actions required to deploy a resource in the platform.
111
112         Deploying a resource most frequently involves invoking the
113         do_discover and do_provision methods. In order to deploy a
114         resource it might be necessary wait until other associated
115         resource is in a given state, as in the following example:
116
```

```
117         other_rm_list = self.get_connected(OtherRMClass.get_rtype())
118         other_rm = other_rm_list[0]
119 
120         if other_rm.state < ResourceState.READY:
121             self.ec.schedule(self.reschedule_delay, self.deploy)
122 
123         elif other_rm.state == ResourceState.FAILED:
124             msg = "Failed to deploy resource"
125             self.error(msg)
126             raise RuntimeError(msg)
127 
128         else:
129             self.do_discover()
130             self.do_provision()
131 
132             super(RMClass, self).do_deploy()
133         """
134 
135         super(RMClass, self).do_deploy()
136 
137     def do_start(self):
138         """
139         Perform actions required to start a resource in the platform.
140         """
141 
142         super(RMClass, self).do_start()
143 
144     def do_stop(self):
145         """
146         Perform actions required to stop a resource in the platform.
147         """
148 
149         super(RMClass, self).do_stop()
150 
151     def do_release(self):
152         """
153         Perform actions required to release a resource in the platform.
154         """
155 
156         super(RMClass, self).do_release()
157 
158     @property
159     def state(self):
160         """
161         Returns the state of the RM.
162 
163         The state method should never raise an exception, instead if an
164         error occurs it should log the error and invoke the self.do_fail
165         method as follows:
166 
167         self.error(msg, out, err)
168         self.do_fail()
169         """
170 
171         return super(RMClass, self).state
172 
173 
```

Listing C.1 – Template for the creation of new ResourceManager classes

Annexe D

Version Française

Résumé

Plusieurs plates-formes d'évaluation, tels que des simulateurs, des émulateurs et des bancs d'essai, sont couramment utilisées pour développer les technologies réseaux et les étudier empiriquement. Dans la plupart des cas, l'exécution de ces expériences nécessite un travail manuel considérable, ce qui entraîne de nombreuses erreurs et bogues et rend des études difficiles à reproduire. D'autant plus que le manque d'abstraction dans le processus d'expérimentation et l'absence d'uniformité des interfaces et des outils entre les plates-formes réseaux rendent la standardisation des pratiques d'expérimentation rigoureuse difficile. Ces aspects compliquent également l'utilisation des différentes plates-formes dans une même étude pour la validation croisée des résultats. L'automatisation du cycle de vie des expériences réseaux est une solution pour établir un processus d'expérimentation rigoureux tout en rendant les plates-formes plus facilement utilisables. En effet, l'automatisation minimise les interventions humaines en s'appuyant sur des descriptions d'expériences bien définies et vérifiables ainsi que des mécanismes d'orchestration reproductibles. Toutefois, les systèmes d'automatisation d'expérimentation réseau existants ciblent des plates-formes et des domaines de recherche réseau spécifiques, ce qui rend leur généralisation difficile sans pour autant autoriser l'utilisation de plates-formes différentes au sein d'une même étude.

La présent thèse propose une approche générique pour automatiser les expériences réseaux sur toute type de plates-formes réseaux, indépendamment de leur domaine d'application. L'approche proposée est basée sur l'abstraction du cycle de vie de l'expérience en étapes génériques qui sont valides pour des simulateurs, des émulateurs et des bancs d'essai. Une architecture d'expérimentation générique est proposée basée sur ces étapes, composée d'un modèle d'expérience abstrait, d'une interface d'expérimentation universelle, et d'un algorithme d'orchestration générique. Le faisabilité de cette approche est démontrée par la mise en œuvre d'un système capable d'automatiser les expériences dans des simulateurs, des émulateurs, des bancs d'essai, et même des combinaisons de ces plates-formes. Les trois aspects principaux du système sont évalués : son

extensibilité pour s'adapter aux plates-formes hétérogènes, son efficacité pour orchestrer des expériences réseaux et sa flexibilité pour permettre des cas d'utilisation divers. Ceci inclus les usages à but d'enseignement mais aussi la gestion de plates-formes et la mise en scene des expériences complexes regroupant bancs d'essai et fédérant des plates-formes de natures différentes. Les résultats montrent que l'approche proposée peut être utilisée pour automatiser efficacement des expériences sur des plates-formes hétérogènes, pour un large éventail de scénarios.

Le reste de ce résumé est consacré à la description des différents de cette thèse.

Le première chapitre de cette thèse donne un aperçu de l'état de l'art lié à l'automatisation des expériences réseaux. Il décrit les plates-formes d'expérimentation existantes ainsi que les solutions d'automatisation d'expériences, leurs avantages et leurs limites. La principale lacune montrée par l'analyse de l'état de l'art est l'absence d'une solution d'automatisation capable d'automatiser des expériences sur des plates-formes d'évaluation arbitraires, y compris les simulateurs, les émulateurs et les bancs d'essai. Les outils d'automatisation étudiés dans la littérature ne fonctionnent qu'avec un nombre limité de plates-formes ou de technologies réseaux.

Le deuxième chapitre de cette thèse propose une architecture générique pour l'automatisation d'expériences réseaux qui répond aux limites des outils d'automatisation existants. Cette architecture permet l'automatisation des expériences sur des plates-formes arbitraires et pour des scénarios réseaux quelconques. L'architecture proposée est composée de trois éléments : une interface de programmation de réseau générique (GNEPI), un modèle générique d'expérience (GNEM) et un moteur d'orchestration générique (GOE). Le composant GNEPI fournit aux expérimentateurs une interface programmable avec des primitives pour exécuter des expériences réseaux. Cet interface est indépendante du type de plate-forme considéré. Le composant GNEM fournit un modèle pour décrire des expériences réseaux sur la base de trois entités : expérience, ressources et données. Il représente les expériences comme des graphes de ressources. Des règles de flux de travail peuvent aussi être spécifiées par l'utilisateur afin de définir le comportement des ressources lors de l'exécution de l'expérience. Les primitives accessibles par l'interface de programmation et le modèle d'expérience sont dérivées d'un cycle de vie d'expérience commun à toutes les plates-formes. Enfin, le composant GOE est le composant responsable de l'exécution et du contrôle des ressources lors de l'expérimentation. Il utilise un algorithme d'orchestration basé sur l'algorithme List de Graham, créé à l'origine pour résoudre le problème job shop d'ordonnancement en ligne, afin d'établir un ordre d'exécution approprié pour les opérations associées aux ressources dans une expérience.

Le troisième chapitre de cette thèse décrit le framework NEPI pour l'automatisation des expériences réseaux. Ce framework met en oeuvre l'architecture générique pour l'automatisation des expériences réseaux à travers une implémentation en langage Python. Il peut être utilisé et modifié par tous les utilisateurs. NEPI utilise une hiérarchie

de classes de ressources extensible pour adapter des ressources génériques aux besoins de plates-formes spécifiques. Ceci permet d'étendre le framework pour travailler avec des ressources arbitraires dans toute plate-forme. Il permet entre autres fonctionnalités le déploiement interactif des expériences, leur reproduction et la collecte et l'archivage de données.

Le quatrième chapitre de cette thèse montre comment le framework NEPI est capable de supporter des simulateurs, des émulateurs et des bancs d'essai en présentant des exemples concrets qui sont actuellement pris en charge par le framework. Ces plates-formes comprennent le simulateur ns-3, les émulateurs DCE et NetNs, et les bancs d'essai OMF et PlanetLab. Une description des étapes à suivre pour ajouter de nouvelles plates-formes et de nouvelles ressources au framework est également exposée dans cet chapitre. L'ajout d'une nouvelle plate-forme nécessite le développement de modèles de ressources spécifiques et aussi de modules de communication pour interagir avec la plate-forme cible.

Le cinquième chapitre évalue l'efficacité de l'algorithme d'orchestration en utilisant une plate-forme idéale (Dummy) ou la durée des opérations des ressources est entièrement contrôlable. Cette évaluation a pour objectif d'étudier l'impact des différents paramètres de l'algorithme d'orchestration, tels que le nombre de threads utilisé par l'algorithme d'orchestration et la durée des opérations des ressources, sur le temps et la mémoire nécessaire à l'exécution des expériences. Un benchmark est réalisé en utilisant un scénario simple et en augmentant le nombre de ressources utilisées pour évaluer la scalability de l'algorithme. Un benchmark est de nouveau effectué en utilisant le simulateur ns-3 et le banc d'essai PlanetLab afin de comparer les résultats avec ceux de la plate-forme idéale Dummy.

Le sixième chapitre montre la capacité de NEPI à satisfaire plusieurs cas d'utilisation utiles à la recherche en réseaux informatiques et à l'enseignement. Ces cas d'utilisation comprennent l'usage de NEPI dans un projet réseaux dans un cours universitaire, la gestion des plates-formes d'évaluation réseaux et l'expérimentation avec de nombreuses plates-formes en parallèle ou inter-connectées. En particulier, cinq cas sont présentés : l'utilisation de NEPI pour un projet étudiant de niveau master, l'automatisation des tests pour effectuer des tâches d'entretien sur des plates-formes, l'utilisation de NEPI pour supporter des expérimentations dans l'environnement de fédération de bancs d'essai Fed4FIRE, l'utilisation de NEPI pour le développement itératif du logiciel CCNx en utilisant des plates-formes émulées et des bancs d'essai en série et l'utilisation de NEPI pour le déploiement distribué et parallèle des simulations ns-3 sur les bancs d'essai PlanetLab.

Le septième chapitre conclut en présentant les avantages et les limitations de ce travail, avec un analyse des améliorations possibles pour le framework et des perspectives pour continuer les travail présenté dans cette thèse. Parmi les améliorations possibles se trouvent l'optimisation de l'algorithme d'orchestration et l'optimisation de la durée et de la mémoire utilisés par le composant GEO mis en oeuvre par NEPI. Parmi les pistes de

travail liées à cette thèse se trouvent la standardisation des techniques rigoureuses d'expérimentation, l'intégration des primitives pour le traitement et l'analyse de données dans le modèle d'expérience, et la traduction automatisée des scénarios des expériences entre plates-formes. L'ajout d'une interface d'utilisateur graphique est également une perspective de travail. Malgré les difficultés soulevées par sa mise en oeuvre, une interface pourrait rendre NEPI plus attractif et plus accessible.

Mots clés

Expériences réseaux, automatisation, reproductibilité, simulation, émulation, bancs d'essai, fédération des bancs d'essai.

Introduction

Un rôle important de la science dans la société est de produire des connaissances et de l'innovation technologique pour l'amélioration des conditions de vie. Des exemples de ces améliorations sont l'augmentation de l'espérance de vie, la production d'énergies non polluantes, et la favorisation de la croissance économique et du développement culturel. En particulier, l'innovation dans les réseaux informatiques a contribué à favoriser les échanges entre les gens et la création de nouveaux marchés. Ceci a profondément changé la façon dont les gens interagissent avec la technologie, et a rendu la communication électronique aussi indispensable que l'eau du robinet ou de l'électricité.

En tant qu'acteurs principaux du processus d'innovation, les chercheurs et les ingénieurs doivent faire face à plusieurs défis. Dans les domaines tels que les réseaux informatiques, où les nouvelles technologies deviennent rapidement obsolètes, le timing est un facteur essentiel et des idées qui ne sont pas réalisés rapidement risquent de manquer leur opportunité dans le marché. Afin de transférer efficacement l'innovation du laboratoire au grand public, les chercheurs et les ingénieurs réseaux doivent mettre en œuvre dans un court delai des logiciels robustes qui peuvent fonctionner correctement dans les environnements de production. Cela nécessite une évaluation approfondie des idées et prototypes initiaux, ainsi que des implémentations de logiciels intermédiaires et finaux.

En raison de la grande complexité des systèmes de réseaux modernes, l'utilisation des méthodes d'évaluation purement analytiques, même pour étudier des idées simples, est souvent pas adapté. Pour cette raison, les méthodes d'évaluation empiriques sont d'une grande importance dans la recherche informatique appliquée [1, 2]. Ils permettent de fournir une alternative pour surmonter les limites des méthodes analytiques. Les chercheurs et les ingénieurs réseau font un usage intensif des plates-formes d'évaluation empiriques, tels que des simulateurs, des émulateurs et des bancs d'essai, afin de développer et d'évaluer des solutions à des problèmes complexes. Cependant, la manipulation de ces plates-formes peut être fastidieuse et devenir une source d'erreurs. C'est surtout le cas quand de nombreuses technologies et divers facteurs doivent être pris en compte, ou lorsque le travail manuel nécessaire pour mener d'expériences liées aux réseaux est considérable.

Assurer une méthodologie d'expérimentation rigoureuse présente un coût supplémentaire ajouté aux difficultés d'utilisation des plate-formes d'évaluation. Une méthodologie rigoureuse repose en grande partie sur des bonnes pratiques adoptées par les expérimentateurs. Si une plate-forme est difficile à utiliser et nécessite de coûteuses

travaux manuels, plus d'attention et d'efforts sont nécessaires pour éviter les erreurs et pour assurer un processus d'expérimentation contrôlé et reproductible. Une façon d'obtenir un meilleur contrôle et une meilleure reproductibilité dans l'expérimentation est d'utiliser des mécanismes d'automatisation d'expériences. Automatiser les parties techniques d'expériences, tels que le déploiement de l'expérience et la collecte des résultats, peut aider à gagner du temps et éviter les erreurs humaines. L'automatisation permet également d'améliorer la reproductibilité à travers des descriptions d'expériences bien définis et des procédures réutilisables.

