EnosLib: A Library for Experiment-Driven Research in Distributed Computing

Ronan-Alexandre Cherrueau, Marie Delavergne, Alexandre van Kempen, Adrien Lebre, Dimitri Pertin, Javier Rojas Balderrama, Anthony Simonet, Matthieu Simonin

EnosLib: A Library for Experiment-Driven Research in Distributed Computing

Ronan-Alexandre Cherrueau, Marie Delavergne, Alexandre van Kempen, Adrien Lebre, Dimitri Pertin, Javier Rojas Balderrama, Anthony Simonet, Matthieu Simonin

Abstract—Despite the importance of experiment-driven research in the distributed computing community, there has been little progress in helping researchers conduct their experiments. In most cases, they have to achieve tedious and time-consuming development and instrumentation activities to deal with the specifics of testbeds and the system under study. In order to relieve researchers of the burden of those efforts, we have developed EnosLib: a Python library that takes into account best experimentation practices and leverages modern toolkits on automatic deployment and configuration systems. EnosLib helps researchers not only in the process of developing their experimental artifacts, but also in running them over different infrastructures. To demonstrate the relevance of our library, we discuss three experimental engines built on top of EnosLib, and used to conduct empirical studies on complex software stacks between 2016 and 2019 (database systems, communication buses and OpenStack). By introducing EnosLib, our goal is to gather academic and industrial actors of our community around a library that aggregates everyday experiment-driven research operations. A library that has been already adopted by open-source projects and members of the scientific community thanks to its ease of use and extension.

Index Terms—Experiment-driven research, Performance evaluation, Distributed computing experimentation library.

1 Introduction

The spectacular advances in Computer Science have led to the development of more and more sophisticated systems. Software architectures have evolved toward micro-service oriented models where each application is developed by leveraging "off-the-shelf" components such as database backends, communication buses, or distributed coordinators. This approach leads to the development of larger software stacks composed of dozens (and sometimes hundreds) of components [17]. Consequently, it is almost impossible to foresee how a system will behave according to the way it has been deployed and the load it will have to deal with. This is particularly true as most of those software components come as simple containerized services (i.e., their internal behavior is hidden by their API).

To evaluate modern software stacks under representative conditions, computer scientists should now act as biologists and perform experiment-driven research activities. However, conducting experiments on a distributed application under different parameters and scales is a tedious task. It requires to develop specific mechanisms to manage the experimental campaign. Moreover, those mechanisms must be robust enough to enable the reproducibility of results, as largely promoted by the scientific community [1].

While several studies have been dealing with the reproducibility challenge in our community [11], [22], [35], [36], [39], progress on how to help researchers conduct experiments on complex distributed software is marginal. Academic solutions such as Execo [21] have been proposed in the early of 2010's with the goal of executing arbitrary processes on thousands of remote hosts. These systems have been then extended in order to ease the execution of some usual operations (e.g., pushing or collecting a file to, or from, a set of hosts). To a certain extent, they paved the way for DevOps tools such as Puppet [40] and Ansible [19], which greatly facilitate the deployment and configuration of software stacks these days.

Although valuable, DevOps tools do not deal with prior and posterior operations that occur during an experiment. For instance, they do not offer abstractions that enable an *experimenter* to first declare and then acquire the expected testbed resources automatically. This lack of general experiment-driven research mechanisms forces experimenters to deal with specifics of *testbeds* and the *to-be-tested system* for each experiment campaign.

With more than 15 years of experiment-driven research in distributed systems [4], [31], [42], [44], and with all progress that has been achieved on automatic deployment and configuration systems, there is an opportunity for our community to identify common building blocks of experimental engines (also known as experimental artifacts) and to consolidate them into a library. This library should relieve researchers of the burdens of getting and configuring resources accordingly to the experiments, deploying and configuring distributed software, executing benchmarks and collecting traces. Finally, this paper advocates a library to elude the *one solution does not fill all* problem. With a library, experimenters use abstractions they need and can extend it with additional mechanisms when relevant. Our ultimate goal is to assemble a set of common tools for building experimental artifacts, so that efforts made by a

- *R.-A. Cherrueau, M. Delavergne, A. van Kempen, A. Lebre, D. Pertin, and J. Rojas Balderrama were with Inria, IMT Atlantique, LS2N, 44230 Nantes, France. E-mail: firstname.lastname@inria.fr*
- *A. Simonet was with iExec Blockchain Tech, 69123 Lyon, France. E-mail: asb@iex.ec*
- *M. Simonin was with Inria, Univ Rennes, CNRS, IRISA, 35700 Rennes, France. E-mail: matthieu.simonin@inria.fr*

research group can be factorized and re-used by others. We emphasize that functionalities we aim at gathering are broader than the ones offered by DevOps tools such as Ansible [19]: experimental research deals with more than initial deployment and configuration of a software.

With this objective in mind, we have been implementing since 2016, ENOSLIB, a Python library for experiment-driven research in distributed computing. ENOSLIB helps researchers not only in the process of developing an experimental artifact, but also in running this artifact over different infrastructures.