Pendant la dernière décennie, différentes solutions ont été proposées dans le domaine des réseaux informatiques pour simplifier l'expérimentation, pour la rendre plus rigoureuse, et pour permettre la construction d'expériences complexes, sous la forme d'outils [3, 4], de frameworks [5, 6, 7], et de fédérations de bancs d'essai [8, 9]. Cependant, les solutions d'automatisation existantes permettent habituellement d'utiliser seulement un sous-ensemble des plates-formes d'évaluation ou ciblent des technologies de recherche en réseaux spécifiques.

L'objectif de cette thèse est triple. Tout d'abord, de proposer une solution générique pour automatiser l'expérimentation sur les plates-formes d'évaluation réseau et tout domaine de recherche sur les réseaux. D'autre part, simplifier l'utilisation de diffèrentes plates-formes dans une même étude. Troisièmement, de fournir un soutien pour l'expérimentation rigoureuse sur toutes les plates-formes.

Terminologie

Plate-forme et environnement. Les termes plate-forme et environnement sont utilisés indifféremment pour désigner à un logiciel ou une infrastructure utilisé pour mener des expériences sur les réseaux. Les plates-formes peuvent être de différents types et pour les besoins de ce travail, elles sont divisées en : simulateurs, émulateurs, bancs d'essai et fédérations de bancs d'essai.

Méthode d'évaluation. Méthode d'évaluation se réfère à la technique utilisée pour réaliser une expérience. Cette thèse considère trois méthodes d'évaluation : la simulation, l'émulation et de l'expérimentation réelle.

Répétabilité, reproductibilité, et réplication. Dans cette thèse, les termes *répétabilité*, *reproductibilité*, et *réplication* sont utilisés pour faire référence à des concepts différents. Tous les trois font référence à l'action de *clonage* d'un aspect d'une expérience. Le mot répétabilité est utilisé pour indiquer les clonage des résultats d'expériences, le mot reproductibilité est utilisé pour indiquer le clonage de la procédure utilisée pour effectuer une expérience, et le mot réplication est utilisé pour indiquer le clonage du scénario utilisé dans une expérience sur une autre plate-forme d'expérimentation. Tous les plate-formes ne permettent pas la répétabilité. En effet, pour cela, il faut pouvoir contrôler les conditions pour obtenir des résultats identiques lors de la reproduction d'une expérience. Par contre, toutes les plates-formes devraient permettre la reproductibilité des expériences. Ceci indique la capacité de rejouer les étapes d'une expérience de telle manière que les résul-

tats obtenus puissent être analysés statistiquement. La réplication d'un scénario sur des plates-formes différentes nécessite certaines similitudes entre les plates-formes pour que les résultats soient comparables.

Hôte et noeud. Les termes hôte et noeud sont utilisés indifféremment pour désigner un dispositif physique, émulé ou simulé dans une expérience.

Une pléthore de plates-formes d'évaluation

La diversité croissante des technologies réseaux et des scénarios d'évaluation a mené à une prolifération de plates-formes d'évaluation sur mesure, crées pour combler des besoins différents. Des nombreuses plates-formes se concentrent sur des technologies ou des domaines de recherche spécifiques, tels que les réseaux sans fil [10, 5] et les services Internet [11, 12], ou utilisent des méthodes d'évaluation spécifiques, comme la simulation [13, 14], l'émulation [15, 16, 17] ou l'expérimentation réel [18, 19, 20]. Les plates-formes peuvent être génériques ou spécifiques. Le première type peut utiliser différentes méthodes d'évaluation pour tester plusieurs technologies [7, 21]. Le deuxième type est limité aux méthodes d'évaluation ou à des scénarios spécifiques [22, 23].

Les expérimentateurs peuvent utiliser ces plates-formes de plusieurs manières. Soit ils peuvent mener une étude avec une seule plate-forme, soit ils peuvent reproduire indépendamment leur étude sur des différentes plates-formes [24, 25] pour obtenir des résultats complémentaires, soit ils peuvent interconnecter des plates-formes différentes afin de construire des scénarios complexes [26].

La diversité des plates-formes d'évaluation dans l'écosystème des réseaux montre le fait qu'aucune plate-forme, ou méthode d'évaluation, n'a été capable de satisfaire tous les besoins d'expérimentation. Alors que les technologies évoluent, il est probable que de nouveaux cas d'utilisation vont encore encourager la création de nouvelles plates-formes et l'utilisation de diverses méthodes d'évaluation. Cette section donne un aperçu des différents types de plates-formes d'évaluation utilisés dans le milieu de la recherche en réseau, et de leurs usages.

Type de Plate-forme

Simulateurs

Les simulateurs imitent le comportement des systèmes réels à travers des modèles qui simplifient l'étude des interactions entre les composants du système. Ils permettent un contrôle précis sur les expériences et donnent des conditions d'évaluation répétable. Cependant, comme les simulateurs simplifient la réalité, dans certains cas, ils peuvent produire des résultats inexactes [27, 28]. Leur atout est qu'ils rendent possible l'étude des scénarios qui seraient prohibitifs ou trop coûteus sur des plates-formes plus réalistes, comme des émulateurs ou des bancs d'essai.

Les simulateurs peuvent être d'usage général ou de domaine spécifique. Ns-2 [29], ns-3 [13] et OMNET++ [14] sont des exemples de simulateurs de réseaux d'usage général. Des exemples de simulateurs de domaine spécifiques sont SENSE [30], pour

les réseaux de capteurs sans fil, OverSim [23], pour les réseaux de superposition et ndnSim [22], pour les réseaux centres sur le contenu (Content Centric Networking).

Voici quelques avantages des simulateurs par rapport aux émulateurs et aux bancs d'essai :

- L'exécution et le débogage des expériences avec des simulations est d'habitude plus simple qu'avec des réseaux réels.
- Les simulateurs permettent de mener des expériences à une plus grand échelle et avec une plus grande variété de technologies réseaux que les environnements réels.
- Les simulateurs permettent d'évaluer des scénarios qui sont prohibitifs sur des environnements réels en raison du coût élevé de l'infrastructure, de l'impossibilité de contrôler le trafic en direct ou à des restrictions d'utilisation.
- Les simulateurs constituent des environnements très contrôlables qui permettent une parfaite répétabilité des expériences.

Le principal inconvénient des simulateurs est leur niveau limité de réalisme. Ils ne sont pas capables d'imiter des comportements imprévisibles, ce qui est nécessaire par exemple lors de la validation des logiciels avant leur mise en œuvre. Les simulateurs à événements discrets peuvent aussi être extrêmement lents lorsque une expérience génère trop d'événements.

Émulateurs

Les émulateurs mélangent des modèles avec des composants réels. Les expériences émulées peuvent avoir des parties synthétiques contrôlables ainsi que des parties réelles. Les émulateurs peuvent être considérées comme offrant une balance entre réalisme et contrôlabilité à trois échelles différentes d'un réseau : l'échelle des dispositifs, l'échelle de protocoles et l'échelle des applications. Deux cas extrêmes peuvent être considérés : les émulateurs software et les émulateurs hardware. Les premièrs, comme DCE [15], utilisent des modèles pour l'échelle des dispositifs mais permettent l'utilisation des protocoles et des applications réelles. Les dernièrs, comme Flexlab [17], permettent utiliser des dispositifs de réseaux réels, tout en offrant la virtualisation des hôtes et des conditions synthétiques pour modéliser le trafic réseau. Par exemple, Flexlab [17] imite des conditions réelles pour le trafic en utilisant des mesures effectuées sur Internet à travers le banc d'essai PlanetLab [11].

D'autres exemples d'émulateurs sont Mininet [16], pour l'émulation OpenFlow, WISER [31] pour l'émulation de protocoles MANET, SUNSHINE [32], pour l'émulation de réseau de capteurs et ModelNet [33] pour l'émulation à grande échelle d'Internet

Les avantages et les inconvénients de l'émulation, par rapport à la simulation et l'expérimentation réel, dépendent des caractéristiques de chaque émulateur. Les principaux facteurs à considérer sont s'ils permettent de mener des expériences de grande taille, s'il permettent de contrôler les conditions d'expérimentation et si le degré de réalisme qu'ils offrent est suffisant.

Banc d'essai

Les bancs d'essai sont des infrastructures réseaux dédiées utilisées pour mener des expériences dans des conditions réels. Dans un banc d'essai, les dispositifs réseaux, les protocoles et les applications utilisées sont réels. L'expérimentation réel sur des bancs d'essai permet d'étudier un système sans simplification et d'exposer des interactions complexes entre les composants du réseau. Ces interactions pourraient passer inaperçu avec des simulateurs ou des émulateurs .

Les bancs d'essai utilisent généralement des frameworks de gestion spécialisés [34] pour contrôler les ressources des expériences et fournir des services aux utilisateurs. Ces services comprennent la gestion des ressources, la gestion de l'expérience, et la gestion des données. Les services de gestion de ressources comprennent l'enregistrement des comptes d'utilisateur, l'authentification des utilisateurs, la découverte de ressources, la réservation de ressources et le provisionnement de ressources. La gestion de l'expérience inclut le déploiement des ressources, l'exécution et le suivi des applications. La gestion de données comprend l'instrumentation et la collecte de données issues des expériences réseaux.

Des exemples de bancs d'essai et leurs frameworks de gestion sont : PlanetLab [11], un banc d'essai pour l'expérimentation sur Internet utilisant MyPLC pour le contrôle des ressources, w-iLab.t [35], NitLab [36] et Norbit [37] pour l'expérimentation sans fil, utilisant OMF [6] pour le contrôle des expériences, Grid5000 [18] pour l'évaluation du logiciel de grille, utilisant OAR [38] pour la réservation de ressources et SensLab [39] pour les réseaux de capteurs à l'aide du logiciel de gestion SensLab.

L'avantage principal des bancs d'essai par rapport aux simulateurs et émulateurs est leur haut niveau de réalisme. Cependant, la plupart des bancs d'essai ciblent des technologies réseaux spécifiques. Afin d'atteindre du réalisme dans les résultats d'une expérience, le banc d'essai choisi doit présenter des caractéristiques précises qui correspondent aux exigences du scénario étudié. Par exemple, un environnement de grille ne pourrait pas refléter fidèlement l'Internet et un banc d'essai pour l'Internet ne sera pas approprié pour l'évaluation fidèle de logiciel de grid computing.

Par rapport aux simulateurs les bancs d'essai présentent les inconvénients suivants :

- La mise en œuvre des expériences dans un banc d'essai est longue car elle implique l'accès et la synchronisation des composants distribués.
- Les bancs d'essai présentent des limitations en termes de taille de réseau et de variété de technologies, puisque les ressources physiques utilisées par les bancs d'essai sont chères et d'habitude d'un même type.
- Les bancs d'essai peuvent imposer des restrictions d'utilisation, par exemple si les ressources sont généralement partagées entre de nombreux utilisateurs, ce que limite leur disponibilité.
- L'environnement réseau dans les bancs d'essai n'est pas entièrement contrôlable ou prévisible, ce qui rend difficile la répétabilité des expériences.

Fédérations banc d'essai

Des fédérations de bancs d'essai ont pour objectif le partage des ressources à travers des domaines administratif gérés de façon indépendante. Cette intégration des bancs d'essai donne accès aux expérimentateurs à un plus grand nombre de ressources et à une plus grande variété de technologies avec un même compte d'utilisateur. Les fédérations fournissent des services à travers des bancs d'essai, de manière à établir des interfaces uniformes pour le contrôle des ressources, le contrôle des expériences et la gestion des données.

Ils existent plusieurs projets de fédération offrant différents degrés d'intégration entre bancs d'essai, ou des services différents, ou qui ciblent des technologies de réseaux différentes. Les fédérations homogènes, comme celle entre PlanetLab Europe et PlanetLab Central [40], exigent que tous les bancs d'essai utilisent un même framework de gestion. Les fédérations hétérogènes, comme celle entre PlanetLab et Emulab [41], sont capables d'intégrer des bancs d'essai qui utilisent des frameworks de gestion différents.

Certaines fédérations utilisent une approche de gestion centralisée, c'est le cas de la fédération PlanetLab-Emulab qui fournit un point d'accès unique pour réserver et accéder aux ressources à travers la fédération. En revanche, d'autres fédérations proposent la gestion décentralisée, à l'aide d'algorithmes et de services distribués pour découvrir et réserver des ressources sur les bancs d'essai. Tel est le cas de GENI [9] aux États-Unis et FIRE [42] en Europe.

Ils existent des fédérations qui se concentrent sur la recherche sur les technologies réseaux spécifiques. C'est le cas de OFELIA [43] pour la recherche OpenFlow, BonFire [44] pour la recherche de services Cloud, CREW [45] pour la recherche de la radio cognitive et WISEBED [46] pour la recherche de capteurs sans fil. Tandis que d'autres fédérations, comme TEFIS [47], Panlab [48], Openlab [49], et Fed4FIRE [8] peut être utilisés dans un large éventail de domaines.

Les fédérations de banc d'essai résolvent plusieurs problèmes de bancs d'essai isolés, tels que les limitation de la taille des expériences et dans la diversité des technologies. Cependant, ils ne traitent pas d'autres limitations des bancs d'essai tels que le manque de contrôlabilité et de répétabilité des expériences.

Modalités d'évaluation

Évaluation sur une plate-forme isolée

L'évaluation sur des plate-formes isolées consiste à mener une étude du réseau en utilisant une seule plate-forme. Un avantage d'utiliser une seule plate-forme est qu'il n'est pas nécessaire de maîtriser plusieurs plates-formes ou d'écrire plusieurs scripts et programmes pour exécuter des expériences, selon les besoins des différentes plates-formes. Cependant, en utilisant seulement une plate-forme, les scénarios qui peuvent être évaluées sont plus limités et fournissent un seul point de vue sur le problème à l'étude. Par exemple, utiliser uniquement un banc d'essai Ethernet pour étudier un protocole de

couche de transport ne permet pas d'évaluer l'impact des conditions de connexion sans fil sur le protocole.

Évaluation multi-plates-formes

L'évaluation multi-plates-formes consiste à utiliser des plates-formes complémentaires, de manière indépendante, pour évaluer un même scénario. Par exemple, en utilisant deux bancs d'essai avec différentes technologies de couche de réseaux physiques, ou en utilisant une simulation et une plate-forme d'émulation pour faire évoluer une idée au stade de prototype logiciel.

Certaines plates-formes d'expérimentation, comme JiST/MobNet [50] et Netbed [7], permettent la réplication des expériences sous conditions simulés, émulés et réels. Des fédérations offrent également un accès uniforme aux différents bancs d'essai, ce qui rend plus facile la réplication d'un même scénario avec des technologies réseaux différentes.

Les expérimentateurs peuvent reproduire leurs expériences en adaptant manuellement les étapes qu'ils doivent utiliser pour exécuter l'expérience sur des différentes plates-formes, mais cela est généralement un processus coûteux. Certains outils, comme BonnMotion [51], fournissent la migration d'une scénario à travers un groupe de plates-formes.

Le principal avantage de l'évaluation multi-plates-formes est qu'elle permet de valider les résultats grâce à la collecte de données complémentaires. Un inconvénient lié à l'utilisation de plate-formes multiples est que cela nécessite souvent de plus de temps et d'effort que l'utilisation d'une seule.

Évaluation cross-plates-formes

L'évaluation cross-plates-formes consiste en l'intégration des plate-formes pour leur utilisation dans une même expérience. Il nécessite de communiquer les plates-formes pour échanger du trafic ou des données pendant l'exécution de l'expérience. Des exemples de évaluation cross-plates-formes comprennent interconnexion des plates-fromes simulés, émules et des bancs d'essai afin de mélanger des composantes réelles et des composantes synthétiques dans une même expérience. L'interconnexion des bancs d'essai pour exécuter des expériences à grande échelle avec un grand nombre de nœuds est aussi un exemple d'évaluation cross-plates-formes. Les fédérations de banc d'essai sont nativement capables de fournir des expérience cross-plates-fromes en tant qu'elles permettent d'intégrer des bancs d'essai indépendants dans une même expérience.

Des plates-formes comme l'émulateur CORE [21] et l'infrastructure NSE/Emulab [52] fournissent naturellement un moyen de mélanger des composantes simulées, émulées et réelles.

L'avantage principal de l'évaluation cross-plates-formes est que cela permet d'étudier des scénarios qui sont difficiles à construire en utilisant une seule plate-forme. Par exemple, l'évaluation cross-plates-formes permet de tester un modèle pour une couche réseau physique future et utiliser le trafic réel de l'Internet, en connectant un simulateur avec un réseau Internet. Un inconvénient est que la plupart des plates-formes ne

supportent pas par défaut leur interconnexion avec d'autres plates-formes, et relier des plates-formes peut être couteux.

Le coût de l'expérimentation

La communauté de recherche en réseau informatique a accès à une grande variété de plates-formes d'évaluation pour mener des expériences, y compris les simulateurs, les émulateurs, les bancs d'essai et les fédérations de bancs d'essai. Chaque plate-forme peut avoir une manière différente d'être utilisé, a travers des interfaces utilisateurs spécifiques ou des outils et des services faits sur mesure. A titre d'exemple, alors qu'un simulateur de réseau peut être utilisé en exécutant un programme C++ [13, 14], un banc d'essai peut nécessiter l'utilisation des interfaces web pour d'abord allouer des ressources et puis des actionns manuels pour lancer l'expérience [11, 6].

Le temps et l'effort requis pour mener une expérience et pour collecter les données sont liés aux outils et services disponibles aux utilisateurs sur chaque plate-forme. Des services permettent aux utilisateurs de gérer les ressources et d'automatiser les actions a réaliser pendant une expérience, par exemple, lors de la réservation d'un hôte ou l'installation d'un logiciel. Les outils mis en place pour les utilisateurs donnent un moyen d'accéder à ces services. Certains services peuvent être externalisés, par exemple l'installation d'un logiciel ou l'exécution d'une application peuvent souvent être effectués par des outils externes [4, 53]. Par contre d'autres services, comme la réservation de ressources, dépendront des mécanismes internes de chaque plate-forme [10]. Puisque les outils et les services varient considérablement entre plates-formes, le coût de mener une expérience est spécifique à chaque plate-forme. L'utilisation de plusieurs plates-formes dans un même étude augmente la complexité du scénario et en conséquence, le temps et les efforts nécessaires pour faire fonctionner une expérience. Cela est spécialement vrai lorsque il n'existe pas des outils ou des services communs qui peuvent être utilisés sur toutes les plate-formes.

Le coût de mener une expérience peut être décomposé en les facteurs suivants :

Coût apprentissage. Le temps passé à maîtriser une plate-forme, ces interfaces, ces outils et ces services, au niveau minimum requis pour mener l'expérience.

Coût de conception. Le temps passé à écrire des scripts et des programmes nécessaires pour déployer et exécuter une expérience.

Coût de déploiement. Le temps passé à allouer et configurer les ressources d'une expérience, par exemple, création d'une machine virtuelle, réservation d'hôtes, installation de logiciels, etc.

Coût d'exécution. Le temps passé à écrire des scripts et des programmes pour synchroniser des ressources, générer des données et contrôler une expérience.

Coût de collecte de données. Le temps passé à récupérer et archiver des données générées lors d'une expérience.

Une solution générique capable d'automatiser des expériences sur toutes les plates-formes peut aider à réduire les coûts d'expérimentation et à diminuer le temps et les efforts nécessaires pour mener des études sur les réseaux.

La nécessité d'une expérimentation rigoureuse

La connaissance scientifique est basée sur la vérifiabilité des résultats et des démonstrations. Des résultats vérifiables doivent être reproductibles et doivent également permettre de tirer d'autres prédictions de l'hypothèse initiale. Ceci permet à la science de construire sur les découvertes précédentes et de faire progresser l'état des connaissances.

Dans les domaines de recherche comme les réseaux informatiques, qui dépendent fortement des évaluation empiriques, la vérifiabilité nécessite une méthodologie d'expérimentation rigoureuse qui doit tenir compte des éléments suivants :

- *Validation des résultats*
- *Reproductibilité des expériences*
- *Archivage des données*

La validation des résultats consiste à vérifier si les résultats d'une expérience sont correctes par rapport aux hypothèses initiales. En particulier, cela nécessite la validation du comportement de la plate-forme utilisée. Dans le cas des simulateurs par exemple, la validation des modèle est nécessaire pour assurent que les expériences simulées donnent des résultats correctes [54]. Des bonnes pratiques en matière de validation des plates-formes comprennent la documentation du comportement attendu, et l'utilisation des techniques de benchmarking pour vérifier ce comportement [5]. Les résultats peuvent être validés à travers la reproduction des expériences par différentes expérimentateurs et par l'utilisation de plates-formes différentes qui puissent donner des données complémentaires.

La reproductibilité des expériences [55] consiste à reproduire toutes les étapes qui ont été réalisées dans l'expérience originelle, afin d'avoir des données supplémentaires pour corroborer ou réfuter les résultats obtenus initialement. La reproductibilité est une propriété de la procédure d'expérimentation. Cela exige qu'il soit possible de recréer la configuration d'une expérience et les étapes suivis. Pour ça, la configuration et la procédure utilisées doivent être détaillés et clairement documentées.

Il y a une différence entre reproductibilité et répétabilité d'une expérience. Reproductibilité est une condition nécessaire mais non suffisante pour la répétabilité. Si une expérience est répétable, les mêmes résultats sont obtenus lors de sa reproduction. La répétabilité est une propriété de la plate-forme, qui doit fournir un environnement d'expérimentation très contrôlable et prévisible. La plupart des simulateurs permettent la répétabilité des expériences, car ils sont des environnements synthétiques hautement contrôlables. En revanche, les bancs d'essai, comme PlanetLab, ne supportent pas la répétabilité en raison de leur nature imprévisible et non-contrôlable [56].

L'archivage des données consiste à stocker les données mesurées lors d'une expérience, ainsi que les configurations utilisée d'une manière ordonnée et bien documentée. Les données archivées permettent aux autres chercheurs d'analyser les résultats d'une étude pour valider les conclusions des expérimentateurs originaux. L'archivage est en partie une propriété de la procédure d'expérimentation, qui doit prendre en compte le stockage de données et de la documentation d'une expérience, et en partie d'une propriété de la plate-forme, qui doit permettre une instrumentation adéquate de l'expérience. La difficulté dans le partage des données issues des expériences, et des instructions détaillées pour reproduire ces expériences, affaiblit la possibilité de vérifier les résultats des expériences et de profiter des données déjà obtenues pour des nouvelles études.

Il y a des efforts actifs dans la communauté de recherche en réseaux pour mener des expérimentations rigoureuses. Les questions relatives à la validité, l'archivage, la reproductibilité et la répétabilité des expériences ont été étudiées pendant des années. Des différents travaux ont abordé ces questions et ont proposé plusieurs outils et méthodes, tels que la validation des modèles [24, 17], le benchmarking des plates-formes [57, 58] et leur étalonnage [59], la surveillance active des plates-formes [60, 61, 62], l'instrumentation des plates-formes [63, 64], la disponibilité des données d'expérimentation détaillées [65], des outils d'orchestration des expériences [55, 66, 67, 68], des descriptions synthétiques des expériences [6, 16], et des outils d'automatisation des expériences pour aider les utilisateurs à reproduire les études [69, 70, 71, 4].

Néanmoins, les outils proposés jusqu'aujourd'hui pour améliorer la rigueur de la recherche empirique sur les réseaux restent liés au plates-formes ou aux technologies spécifiques, au lieu d'être génériques et de pouvoir s'adapter à toutes les plates-fromes.

Automatisation des expériences sur les réseaux : État de l'Art

Plusieurs travaux ont jusqu'ici traité de l'automatisation des expériences réseaux en faisant abstraction de la représentation de l'expérience et en fournissant des services d'orchestration. Des exemples de ces services sont la génération automatique des topologies réseaux, l'allocation des ressources, la surveillance des expériences et la collecte automatique de données. Les outils existants répondent souvent à des besoins spécifiques des communautés d'expérimentateurs, en se concentrant sur certaines plates-formes ou sur certains technologies réseaux. Certains outils et frameworks pour l'expérimentation réseaux se concentrent sur des cibles spécifiques, comme la modélisation d'applications sur des systèmes distribués. Il y a aussi des outils qui permettent d'automatiser des expériences sur une plate-forme spécifique, tandis que d'autres sont en mesure de travailler sur toutes les plate-formes qui utilisent un même logiciel de gestion.

Automatisation des expériences sur les systèmes distribués

Les outils qui abordent l'automatisation des expériences pour les systèmes distribués se concentrent généralement sur la modélisation des applications dans une expérience, et ne permettent pas de spécifier la couche de protocoles ou la couche physique du réseau.

Dans la plupart des cas, ils permettent l'automatisation sur des plates-formes de type banc d'essai seulement.

Plush [72, 4], son descendant Gush [73], et Splay [3, 53] permettent d'automatiser l'évaluation des applications distribuées sur des bancs d'essai tels que PlanetLab, ModelNet et Emulab. Dans les trois cas, un contrôleur d'expérience qui communique avec un processus agent, exécuté sur les nœuds du banc d'essai, est en charge de l'automatisation de l'expérience. Dans le cas de Plush et Gush, le contrôleur est exécuté dans l'ordinateur de l'utilisateur, tandis que dans Splay le contrôleur fonctionne à l'intérieur du banc d'essai et reçoit des instructions de l'utilisateur grâce à un outil d'interface Web ou une ligne de commande. Plush est capable de déployer et démarrer les processus agent dans les nœuds de banc d'essai. Plush ne repose pas sur un backend pré-installé dans le banc d'essai, autre qu'un service SSH dans les hôtes. Splay, au contraire, nécessite que le backend Splay soit déjà installé dans tous les nœuds du banc d'essai.

Plush et Gush utilisent un fichier de spécification XML pour décrire des expériences, tandis Splay utilise une approche programmatique basé sur le langage Lua. Gush et Splay permettent de décrire des ressources émulées et Splay permet également de décrire des topologies émulés simples sous forme de graphes. Plush permet la définition de flux de travail pour les application en utilisant des processus, des barrières et des blocs des applications, alors que Splay permet décrire des flux de travail et de déploiement progressif des expériences en utilisant des jobs et des événements.

ExCovery [74] se concentre sur l'analyse de la fiabilité des processus distribués. ExCovery utilise un schéma XML pour représenter des expériences, décrivant les processus à exécuter, les facteurs d'entrée, les injections de fautes et les opérations de manipulation de l'environnement. La description de l'expérience en format XML est passée à une processus maître qui l'interprète et envoie des instructions aux nœuds de la plate-forme. Le programme maître communique avec les nœuds à travers des appels XML sur un canal de communication dédié. ExCovery impose plusieurs exigences sur les bancs d'essai, tels que l'accès privilégié complet à tous les nœuds.

Weevil [75] se concentre sur l'évaluation des systèmes hautement distribués composés par des services fournis à travers un grand nombre de points d'accès. Il représente l'expérimentation comme un processus en deux phases : la génération de la charge de travail, et le déploiement de l'expérience et son exécution. La charge de travail est générée par synthèse, grâce à la simulation hors ligne, et la séquence des appels est rejouée au cours de l'expérience exécuté sur un banc d'essai. Le déploiement de l'expérience et son exécution reposent sur une architecture centrale avec un contrôleur, dans laquelle un script maître est responsable de la coordination des composants du système, le déploiement de son exécution et l'exécution de la charge de travail. Weevil génère automatiquement le script maître, et d'autres scripts, à partir d'un fichier de configuration créé par l'utilisateur. Weevil nécessite de l'accès d'utilisateur au banc d'essai à distance, afin d'automatiser le déploiement de l'expérience, l'exécution et la collecte de données.

Expo [76, 77] se concentre sur des applications distribuées sur des architectures de grille et des bancs d'essai distribués. Sa principale caractéristique est la capacité de distribuer efficacement les instructions émises par un expérimentateur sur les ressources hétérogènes distribuées. L'expérimentateur interagit avec les ressources du banc d'es-

sai à travers une interface interactive, qui communique avec une API de banc d'essai et avec les composants du contrôleur de l'expérience. L'API de banc d'essai enveloppe des services fournis par les bancs d'essai et interagit avec un système de réservation qui permet d'allouer des ressources pour des expériences. Le contrôleur de l'expérience envoit des commandes aux ressources et aux processus de journaux, et stocke des informations sur l'expérience. Une abstraction des tâches permet d'associer une commande à une ressource spécifique. Les tâches peuvent être exécutées de manière synchrone ou asynchrone. Dans le cas d' Expo, les expériences sont décrites à l'aide d'un script Ruby.

XFlow [78] est un moteur de flux de travaux qui exécute des expériences comme des flux de contrôle. Des expériences sont modélisés comme des graphes orientés composées d'activités, et sont décrits en utilisant un langage de domaine spécifique. Des activités représentent des tâches à exécuter sur les ressources d'un banc d'essai. Elles peuvent être agrégées à différents niveaux, formant une structure hiérarchique, ou composées pour former des flux de travaux réutilisables dans plusieurs expériences. Les expérimentateurs peuvent utiliser des activités prédéfinis, fournis par les bibliothèques de base de Xflow, ou développer leurs propres activités en utilisant un langage de programmation de bas niveau. Une même expérience de flux de travail peut être adaptée aux différents bancs d'essai en remplaçant des activités spécifiques.

Execo [79] est un outil pour automatiser l'exécution des opérations parallèles distribuées sur des hôtes Unix. Des expériences sont modélisées en termes de processus locaux ou distants, en utilisant un API Python. Les nœuds distants sont accessibles via SSH. Des services de banc d'essai spécifiques, comme la réservation des ressources, peuvent être contrôles par la construction d'une couche au-dessus d'Execo. L'execo_engine est le module responsable de l'exécution de l'expérience. Il fournit un cycle de vie de l'expérience, et peut être étendu afin de définir des flux de travaux plus complexes. Parmi les autres caractéristiques, Execo est capable d'utiliser des paramètres de balayage, de télécharger et de collecter des fichiers et il permet l'agrégation des résultats.

Automatisation des Expériences sur des Plates-formes Spécifiques

Diverses solutions d'automatisation ont été développées pour répondre aux besoins des communautés d'utilisateurs qui travaillent avec les plates-formes spécifiques. Ces outils et frameworks sont généralement conçus pour profiter des caractéristiques de bas niveau des plates-formes cibles. Ils sont capables de modifier la configuration de la topologie du réseau et des protocoles utilisés, au lieu d'être limités à la modélisation du niveau des applications d'une expérience.

WISEBED [80] est un framework de domaine spécifique pour automatiser l'expérimentation sur les bancs d'essai de capteurs sans fil. Il met en œuvre le concept de bancs d'essai virtuelles (VTBS), similaire au concept de sliver in PlanetLab, afin de fournir un accès unifié à des groupes isolés de ressources, y compris potentiellement des ressources simulés, émulés et des éléments réels. Les VTBS sont définis en utilisant le schéma XML WiseML et ils exposent une interface d'expérimentation uniforme aux expérimentateurs à travers un interface de services web (iWSN) installée dans le banc d'essai. Après l'instanciation d'un VTB, les expérimentateurs peuvent modifier en temps réel l'expérience

et exécuter des commandes en invoquant des opérations VTB à travers une interface graphique ou en utilisant un contrôleur de lignes de commande.

SAFE [70] est un environnement d'automatisation des expériences pour le simulateur ns-3. Il permet l'exécution en parallèle de simulations, l'automatisation de la collecte de données, le stockage et l'analyse des données et leur visualisation. Les expérimentateurs exécutent des simulations en lots, en soumettant à l'infrastructure SAFE un programme C++ ns-3 et un fichier de configuration décrivant les paramètres à explorer. SAFE lance alors des réalisations parallèles d'une même simulation avec des paramètres différents, afin de balayer les tranches de paramètres à étudier. L'effort de calcul lié à l'exécution de plusieurs reàlisations d'une simulation ns-3 est partagé parmi un groupe de nœuds commandés par un serveur. Chaque simulation est exécuté jusqu'à ce qu'un détecteur de terminaison l'arrête, puis les données sont automatiquement collectées sur le serveur SAFE pour leur traitement.

Emulab Workbench [65] permet d'automatiser l'exécution des expériences dans des bancs d'essai Emulab. Les expériences sont décrites en utilisant la syntaxe NS, qui divise la définition de l'expérience entre une partie statique et une partie dynamique. La partie statique décrit les hôtes, les périphériques et les liens du réseau, les agents de programme pour exécuter des applications et la configuration de la topologie du réseau. La partie dynamique décrit le comportement pendant l'exécution de l'expérience en utilisant des événements. L'intégration des éléments de simulation avec des hôtes réels est possible à travers l'utilisation du backend de simulation NSE [81, 52]. Des segments dans la définition de l'expérience peuvent être marqués comme simulés en utilisant un marqueur spécial. Lors de leur exécution, les expériences sont contrôlées et surveillées par le logiciel du banc d'essai Emulab en utilisant un système d'événements distribués. Emulab Workbench automatise également l'allocation des ressources et leur configuration, a travers des informations fournies par l'expérimentateur dans la définition de l'expérience.

Automation des Expériences sur des Bancs d'Essai hétérogènes

L'automatisation des expériences sur des bancs d'essai hétérogènes se fait la plupart du temps à travers la fédération des bancs d'essai. Plusieurs initiatives des fédération des banc d'essai ont été développées ces dernières années. Deux des plus ambitieuses sont GENI [9] aux États-Unis et Fed4FIRE [8] en Europe. Les deux fédérations sont composées par des bancs d'essai avec des backends de gestion divers. Elles sont même composées par d'autres sous-fédérations, comme ProtoGENI [82] dans GENI et Panlab [48] dans Fed4FIRE.

Les fédérations hétérogènes fournissent des outils d'expérimentation qui peuvent être utilisés sur des bancs d'essai différents pour automatiser des expériences. Ces outils permettent de décrire des ressources, de découvrir et de réserver ces ressources, d'exécuter des applications et de recueillir des données. Afin de pouvoir fonctionner sur des bancs d'essai différents, ces outils reposent sur des logiciels de backends communs, à savoir, des frameworks de contrôle et de gestion de bancs d'essai, qui doivent être déployés dans tous les bancs d'essai dans la fédération. Ces backends permettent de fournir des

services d'expérimentation communs à travers les bancs d'essai, et une même interface d'expérimentation pour les expérimentateurs. Un exemple d'un de ces backends est la Slice Federation Architecture (SFA) [83], adoptée à la fois par GENI et Fed4FIRE. SFA fournit des services pour découvrir, allouer et réserver des ressources parmi les bancs d'essai de la fédération d'une manière non centralisée.

Tandis que Fed4FIRE suit une approche homogène et impose un même backend à tous les bancs d'essai, GENI accepte une approche hétérogène où des backends différents peut être adoptés par des bancs d'essai, formant des clusters de sous-fédération. Ces clusters correspondent aux backends ProtoGENI, Open Resource Control Architecture (ORCA) [84], cOntrol and Management Framework (OMF) [85], et PlanetLab. Par contre, l'architecture de Fed4FIRE est basée sur l'adoption stricte du même backend par tous les bancs d'essai et sous-fédérations. Ce backend utilise SFA et le Federated Resource Control Protocol (FRCP) [85] pour fournir des services d'expérimentation uniformes à travers la fédération. SFA est utilisé pour allouer des ressources pour les expériences et FRCP est utilisé pour contrôler les ressources au cours de l'expérimentation.

SFA est ajouté à des bancs d'essai Fed4FIRE à travers le SfaWrap [86], afin d'exposer des ressources spécifiques de chaque banc d'essai d'une manière générique. Les ressources ainsi exposées peuvent être explorées et allouées sur le portail MySlice [87], ou tout autre outil capable d'exécuter des requêtes SFA.

FRCP est basé sur le framework OMF pour la gestion des bancs d'essai. Il définit un protocole de messagerie afin de contrôler les ressources du banc d'essai et une infrastructure avec un serveur publish/subscribe capable de livrer des messages aux ressources ciblées à l'intérieur d'un banc d'essai. Un contrôleur d'expérience, comme le OMF Experiment Controller (EC), peut orchestrer des expériences en traduisant une description de l'expérience définie par l'utilisateur en messages FRCP, et les envoyer au serveur publish/subscribe du banc d'essai. Lors de la réception des messages, le serveur les transmet au Resource Controller (RC) qui gère la ressource cible dans le banc d'essai. L'OMF EC permet de décrire des expériences en terme de nœuds, de liens de réseaux et d'applications, et utilise des événements pour définir le comportement des ressources pendant l'éxecution de l'expérience.

Bien que non standardisée dans Fed4FIRE, PanLab offre une alternative au backend SFA/FRCP. Les services de gestion Panlab sont couplés avec l'outil Teagle pour automatiser l'exécution des expériences. Teagle permet la création d'un environnement de banc d'essai virtuel, grâce à l'outil de VCT, ainsi que la génération des flux de travail exécutables à travers son moteur d'orchestration. Par ailleurs, Teagle permet la création de scripts de simulation ns-3 à partir d'une interface utilisateur graphique [88], permettant d'exécuter un même scénario à la fois en mode banc d'essai et en mode simulation.

GENI ne dispose pas des outils standardisés pour automatiser l'exécution des expériences ou la collecte de données. De plus, chaque cluster de la fédération repose sur des outils d'orchestration spécifiques tels que Gush et OMF EC. Le projet PrimoGENI [89], dérivé du logiciel de gestion Emulab, se distingue en fournissant un environnement de développement intégré, appelé slingshot, pour gérer le cycle de vie complèt des expériences. De plus, PrimoGENI offre des supports pour la simulation et l'émulation des expériences. Les expérimentateurs peuvent programmer des commandes sur slingshot

pour leur exécution sur des hôtes émulés.

Comparaison de l'expérience Automation Outils

Outil	Techonlogie	Plate-forme	Backend	Flux	Interactivité
Plush/Gush [72, 4, 73]	S/D	BDE	SSH	✓	✓
Splay [3, 53]	S/D	BDE	Splay	✓	✓
ExCovery [74]	S/D	BDE	ExCovery	✓	
Weevil [75]	S/D	BDE	Tous	✓	
Expo [76, 77]	S/D	BDE	Tous	✓	✓
XPFlow [78]	S/D	BDE/Ému.	Tous	✓	
Execo [79]	S/D	BDE	SSH	✓	✓
Wisebed [80]	Sensors	BDE	Wisebed		✓
SAFE [70]	Tous	Simu.	ns-3		
Workbench [65]	Tous	BDE/Simu.	Emulab/NSE	✓	✓
OMF EC [85, 6]	Tous	BDE	FRCP	✓	✓
PrimoGENI [89]	Tous	BDE	ProtoGENI	✓	✓
Teagle [88]	Tous	BDE/Simu.	Teagle/ns-3	✓	✓
NEPI 2.0 [90, 91, 92]	Tous	Tous	Tous		

TABLE D.1 – Comparaison des outils pour automatiser des expériences réseaux, basée sur la technologie qu'ils ciblent (Technologie), le type de plate-forme qu'ils peuvent contrôler (Plate-forme), les type de backend requis (Backend) et leur capacité à soutenir des flux de travaux (Flux) et l'exécution interactive de l'expérience (Interactivité). Dans les valeurs, 'S/D' signifie systèmes distribués, 'BDE' banc d'essai et 'Tous' indique qu'il n'y a pas de limitations.

Le tableau D.1 compare des outils d'automatisation pour des expériences réseaux. Cette comparaison prends en compte la technologie réseau ciblée par l'outil (Technologie), le type de plate-forme qu'ils peuvent contrôler (Plate-forme), les type de backend requis (Backend) et leur capacité à soutenir des flux de travaux (Flux) et l'exécution interactive de l'expérience (Interactivité). Une comparaison plus détaillée de certains outils de gestion des expériences sur des réseaux est disponible dans l'article par Buchert et al [93].

La comparaison dans le tableau D.1 se concentre sur la généralité et l'adaptabilité des outils et des frameworks existants, et leur capacité de permettre l'expérimentation sur des types de plates-formes et des domaines divers. La liste des outils et des frameworks existants est vaste et cette comparaison n'est pas exhaustive. Son objectif est de donner un aperçu général des différentes approches existantes.

En général, les outils existants sont soit spécialisés dans un certain domaine ou une type de plate-forme, soit ils dépendent d'un backend qui doit être pré-installé dans la plate-forme cible.

Certaines outils sont nés des problématiques spécifiques ou dans des communautés spécifiques, par exemple, les systèmes distribués, et leur objectif n'a jamais été d'être

utiles à tous les domaines de recherche en réseaux ou être compatibles avec toutes les plates-formes. D'autres approches tentent de fournir des solutions complètes qui permettent non seulement l'automatisation des expériences, mais qui prennent également en charge la gestion des plate-forme et le contrôle des ressources. Ces approches sont basées sur l'utilisation d'un backend qui fournit des services, c'est le cas des fédérations de banc d'essai.

L'utilisation d'un backend simplifie la fédération des bancs d'essai et facilite l'automatisation des expériences de manière uniforme, car le backend fournit les mêmes services et les mêmes interfaces pour toutes les plates-formes. Néanmoins, l'adoption d'un backend peut être impossible pour certaines plates-formes, lorsque ses ressources sont trop légères pour tourner des services liés aux backend, par exemple certains bancs d'essai de capteurs légers. Deplus, l'utilisation d'un backend pourrait empêcher l'intégration des ressources indépendants dans une expérience, comme des ordinateurs portables ou des serveurs externes tant que les services du backend ne sont pas installés.

Le travail présenté dans cette thèse formalise et étend des travaux antérieurs sur une première version de l'interface de programmation de l'expérience du réseau (NEPI) [90, 91, 92], NEPI version 2.0. NEPI a été initialement conçu pour être un outil générique pour l'automatisation des expériences réseaux. Son approche consiste à découpler l'interface d'automatisation de l'expérience de l'interface de la plate-forme. Ceci permet de l'adapter aux différents plates-formes tout en offrant une même interface d'automatisation des expériences à l'utilisateur. Néanmoins, le modèle d'expérimentation originalement proposée par NEPI n'était pas assez flexible, et il ne permettait pas la définition des flux de travaux, ou la possibilité de modifier les expériences pendant leur exécution. De nombreuses plates-fromes permettent l'expérimentation interactive, surtout les émulateurs et les bancs d'essai particulières. NEPI 2.0 permettait seulement la modélisation d'un cycle de vie statique des expériences, ce qui limitait les possibilités d'expérimentation sur plusieurs plates-formes.

Cette thèse généralise et simplifie le modèle et l'interface de programmation initialement proposés par NEPI, en offrant un cycle de vie plus général qui s'adapte aussibien aux bancs d'essai qu'aux émulateurs et simulateurs. Ce travail intègre un moteur d'orchestration capable de résoudre l'orchestration des ressources arbitraires dans une expérience, la définition des flux de travail et l'expérimentation interactive. également, ce travail vise à faciliter l'expérimentation rigoureuse, comprenant la reproductibilité des expériences, la réplication, la documentation, et l'archivage de données.

Hypothèse de recherche

La question qui motive ce travail est de savoir si il est possible de définir une solution générique pour automatiser des expériences réseaux sur des plates-formes d'expérimentation arbitraires et pour des scénarios de expérimentation de réseau arbitraires.

L'automatisation signifie minimiser ou éliminer l'intervention humaine dans le cycle de vie de l'expérience réseau, en prévoyant des mécanismes pour effectuer les étapes de l'expérimentation à la place de l'utilisateur. Ces mécanismes d'automatisation doivent

permettre reproduire la même procédure d'expérimentation à travers des exécutions successives d'une même expérience.

La compatibilité avec des plates-formes arbitraires signifie que l'approche choisit pour automatiser l'expérimentation doit être potentiellement adaptable à toutes plates-formes d'expérimentation, existants ou futures.

La compatibilité avec des scénarios arbitraires signifie que la solution proposé ne peut être limitée à un sous-ensemble de scénarios ou'de domaines du réseau, et qu'elle doit être capable de capturer potentiellement tout scénario de recherche. En particulier, il devrait être possible de combiner des plates-formes pour créer des expériences multi-plates-formes et cross-plates-formes.

Suite à ces considérations, l'hypothèse de recherche formulée dans cet thèse est la suivante :

Il est possible de concevoir et de mettre en œuvre une technique pour automatiser le cycle des vie d'expériences réseaux sur toutes plates-formes de évaluation, ou leur combinaison, afin de d'évaluer toutes expériences sans limitation du scénario.

Cette hypothèse est motivée par l'existence de solutions d'automatisation d'expérimentation qui résolvent partiellement le problème de l'automatisation des expériences réseaux sur des plates-formes différentes. Ainsi que par la possibilité de trouver une généralisation au cycle de vie des expériences réseaux qui permet l'automatisation pour des plates-formes arbitraires et des scénarios arbitraires.

Approche

L'approche proposée dans cette thèse pour automatiser l'expérimentation de réseau sur des plates-formes arbitraires ne tente pas de remplacer les solutions existantes, mais de les intégrer et de les compléter. Cette approche essaie aussi de favoriser la réutilisation des solutions existantes et la coopération entre plusieurs outils et plates-formes d'expérimentation. Cette approche est basée sur les principes suivants :

Expérience du cycle de vie, interface et modèle genériques L'automatisation des expériences est généralisable à des plates-formes et des ressources arbitraires a travers l'utilisation d'une interface générique et d'une modèle d'expérimentation générique. L'interface d'expérimentation générique est basée sur les opérations provenant des étapes du cycle de vie générique. Ces étapes doivent être valables pour des plates-formes arbitraires. Le modèle d'expérience générique est basée sur la représentation des scénarios arbitraires de façon générique en utilisant trois entités : l'expérience, les ressources, et les données.

Adaptation grâce aux composants informatiques externes. Permettre à un même outil d'automatisation de s'adapter à toutes les plates-formes, sans imposer des changements dans les plates-formes elles-mêmes. Cela est réalisé à travers le découplage entre l'interface spécifique d'expérimentation fourni par la plate-forme et celle générique exposée à l'utilisateur. Des plates-formes individuelles sont adaptés à l'interface générique grâce aux composantes informatiques adaptateurs externes, de la même manière qu'un

dispositif de réseau est adapté à l'interface sockets en utilisant un pilote informatique spécifique à chaque périphérique réseau.

Flux de travaux basée sur le état des ressources Une machine à états finis est utilisée pour modéliser le comportement des ressources arbitraires pendant l'exécution des expériences réseaux. Le comportement de base définit par cette machine à états finis peut être modifiée par les expérimentateurs en ajoutant des contraintes supplémentaires sur les transitions des états de ressources, en relation avec d'autres ressources. Ces contraintes entre les états des ressources permettent de définir des flux de travaux. D'autres outils d'automatisation des expériences réseaux, tels que Emulab Workbench [65], permettent aussi la définition des flux de travaux pour modéliser le comportement des ressources pendant l'exécution des expériences.

Moteur d'orchestration générique. L'ordre d'exécution des tâches d'expérimentation associées aux ressources arbitraires est résolu par un composant générique : un moteur d'orchestration. Ce moteur utilise un algorithme 'online' pour résoudre l'ordre d'exécution des tâches, et utilise un approche de type 'black box' pour résoudre les contraintes d'exécution, par exemple en prennent en compte les flux de travaux définis par l'utilisateur.

Contributions

Ce travail apporte les contributions suivantes :

1. **La formalisation et la spécification d'une approche générique pour automatiser des expériences réseaux pour des plates-formes et des scénarios arbitraires.** Cet approche vise à permettre l'automatisation des expériences réseaux avec des méthodes d'évaluation différentes, y compris la simulation, l'émulation, et l'expérimentation réel, en utilisant une seule plate-forme ou des plates-formes multiples, pour des scénarios cross-plates-formes ou multi-plates-formes. Les contributions suivantes sont apportées :

   - **Expérience du cycle de vie générique.** L'identification d'un cycle de vie générique pour des expériences réseaux, adapté aux simulateurs, émulateurs, bancs d'essai, et banc d'essai fédérés.
   - **Interface générique d'expérimentation.** La spécification d'un ensemble d'opérations qui englobent toutes les tâches nécessaires pour automatiser l'expérience sur des simulateurs, émulateurs, bancs d'essai et bancs d'essai fédérés, ainsi que la spécification des opérations supplémentaires pour offrir la reproductibilité des expériences, l'archivage de données et leur partage.
   - **Modèle générique d'expérience.** La définition d'un modèle générique pour représenter des scénarios des expériences arbitraires est basés sur trois entités simples : expérience, ressources et des données.
   - **Moteur d'orchestration générique.** La spécification d'un algorithme pour l'orchestration des expériences capable de résoudre des contraintes arbitraires sur l'ordre d'exécution des tâches d'expérimentation liées aux ressources.

2. **Une implémentation logicielle de l'approche générique d'expérimentation réseau sous la forme d'un framework de programmation.** Le framework fourni une description de haut niveau des expériences réseaux arbitraires et permet l'orchestration automatique sur des simulateurs, des émulateurs et des bancs d'essai. Le framework est composé d'une bibliothèque Python open source qui peut être utilisé et étendu librement par les expérimentateurs. Les contributions connexes suivantes sont apportées :

   - **Hiérarchie des classes 'ResourceManager' extensible.** Une abstraction de classe extensible pour modéliser les ressources sur des plates-formes arbitraires.
   - **Une implémentation logicielle du moteur d'orchetration génerique.** Une implémentation logicielle de l'algorithme d'orchestration, capable de résoudre des dépendances arbitraires sur des ressources et de répondre dynamiquement aux changements dans la définition de l'expérience. L'algorithme d'orchestration générique est implémenté dans le composant 'Scheduler' dans l'entité 'ExperimentController', responsable de la coordination des expériences.
   - **Collecte et archivage des resultats automatisé**. Soutien à la collecte et l'archivage de données à travers le composant 'Collector'. Ce composant est tiré de la hiérarchie de classe des ResourceManagers. Cette hiérarchie peut être étendu pour fournir des services associés aux ressources dans l'expérimentation, tels que le traitement et la collecte des données.
   - **Détection des erreurs et des conditions de l'échec**. Politiques d'échec personnalisables, définies par l'expérimentateur, utilisées pour la détection des erreurs dans l'exécution des expériences et leur interruption.
   - **Reproduction automatisée des experiences**. Un mécanisme visant à automatiser la ré-exécution des expériences jusqu'à l'occurrence d'une critère défini par l'utilisateur.
   - **Réplication des expériences.** Soutien pour la génération automatisé des topologies des expériences visant à simplifier la traduction des scénarios entre des plates-formes différentes.
   - **Soutien à la simulation**. Soutien à la simulation en utilisant le simulateur ns-3.
   - **Soutien à l'eémulation**. Soutien à l'émulation de logiciel en utilisant DCE, l'extension d'émulation pour ns-3, et NetNS, un émulateur basé sur la virtualisation Linux avec des namespaces.
   - **Soutien à l'expérimentation réel**. Soutien à l'expérimentation réel sur Linux, en utilisant des bancs d'essai avec accès SSH et authentification par clé privé, sur des bancs d'essai OMF et PlanetLab et sur le bancs d'essai dans la fédération Fed4Fire.

3. **Evaluation du framework d'expérimentation.** L'évaluation de l'approche proposée, effectuée en utilisant le framework développé, sur la base de trois aspects clés : l'extensibilité, l'efficacité et la flexibilité de la solution proposée.

   - **Extensibilité.** Une évaluation de la capacité du framework à être étendu aux plates-formes arbitraires, y compris les simulateurs, les émulateurs et les bancs d'essai, est faite sur la base de l'analyse du développement des plates-formes actuellement supportées par le framework.
   - **Efficacité.** Une évaluation de l'efficacité de l'algorithme d'orchestration, en termes de temps et de mémoire nécessaires pour l'exécution des expériences, est réalisée sur trois plates-formes. Les critères d'évaluation prennent en compte différents facteurs qui pourraient impacter l'efficacité de l'algorithme, tels que le temps requis pour exécuter des tâches et le nombre de threads utilisés pour paralléliser orchestration.
   - **Flexibilité.** Une démonstration de la capacité du framework pour satisfaire différents scénarios d'expérimentation basée sur cinq cas d'utilisation : *maintenance des plates-formes*, *soutien à l'enseignement*, *expérimentation sur des bancs d'essai fédérés*, *expérimentation cross-plates-fromes* et *expérimentation multi-plates-formes*

Plan de thèse

Le reste de cette thèse est constitué des six chapitres suivants :

Chapitre 2 : Automatisation générique des expériences réseaux. Cet chapitre définit un cycle de vie générique des expériences pour des simulateurs, des émulateurs et des bancs d'essai. En outre, il définit une architecture composée d'un interface d'expérimentation générique et d'un modèle d'expérience générique, et propose un algorithme générique pour l'orchestration des expériences arbitraires.

Chapitre 3 : Implémentation logiciel du framework d'automatisation. Ce chapitre décrit l'implémentation logiciel d'un framework basé sur l'architecture définie dans le Chapitre 2. Il détaille l'ensemble des fonctionnalités offertes, en particulier, la hiérarchie de classe ResourceManager pour étendre les fonctionnalités du framework et l'implémentation de l'algorithme d'orchestration.

Chapitre 4 : évaluation de l'extensibilité du framewok. Ce chapitre évalue la faisabilité et le coût d'étendre le framework pour automatiser l'expérimentation sur des simulateurs, des émulateurs et des bancs d'essai.

Chapitre 5 : évaluation de l'efficacité de l'algorithme d'orchestration utilisé par le framework. Ce chapitre évalue la performance de l'algorithme d'orchestration sur le temps et la mémoire requis pour exécuter une expérience en utilisant une plate-forme Dummy, la plate-forme ns-3 et la plate-forme PlanetLab.

Chapitre 6 : évaluation de la flexibilité du framework. Ce chapitre fournit des exemples de cas d'utilisation pour démontrer la capacité du framework pour modéliser une variété de scénarios réseaux différents.

Chapitre 7 : Conclusions. Ce chapitre résume les résultats de ce travail de thèse et discute les avantages et les limitations de l'approche proposée. Il discute aussi des améliorations possibles et des extensions.

Conclusions

Cette thèse présente une approche générique pour l'automatisation des expériences sur des réseaux informatiques, adaptée à toutes les plates-formes et les scénarios d'expérimentation. L'approche proposée repose sur l'abstraction du processus d'expérimentation basé sur un cycle de vie d'expérience composé d'étapes génériques qui peuvent être adaptées aux simulateurs, émulateurs et bancs d'essai réseaux. Ces étapes génériques sont utilisées comme base pour définir une architecture d'automatisation générique formée de trois composantes : un modèle générique pour décrire des expériences réseaux (GNEM), une interface de programmation générique gérer des expériences réseaux (GNEPI) et un moteur d'orchestration générique pour mettre en œuvre des expériences réseaux (GOE). Grâce à ces composantes, l'architecture permet de représenter et d'exécuter des expériences d'une manière uniforme sur des plates-formes diverses. En utilisant une même architecture pour automatiser l'expérimentation sur des plates-formes différentes il est possible d'appliquer une même méthodologie de travail pour parvenir à un processus d'expérimentation rigoureu de manière indépendante des plates-formes. Ceci simplifie l'utilisation de multiples plates-formes pour valider des résultats ou pour modéliser des scénarios complexes.

Pour valider la faisabilité de cette architecture générique pour l'automatisation des expériences, le framework NEPI [90, 91, 92] a été implémenté à nouveau suivant l'approche proposée dans ce travail. NEPI, l'interface de programmation du réseau de l'expérience, est un framework qui prend en charge l'automatisation des expériences d'évaluation de réseau sur des plates-formes hétérogènes, y compris des simulateurs, des émulateurs et des bancs d'essai.

Outil	Techonlogie	Plate-forme	Backend	Flux	Interactivité
NEPI 2.0 [90, 91, 92]	Tous	Tous	Tous		
NEPI 3.0	Tous	Tous+	Tous+	✓	✓

TABLE D.2 – Comparaison entre les versions 2.0 et 3.0 de NEPI. NEPI 3.0 ajoute des flux de travaux et permet la modification interactive des expériences. Cette version améliore aussi le soutien aux des plates-formes arbitraires grâce à un cycle plus flexible.

Le tableau D.2 montre une comparaison entre l'ancien version 2.0 et la nouvelle version 3.0 de NEPI, développée dans le cadre de cette thèse. Ce tableau complète le tableau D.1 dans le chapitre 1. Les deux implémentations de NEPI sont comparées sur la base de leur capacité à permettre la modélisation des expériences avec des technologies

de réseaux diverses (technologie), supporter des types de plates-formes différentes, y compris les simulateurs, émulateurs et bancs d'essai (plate-forme), d'être indépendant d'un framework de gestion de plate-forme pré-installé (Backend), permettre la spécification des flux de travaux (flux) et l'expérimentation interactive (interactive).

Le modèle pour décrire les expériences en NEPI 3.0 est basé sur trois entités abstraits : expérience, ressources et données. Ces entités son définis par un modèle générique pour décrire des expériences réseaux (GNEM). Les ressources sont implémentées comme des classes Python, dérivées d'une hiérarchie de classes enracinée sur la classe abstraite ResourceManager. La classe ResourceManager définit une API pour gérer les opérations sur les ressources, le Resource API, qui reflet les opérations définis par l'interface de programmation générique (GNEPI).

Les expériences sont contrôlées par la classe ExperimentController, qui expose à l'utilisateur une API, le Experiment API, pour programmer des expériences réseaux. Cet API est également défini par le GNEPI. Un composant nommé Scheduler, inclus dans le ExperimentController, fourni une implémentation du moteur d'orchestration générique (GOE). L'algorithme utilisé par le Scheduler est capable de trouver un ordre d'exécution correcte pour les tâches qui correspondent à chaque ressource d'une expérience, prend en compte des contraintes arbitraires entre des ressources et des conditions imposés par les plates-formes, ainsi que le flux de travaux définis par l'expérimentateur.

NEPI peut être ètendu à toutes les ressources dans toutes plates-formes à travers la hiérarchie de classe ResourceManager. Des nouvelles ressources peuvent être contrôlées à travers la création de classes dérivèes de la classe ResourceManager, et l'implémentation des méthodes spécifiques de Resource API. Actuellement, NEPI 3.0 permet de contrôler des expériences sur des bancs d'essai Linux, avec authentification par clé SSH, PlanetLab [11], OMF [85], sur le simulateur ns-3 [13] et sur les émulateurs DCE [15] et NetNS.

L'efficacité de l'algorithme d'orchestration des expériences a été évaluée sur la base du temps et de la mémoire requis pour compléter une expérience. Le scénario choisi pour l'évaluation consiste à échanger des messages ICMP entre des hôtes dans un réseau. Ce scénario a été répliqué sur trois plates-formes, une plate-forme Dummy, le simulateur ns-3 et le banc d'essai PlanetLab. Dans tous les cas NEPI était capable d'orchestrer des expériences avec un grand nombre de ressources, à savoir, des milliers de ressources. NEPI a aussi montré être capable de résoudre l'orchestration des expériences en temps linéaire et avec une utilisation de mémoire linéaire par rapport au nombre des ressources dans l'expérience.

La flexibilité pour modéliser des scénarios réseaux divers et des cas d'utilisation différentes avec NEPI, a été montré à travers des exemples concrets utilisant différentes technologies et plates-formes réseaux. Les cas d'utilisation montrés incluent l'utilisation de NEPI dans l'éducation, la gestion des plates-formes, l'expérimentation sur des bancs d'essai fédérés et l'expérimentation multi-plates-formes et cross-plates-formes.

Avantages et limitations

Plusieurs avantages liés à l'approche d'automatisation choisie ont été présentés grâce à l'évaluation de l'extensibilité, l'efficacité et la flexibilité du framework NEPI. Il a été montré que l'approche proposée peut être utilisée pour automatiser efficacement des expériences sur des simulateurs, des émulateurs et des bancs d'essai ou des combinaisons d'entre eux. Il a aussi été montré que l'algorithme d'orchestration développé dans NEPI est bien capable de gérer des expériences impliquant l'orchestration de miliers de ressources en temps linéaire. Enfin, il a été montré que ce framework peut être utilisé pour modéliser des scénarios divers, impliquant des technologies différentes, telles que les réseaux avec ou sans fil, l'Internet et la distribution de contenu. En autre, le framework peut servir à des usages variés dont l'expérimentation, mais aussi l'éducation et la gestion de plates-formes.

Néanmoins, l'évaluation du framework a également montré certaines limitations liées à son implémentation. En particulier, même si le framework peut être étendu pour contrôler des ressources arbitraires sur toute plate-forme, ceci nécessite du travail car les utilisateurs doivent ajouter les classes ResourceManagers manquantes. Cependant, pas tous les expérimentateurs, les enseignants ou les propriétaires de plates-formes sont prêts à investir du temps pour permettre de contrôler des nouvelles ressources à travers NEPI. La plupart des expérimentateurs peuvent préférer choisir un méthode d'expérimentation plus manuelle avec laquelle ils sont déjà familiers, même si dans le longue terme ça demande plus d'effort. Des expérimentateurs qui sont confrontés à un véritable besoin, qui ne peux pas être comblé avec des outils qu'ils maitrisent déjà, seront plus inclinés à essayer les possibilités ouvertes par NEPI au lieu de continuer à écrire plusieurs scripts et à effectuer des opérations d'expérimentation manuelles.

L'utilisation du framework NEPI présent plusieurs avantages. Par exemple, l'utilisation de multiples plates-formes pour répliquer des expériences et comparer des résultats ou encore les fonctionnalités fournises qui permettent de simplifier un processus d'expérimentation rigoureux. Malgré ces avantages, la croissance d'une communauté d'utilisateurs autour de NEPI reste limitée. Des efforts plus actifs pour faire connaître le framework, en particulier, dans l'université, pourraient avoir un impact positif sur le développement d'une communauté actif d'utilisateurs.

Perspectives

Ce travail offre de nombreuses perspectives d'évolution, y compris des améliorations sur l'implémentation du framework NEPI et sur l'algorithme proposé pour l'orchestration des expériences. Le reste de cette section présente des perspectives possibles de travail future :

Améliorations

Les améliorations considérées dans cette section se concentrent sur l'étude de la performance du framework afin de proposer des optimisations de son implémentation et sur

l'efficacité de l'algorithme d'orchestration.

Optimisation de l'algorithme d'orchestration

Plusieurs chemins peuvent être explorés pour améliorer l'algorithme d'orchestration proposé dans ce travail. L'analyse compétitif de l'algorithme permettra de le comparer à d'autres variantes d'algorithmes de planification en ligne [95] (online scheduling). Une comparaison avec une version boîte blanche de l'algorithme, qui codifiés explicitement les dépendances entre les opérations des ressources, peut aider à mieux quantifier les avantages et les inconvénients de la version boîte noire choisi. Approfondir l'étude des différents paramètres qui ont un impact sur la performance de l'algorithme d'orchestration, tels que le nombre de threads ou le délai de replanification, pourrait aider à améliorer les stratégies de replanification pour réduire la durée de l'expérience. Notamment, une piste à étudier consiste à faire varier dynamiquement le nombre de threads utilisés par le moteur d'orchestration sur la base de l'état instantané du système, au lieu de fixer leur nombre au début de l'expérience.

Optimisation de la durée et la mémoire utilisé

Python est un langage de prototypage rapide. Il fournit un interpréteur interactif et un grande nombre des bibliothèques prêtes à l'emploi pour des tâches diverses. Toutefois, en ce qui concerne la consommation de mémoire et l'exécution multi-thread, Python n'est pas le langage le plus efficace. Comme indiqué dans le chapitre 5, l'efficacité du framework pourrait être améliorée par la implémentation de certains parties centrales du code en langage C. En particulier, l'implémentation du composant Scheduler en langage C permettrait d'éviter la surcharge de performance produit par le Global Interpreter Lock (GIL) utilisé par Python pour la gestion des threads. De même, l'implémentation de la classe de base ResourceManager en langage C réduirait la quantité de mémoire consommée par NEPI, car les types de base en C utilisent moins de mémoire qu'en Python. L'implémentation à neuf de certaines pièces critiques du framework en langage C permettra l'optimisation de la durée et de la mémoire utilisée au cours des expériences, tout en conservant le prototypage rapide des expériences en Python et l'extensibilité du framework.

Extensions

De nombreuses extensions peuvent être faites à partir de ce travail. Cette section présente les extensions qui apparaissent comme des prolongements des aspects déjà considérés dans le framework.

Expérimentation rigoureuse

Le soutien à l'expérimentation rigoureuse est l'une des principales motivations des travaux présentés dans cette thèse. Le framework NEPI intègre des fonctionnalités destinées à faciliter la reproductibilité des expériences, la validation des résultats et l'ar-

chivage des données. La reproductibilité des expériences est permise par la description abstraite des expériences et des mécanismes d'orchestration reproductibles. La validation des résultats est fournise par l'exécution en lot des expériences et la possibilité de comparer des résultats obtenus pour un même scénario en utilisant des plates-formes différentes. L'archivage des données est permis par la collecte automatisée et la sauvegarde des données issues des expériences.

Le soutien à l'expérimentation rigoureuse fournit par le framework peut être encore amélioré à travers l'introduction de critères de comparabilité des expériences et des plates-formes, définis par l'utilisateur. Ceci pourrait permettre de comparer des résultats obtenus sur des plates-formes différentes [5]. Cela aiderait à standardiser des critères pour caractériser des plates-formes et encouragerait les expérimentateurs à effectuer des tests afin de fournir toutes les informations liés à l'environnement utilisé dans leur étude. En outre, un journal de bord pourrait être intégrée au framework, en association à une entité 'Study' qui regrouperait plusieurs expériences liées a un même étude. Un journal de bord numérique aiderait à garder une trace des informations supplémentaires que les expérimentateurs pourraient vouloir stocker ou partager sur une étude.

Traitement et analyse de données

Le traitement des données est l'une des étapes du cycle de vie générique des expériences réseaux défini dans le chapitre 2. Néanmoins, cette étape n'a pas été entièrement développée dans le framework d'automatisation. Alors que le ExperimentRunner permet une intégration partielle du traitement de données et leur analyse dans le flux de reproduction des expériences, c'est l'utilisateur qui doit implémenter les routines de traitement et d'analyse, en utilisant par exemple des bibliothèques Python telles que numpy et matplotlib. Une intégration plus complète des opérations de traitement et d'analyse des données, tels que des fonctions pour générer des graphiques des résultats, pourrait simplifier ces tâches, en particulier pour les utilisateurs non expérimentés qui ne seront peut-être pas familier avec le traitement de données ou les techniques de traçage.

Des informations détaillées sur la structure des données collectées par le framework pourraient être ajoutées par l'expérimentateur à la description des expériences. La description des données pourrait inclure des informations sur le format des données et la manière de les analyser afin d'automatiser leur analyse. Cette description pourrait être utilisée après la collecte des données pour, par exemple, appliquer automatiquement des opérations de filtrage ou de fusion des parties des données recueilles. Des outils existants proposent déjà de tels fonctionnalités de traitement et d'analyse des données [109, 63].

Traduction des scénarios des expériences entre plates-formes

Traduire la description d'une expérience afin de faire correspondre les ressources et leurs comportements est nécessaire pour répliquer des expériences dans des plates-formes différentes. La réplication des expériences est utile pour valider des résultats ou pour faire évoluer un logiciel réseau en utilisant des environnements d'évaluation complémentaires, par exemple des émulateurs et des bancs d'essai. Le framework d'automatisation

proposé dans cette thèse fournit des abstractions de haut niveau pour décrire des expériences de réseaux de manière uniforme. Néanmoins, les ressources disponibles et le niveau de détail permit pour décrire des expériences sont différents entre plates-formes. Cela veux dire que l'expérimentateur doit faire des choix sur l'équivalence des ressources et des configurations entre plates-formes.

Comme mentionné dans le chapitre 3, le problème de la traduction des scénarios entre plates-formes est un problème difficile, car l'équivalence entre ressources peut ne pas être claire ou bien défini. Toutefois, il serait possible de développer un système intelligent capable d'apprendre des décisions de traduction des scénarios prises par les expérimentateurs, afin d'automatiser cette traduction. Un tel système aurait besoin d'un algorithme d'apprentissage supervisé qui puisse prendre des décisions et les communiquer aux expérimentateurs, afin qu'ils puissent les modifier en cas de besoin.

Interface d'utilisateur

Les interfaces graphiques peuvent être utiles pour faire des démonstrations et des tutoriels liés aux technologies réseaux. Pour les utilisateurs inexpérimentés, tels que les étudiants, une interface graphique peut être plus intuitive et facile à utiliser que des scripts. La modélisation d'une expérience comme un graphe de ressources, comme dans le cas de NEPI, permet une représentation graphique simple des expériences. Les ressources peuvent tout simplement être représentées comme des boîtes liés aux autres boîtes, et exposant des listes d'attributs et des traces. NEPI 2.0 avait une interface d'utilisateur graphique ainsi qu'une interface de programmation Python pour modéliser et exécuter des expériences. L'interface graphique permettait aux utilisateurs de glisser et de déposer des boîtes représentant des ressources sur une toile représentânt une expérience. Cependant, cette interface graphique n'a pas été recréée pour NEPI 3.0 car elle limitée la capacité à décrire l'expérience. NEPI 2.0 ne permettait pas aux utilisateurs de définir des flux de travaux. Trouver une manière simple et claire de décrire graphiquement des flux de travaux pour des expériences composées d'un grand nombre de ressources n'est pas trivial. De plus, la description graphique des expériences de grande taille à travers l'ajout et l'interconnexion de centaines de boîtes dans une toile, comme cela était fait par NEPI 2.0, peut être problématique pour l'utilisateur et rendre la description de l'expérience pas claire et difficile à configurer. Une interface graphique pour NEPI 3.0 doit prendre en compte la représentation facile des flux de travaux et des expériences composées de milliers de ressources.

Bibliography

[1] Jacques Wainer, Claudia G Novoa Barsottini, Danilo Lacerda, and Leandro Rodrigues Magalhães de Marco. Empirical evaluation in computer science research published by acm. *Information and Software Technology*, 51(6):1081–1085, 2009.

[2] Walter F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, May 1998.

[3] 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.

[4] 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*, pages 15:1–15:19, Berkeley, CA, USA, 2007. USENIX Association.

[5] Shafqat Ur Rehman. *Benchmarking in wireless networks*. PhD thesis, Ph. D. dissertation, Ecole doctorale Stic, Université de Nice Sophia Antipolis, January 2012.

[6] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. Omf: a control and management framework for networking testbeds. *ROADS*, 43:54–59, January 2009.

[7] 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. *SIGOPS Oper. Syst. Rev.*, 36:255–270, December 2002.

[8] Wim Vandenberghe, Brecht Vermeulen, Piet Demeester, Alexander Willner, Symeon Papavassiliou, Anastasius Gavras, Michael Sioutis, Alina Quereilhac, Yahya Al-Hazmi, Felicia Lobillo, et al. Architecture for the heterogeneous federation of future internet experimentation facilities. In *Future Network and Mobile Summit (FutureNetworkSummit), 2013*, pages 1–11. IEEE, 2013.

[9] 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*, 61:5–23, 2014.

[10] Angelos-Christos Anadiotis, Apostolos Apostolaras, Dimitris Syrivelis, Thanasis Korakis, Leandros Tassiulas, Luis Rodriguez, and Maximilian Ott. A new slicing scheme for efficient use of wireless testbeds. In *Proceedings of the 4th ACM international workshop on Experimental evaluation and characterization*, WINTECH '09, pages 83–84, New York, NY, USA, 2009. ACM.

[11] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[12] Joseph D Touch, Y-S Wang, Venkata Pingali, Lars Eggert, Runfang Zhou, and Gregory G Finn. A global x-bone for network experiments. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2005. Tridentcom 2005. First International Conference on*, pages 194–203. IEEE, 2005.

[13] George F Riley and Thomas R Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.

[14] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, page 60. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008.

[15] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Câmara, 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*, pages 217–228. ACM, 2013.

[16] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

[17] Robert Ricci, Jonathon Duerig, Pramod Sanaga, Daniel Gebhardt, Mike Hibler, Kevin Atkinson, Junxing Zhang, Sneha Kumar Kasera, and Jay Lepreau. The flexlab approach to realistic evaluation of networked systems. In *NSDI*. USENIX, 2007.

[18] Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Yvon Jégou, Pascale Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine

Melab, et al. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*, pages 99–106. IEEE Computer Society, 2005.

[19] Antônio Abelém, Iara Machado, José Augusto Suruagy Monteiro, LCS Magalhaes, M Stanton, and TCM Carvalho. Advances in developing a future internet testbed in brazil. 2011.

[20] Luis Sanchez, Jose A Galache, Verónica Gutiérrez, Jose M Hernández, Jesús Bernat, Alex Gluhak, and Tomás García. Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In *Future Network & Mobile Summit (FutureNetw), 2011*, pages 1–8. IEEE, 2011.

[21] *CORE: A real-time network emulator*, 2008.

[22] Alexander Afanasyev, Ilya Moiseenko, and Lixia Zhang. ndnsim: Ndn simulator for ns-3. Technical Report NDN-0005, NDN, October 2012.

[23] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84. IEEE, 2007.

[24] Svilen Ivanov, André Herms, and Georg Lukas. Experimental validation of the ns-2 wireless model using simulation, emulation, and real network. In *Communication in Distributed Systems (KiVS), 2007 ITG-GI Conference*, pages 1–12. VDE, 2007.

[25] James PG Sterbenz, Egemen K Cetinkaya, Mahmood A Hameed, Abdul Jabbar, Shi Qian, and Justin P Rohrer. Evaluation of network resilience, survivability, and disruption tolerance: analysis, topology generation, simulation, and experimentation. *Telecommunication systems*, 52(2):705–736, 2013.

[26] Torsten Braun, Geoff Coulson, Thomas Staub, Horst Hellbrück, Norbert Luttenberger, and Volker Turau. Towards virtual mobility support in a federated testbed for wireless sensor networks. 2011.

[27] Paxson V. Floyd, S. Difficulties in simulating the internet. In *IEEE/ACM Transactions on Networking (TON)*, volume 9, 2001.

[28] Jungkeun Yoon, Mingyan Liu, and Brian Noble. Random waypoint considered harmful. In *INFOCOM*, 2003.

[29] The network simulator ns-2. `http://www.isi.edu/nsnam/ns`.

[30] Gilbert Chen, Joel Branch, Michael Pflug, Lijuan Zhu, and Boleslaw Szymanski. Sense: a wireless sensor network simulator. In *Advances in pervasive computing and networking*, pages 249–267. Springer, 2005.

[31] Michael A Kaplan, Ta Chen, Mariusz A Fecko, Provin Gurung, Ibrahim Hokelek, Sunil Samtani, Larry Wong, Mitesh Patel, Aristides Staikos, and Ben Greear. Realistic wireless emulation for performance evaluation of tactical manet protocols. In *Military Communications Conference, 2009. MILCOM 2009. IEEE*, pages 1–7. IEEE, 2009.

[32] Jingyao Zhang, Yi Tang, Sachin Hirve, Srikrishna Iyer, Patrick Schaumont, and Yaling Yang. A software-hardware emulator for sensor networks. In *Sensor, Mesh and Ad Hoc Communications and Networks (SECON), 2011 8th Annual IEEE Communications Society Conference on*, pages 440–448. IEEE, 2011.

[33] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36:271–284, December 2002.

[34] 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.

[35] w-ilab.t. `http://www.wilab2.ilabt.iminds.be/`.

[36] Nitlab. `http://nitlab.inf.uth.gr/NITlab/`.

[37] Norbit. `http://mytestbed.net/projects/omf/wiki/OMFatNICTA`.

[38] P. Neyron G. Huard Y. Georgiou, O. Richard and C. Martin. A batch scheduler with high level components. 2005.

[39] Clément Burin Des Rosiers, Guillaume Chelius, Eric Fleury, Antoine Fraboulet, Antoine Gallais, Nathalie Mitton, and Thomas Noël. Senslab very large scale open wireless sensor network testbed. In *Proc. 7th International ICST Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCOM)*, Shanghai, Chine, April 2011.

[40] Larry Peterson, Andy Bavier, Marc E Fiuczynski, and Steve Muir. Experiences building planetlab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 351–366. USENIX Association, 2006.

[41] Kirk Webb, Mike Hibler, Robert Ricci, Austin Clements, and Jay Lepreau. Implementing the emulab-planetlab portal: Experience and lessons learned. In *WORLDS*, 2004.

[42] Anastasius Gavras, Arto Karila, Serge Fdida, Martin May, and Martin Potts. Future internet research and experimentation: The fire initiative. *SIGCOMM Comput. Commun. Rev.*, 37(3):89–92, July 2007.

[43] Marc Suñé, Leonardo Bergesio, Hagen Woesner, Tom Rothe, Andreas Köpsel, Didier Colle, Bart Puype, Dimitra Simeonidou, Reza Nejabati, Mayur Channegowda, et al. Design and implementation of the ofelia fp7 facility: the european openflow testbed. *Computer Networks*, 61:132–150, 2014.

[44] Alastair C Hume, Yahya Al-Hazmi, Bartosz Belter, Konrad Campowsky, Luis M Carril, Gino Carrozzo, Vegard Engen, David García-Pérez, Jordi Jofre Ponsatí, Roland Kűbert, et al. Bonfire: A multi-cloud test facility for internet of services experimentation. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 81–96. Springer, 2012.

[45] Piet Demeester. Ip crew: Cognitive radio experimentation world. In *Future Internet Assembly (FIA) workshop*, 2010.

[46] H Hellbruck, Max Pagel, A Kroller, Daniel Bimschas, Dennis Pfisterer, and Stefan Fischer. Using and operating wireless sensor network testbeds with wisebed. In *Ad Hoc Networking Workshop (Med-Hoc-Net), 2011 The 10th IFIP Annual Mediterranean*, pages 171–178. IEEE, 2011.

[47] Marcelo Yannuzzi, Muhammad Shuaib Siddiqui, Annika Sällström, Brian Pickering, René Serral-Gracià, Anny Martínez, W Chen, S Taylor, Farid Benbadis, Jeremie Leguay, et al. Tefis: A single access point for conducting multifaceted experiments on heterogeneous test facilities. *Computer Networks*, 63:147–172, 2014.

[48] Sebastian Wahle, Christos Tranoris, Spyros Denazis, Anastasius Gavras, Konstantinos Koutsopoulos, Thomas Magedanz, and Spyros Tompros. Emerging testing trends and the panlab enabling infrastructure. *Communications Magazine, IEEE*, 49(3):167–175, 2011.

[49] Openlab. `http://www.ict-openlab.eu/project-info.html`.

[50] Tronje Krop, Michael Bredel, Matthias Hollick, and Ralf Steinmetz. Jist/mobnet: combined simulation, emulation, and real-world testbed for ad hoc networks. In *Proceedings of the second ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 27–34. ACM, 2007.

[51] Nils Aschenbruck, Raphael Ernst, Elmar G. Padilla, and Matthias Schwamborn. Bonnmotion: a mobility scenario generation and analysis tool. In *Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, SIMUTools '10, ICST, Brussels, Belgium, Belgium, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[52] Shashi Guruprasad, Robert Ricci, and Jay Lepreau. Integrated network experimentation using simulation and emulation. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2005. Tridentcom 2005. First International Conference on*, pages 204–212. IEEE, 2005.

[53] Lucas Charles Pascal Felber Raluca Halalai, Etienne Riviere, and Valerio Schiavoni José Valerio. An overview of new features in the splay framework for simple distributed systems evaluation. 2012.

[54] J. Heidemann, K. Mills, and S. Kumar. Expanding confidence in network simulations. *Network, IEEE*, 15(5):58–63, 2001.

[55] Eric Eide. Toward replayable research in networking and systems. In *Proc. of the NSF Workshop on Archiving Experiments to Raise Scientific Standards (Archive'10)*. Citeseer, 2010.

[56] Jeannie Albrecht. Achieving experiment repeatability on planetlab. *Archive '10 workshop*, 2010.

[57] Stratos Keranidis, Wei Liu, Michael Mehari, Pieter Becue, Stefan Bouckaert, Ingrid Moerman, Thanasis Korakis, Iordanis Koutsopoulos, and Leandros Tassiulas. Concrete: A benchmarking framework to control and classify repeatable testbed experiments. In *FIRE Engineering Workshop*, 2012.

[58] Christos Siaterlis, Andres Perez-Garcia, and Béla Genge. On the use of emulab testbeds for scientifically rigorous experiments. *IEEE Communications surveys and tutorials*, 15(2):929–942, 2013.

[59] Sachin Ganu, Haris Kremo, Richard Howard, and Ivan Seskar. Addressing repeatability in wireless experiments using orbit testbed. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, TRIDENTCOM '05, pages 153–160, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Comon: A mostly-scalable monitoring system for planetlab.

[61] Thomas Bourgeau, Jordan Augé, and Timur Friedman. Tophat: supporting experiments through measurement infrastructure federation. In *Proceedings of TridentCom'2010*, Tridentcom, Berlin, Germany, 18-20 May 2010.

[62] Constantine Dovrolis, Krishna Gummadi, Aleksandar Kuzmanovic, and Sascha D Meinrath. Measurement lab: Overview and an invitation to the research community. *ACM SIGCOMM Computer Communication Review*, 40(3):53–56, 2010.

[63] L. Felipe Perrone, Thomas R. Henderson, Felizardo, Vinícius D., and Mitchell J. Watrous. The design of an output data collection framework for ns-3. In R. Pasupathy, S.-H. Kim, A. Tolk, R. Hill, and M. E. Kuhl, editors, *Proceedings of the 2013 Winter Simulation Conference (WSC '13)*. Winter Simulation Conference, 2013.

[64] Jolyon White, Guillaume Jourjon, Thierry Rakatoarivelo, and Maximilian Ott. Measurement architectures for network experiments with disconnected mobile nodes. In *Testbeds and Research Infrastructures. Development of Networks and Communities*, pages 315–330. Springer, 2011.

[65] Eric Eide, Leigh Stoller, and Jay Lepreau. An experimentation workbench for replayable networking research. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 16–16, Berkeley, CA, USA, 2007. USENIX Association.

[66] Maximilian Ott, Ivan Seskar, Robert Siraccusa, and Manpreet Singh. Orbit testbed software architecture: Supporting experiments as a service. In *Proceedings of the First International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities*, TRIDENTCOM '05, pages 136–145, Washington, DC, USA, 2005. IEEE Computer Society.

[67] Ilia Baldine, Yufeng Xin, Anirban Mandal, Chris Heermann Renci, Jeffrey Chase, Varun Marupadi, Aydan Yumerefendi, and David Irwin. Networked cloud orchestration: A geni perspective. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 573–578. IEEE, 2010.

[68] Sebastian Wahle. *A Generic Framework for Heterogeneous Resource Federation*. PhD thesis, Universität Wien, November 2011.

[69] Thierry Rakotoarivelo, Guillaume Jourjon, Olivier Mehani, Maximilian Ott, and Mike Zink. Repeatable experiments with labwiki. *arXiv preprint arXiv:1410.1681*, 2014.

[70] L. Felipe Perrone, Christopher S. Main, and Bryan C. Ward. Safe: Simulation automation framework for experiments. In C. Laroque, J. Himmelspach, R. Pasupathy, and O. Roseand A. M. Uhrmacher, editors, *Proceedings of the 2012 Winter Simulation Conference (WSC '12)*. Winter Simulation Conference, 2012.

[71] Deepal Jayasinghe, Galen Swint, Simon Malkowski, Jack Li, Qingyang Wang, Junhee Park, and Calton Pu. Expertus: A generator approach to automate performance testing in iaas clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 115–122. IEEE, 2012.

[72] Jeannie Albrecht, Christopher Tuttle, Alex C Snoeren, and Amin Vahdat. Planetlab application management using plush. *ACM SIGOPS Operating Systems Review*, 40(1):33–40, 2006.

[73] Jeannie Albrecht and Danny Yuxing Huang. Managing distributed applications using gush. In *Proceedings of the Sixth International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, Testbed Practices Session (TridentCom)*, May 2010.

[74] Andreas Dittrich, Stefan Wanja, and Miroslaw Malek. Excovery – a framework for distributed system experiments and a case study of service discovery. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW '14, pages 1314–1323, Washington, DC, USA, 2014. IEEE Computer Society.

[75] Yanyan Wang, Matthew J Rutherford, Antonio Carzaniga, and Alexander L Wolf. Weevil: A tool to automate experimentation with distributed systems. Technical report, Technical Report CU-CS-980-04, Department of Computer Science, University of Colorado, 2004.

[76] 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 '07, pages 22:1–22:8, ICST, Brussels, Belgium, Belgium, 2007. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[77] Cristian Camilo Ruiz Sanabria, Olivier Richard, Brice Videau, and Iegorov Oleg. Managing large scale experiments in distributed testbeds. Research Report RR-8106, October 2012.

[78] Tomasz Buchert, Lucas Nussbaum, Jens Gustedt, et al. A workflow-inspired, modular and robust approach to experiments in distributed systems. 2013.

[79] 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, United Kingdom, December 2013.

[80] Ioannis Chatzigiannakis Christos Koninis Stefan Fischer Dennis Pfisterer Daniel Bimschas Torsten Braun Philipp Hurni Markus Anwander Gerald Wagenknecht Sándor P. Fekete Alexander Kröller By Geoff Coulson, Barry Porter and Tobias Baumgartner. Flexible experimentation in wireless sensor networks. *Communications ACM*, 55:82–90, 2012.

[81] K. Fall. Network emulation in the vint/ns simulator. 1999.

[82] Robert Ricci, Jonathon Duerig, Leigh Stoller, Gary Wong, Srikanth Chikkulapelly, and Woojin Seok. Designing a federated testbed as a distributed system. In *Testbeds and Research Infrastructure. Development of Networks and Communities*, pages 321–337. Springer, 2012.

[83] Jay Leperau Robert Ricci John Wroclawski Ted Fabber Stephen Schwab Scott Baker Larry Peterson, Soner Sevinc. Slice-based facility architecture. Technical report, 2009.

[84] Jeff Chase. Orca control framework architecture and internals. Technical report, Technical report, Duke University, 2009.

[85] Thierry Rakotoarivelo, Guillaume Jourjon, and Max Ott. Designing and orchestrating reproducible experiments on federated networking testbeds. *Computer Networks*, 63:173–187, 2014.

[86] Thierry Parmentelat, Jordan Auge, Loïc Baron, Mohamed Amine Larabi, Nikos Mouratidis, Harris Niavis, Mohammed-Yasin Rahman, Thierry Rakotoarivelo, Florian Schreiner, Donatos Stavropoulos, et al. Control plane extension-status of the sfa deployment. 2013.

[87] Jordan Augé, Thierry Parmentelat, Nicolas Turro, Sandrine Avakian, Loïc Baron, Mohamed Amine Larabi, Mohammed Yasin Rahman, Timur Friedman, and Serge Fdida. Tools to foster a global federation of testbeds. *Computer Networks*, 63:205–220, 2014.

[88] Marcial Fernandez, Sebastian Wahle, and Thomas Magedanz. A new approach to ngn evaluation integrating simulation and testbed methodology. In *ICN 2012, The Eleventh International Conference on Networks*, pages 22–27, 2012.

[89] Nathanael Van Vorst, Miguel Erazo, and Jason Liu. Primogeni: Integrating real-time network simulation and emulation in geni. In *Proceedings of the 2011 IEEE Workshop on Principles of Advanced and Distributed Simulation*, PADS '11, pages 1–9, Washington, DC, USA, 2011. IEEE Computer Society.

[90] Mathieu Lacage, Martin Ferrari, Mads Hansen, Thierry Turletti, and Walid Dabbous. Nepi: using independent simulators, emulators, and testbeds for easy experimentation. *Operating Systems Review*, 43(4):60–65, 2009.

[91] Mathieu Lacage. *Experimentation tools for networking research*. PhD thesis, Ph. D. dissertation, Ecole doctorale Stic, Université de Nice Sophia Antipolis, November 2010.

[92] Alina Quereilhac, Mathieu Lacage, Claudio Freire, Thierry Turletti, and Walid Dabbous. Nepi: An integration framework for network experimentation. In *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on*, pages 1–5. IEEE, 2011.

[93] 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.

[94] Richard W Conway, William L Maxwell, and Louis W Miller. *Theory of scheduling*. Addison-Wesley, 1967.

[95] Ji r Sgall. On-line scheduling| a survey. In *Dagstuhl Seminar on On-Line Algorithms (Schlo Dagstuhl (Wadern), Germany, June 24 28, 1996), to appear in LNCS. Springer-Verlag, Berlin-Heidelberg-New York*. Citeseer, 1998.

[96] Ronald L Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[97] Ronald L. Graham. Bounds on multiprocessing timing anomalies. *SIAM journal on Applied Mathematics*, 17(2):416–429, 1969.

[98] Ola Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 745–754, New York, NY, USA, 2010. ACM.

[99] Eric Eide, Leigh Stoller, Tim Stack, Juliana Freire, and Jay Lepreau. Integrated scientific workflow management for the emulab network testbed. In *In Proc. USENIX*, pages 363–368, 2006.

[100] 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*, 63:238–250, 2014.

[101] Alina Quereilhac, Damien Saucez, Thierry Turletti, and Walid Dabbous. Automating ns-3 experimentation in multi-host scenarios. 2015. To appear in WNS3 2015.

[102] Netns. `https://github.com/TheTincho/nemu/wiki`.

[103] Linux contrainers project. `https://linuxcontainers.org/`.

[104] Fed4fire. `http://www.fed4fire.eu/`.

[105] G González, R Pérez, J Becedas, MJ Latorre, and F Pedrera. Measurement and modelling of planetlab network impairments for fed4fire's geo-cloud experiment. In *Teletraffic Congress (ITC), 2014 26th International*, pages 1–4. IEEE, 2014.

[106] Alina Quereilhac, Damien Saucez, Priya Mahadevan, Thierry Turletti, Walid Dabbous, et al. Demonstrating a unified icn development and evaluation framework. In *ACM Conference on Information-Centric Networking*, 2014.

[107] Ccnx: Content centric networking project. `http://www.ccnx.org`.

[108] Van Jacobson, Diana K Smetters, James D Thornton, Michael F Plass, Nicholas H Briggs, and Rebecca L Braynard. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies*, pages 1–12. ACM, 2009.

[109] Wolfgang Kiess, Nadine Chmill, Ulrich Wittelsbürger, and Martin Mauve. Modular network trace analysis. In *Proceedings of the 5th ACM symposium on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks*, PE-WASUN '08, pages 1–6, New York, NY, USA, 2008. ACM.

[110] Waf. `http://code.google.com/p/waf/`.

[111] Nitos testbed scheduler. `http://nitlab.inf.uth.gr/NITlab/index.php/scheduler`.