ENOSLIB defines six basics concepts:

- *Providers* enable experimenters to acquire resources from a testbed, dealing with the heterogeneity of their APIs.
- *Resources* abstract machines and networks in order to implement the artifact in a testbed-agnostic manner. Resources are first-class citizen and so can be used to finely supervise, and reconfigure the testbed during the experiment lifetime.
- *Roles and modules* structure the artifact code in order to deal with software deployments and configuration complexities *la* Ansible.
- *Services* provide state-of-the-pratice tools for experiment driven activities, such as a monitoring stack, a network analyzer, and a load generator to evaluate the to-be-tested system.
- *Tasks* finally enable iterative development of the artifact while offering execution guarantees.

All these concepts result from a general analysis of the differents resources and operations experimenters are used to perform. In addition to this analysis, we discuss in this article each concept in detail using a common thread related to the development of a reusable benchmarking artifact for Distributed Relational Database Management Systems (RDBMS). This use case is implemented in the Juice artifact [15]. We then give two additional examples of artifacts built with ENOSLIB. The first one deals with the performance evaluation of communication bus solutions [33]. It illustrates how the experimental steps can be organized to allow the exploration of a large set of parameters in a flexible way. The second one is related to OpenStack, the de facto resource management system. It shows the added value of the resources abstraction to replicate performance evaluations of complex software stacks on different testbeds [8]. We underline that some of these engines have been already used by other people to conduct their own evaluations [26], [34]. Members of the OpenStack community also started to evaluate the use of ENOSLIB to conduct their regression performance tests [32]. [1]

The structure of the article is as follows. Section 2 motivates the interest of a library by presenting experimental protocols from a general perspective, leading to the ENOSLIB concepts introduced in Section 3 and Section 4. Section 5 presents concrete artifacts built with ENOSLIB. Finally, Section 6 presents the related work, Section 7 discusses some of the lessons learned developing the library and Section 8 concludes the paper.

1. For an up-to-date list, please visit https://discovery.gitlabpages.inria.fr/enoslib/theyuseit.html

2 MOTIVATIONS

An experimental study is likely governed by a protocol that defines the execution environment as well as the sequence of the different operations used for the experiment. Implementing and orchestrating these operations can quickly become complicated as it includes numerous error-prone tasks. To illustrate this complexity, we depict in this section a minimalist workflow of an experimental campaign. The description obviously does not cover all possible scenarios (some operations may be optional or achieved in a different order). However, it enables us to point out several opportunities to consolidate common operations into a library.

2.1 Experimental campaigns: A bird's eye view

Before discussing the different steps of an experimental campaign, let us define two concepts that are adopted across this document:

To-be-tested system The subject (e.g., software, protocol) of an experimental campaign whose behavior is studied.

Artifact The software or set of scripts that implements the experimental protocol and allows for studying the to-be-tested system.

In its minimal form, a workflow of an experiment can be modeled as a pipeline of five steps as depicted in Fig. 1.

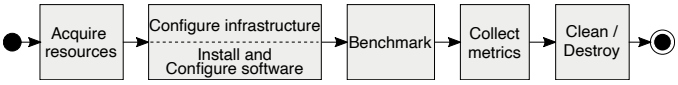


Fig. 1: A minimalist workflow of an experiment.

In the early stage of the artifact execution, interactions with the platform must be performed to acquire an infrastructure for the experiment (e.g., compute, storage, network). It corresponds to the first step of the workflow. Note that the platform can be a simple workstation, a cloud infrastructure or a dedicated academic testbed such as Grid'5000 [4] or Chameleon [23].[2] The second step is composed of two parts. On one hand, the configuration of the infrastructure. This generally requires additional interactions with the resource management system of the testbed. It includes deploying a dedicated environment such as a particular version of Linux or setting up resource constraints to emulate a specific infrastructure. On the other hand, the installation and configuration of the to-be-tested system and any third-party services mandatory for the experimental campaign duration. It covers all operations required to put the system in the expected state before running the experiment itself (installing/configuring the to-be-tested system, its dependencies, third-party applications such as a monitoring stack, and the binaries related to the benchmark). The third step is the execution of the benchmark. It corresponds to the code in charge of evaluating the to-be-tested system. It usually collects application and system metrics (e.g., CPU and RAM utilization of the nodes of the infrastructure). The fourth step corresponds to the collection of relevant data generated by the artifact in order to allow postmortem

2. Unless specified, we used the term testbed in the rest without any distinction between the different kinds of underlying platforms.

analyses. The last step is the release of the testbed resources (i.e., destroy).

As indicated, the above workflow embraces simple experimental protocol needs. In most cases, the artifact is more complex. First, an experiment should be performed multiple times to ensure the statistical significance of the results. Second, an experiment is often executed with different input parameters to explore multiple scenarios. In both cases, the artifact should orchestrate the different iterations of the workflow execution. More precisely, moving to the next iteration may involve cleaning the environment used during the previous iteration and proceeding with the reconfiguration of the system according to the new parameters. The cleaning action is experiment dependent. It ranges from resetting the to-be-tested system to its initial state (deleting processes, files, ...) to redeploying a fresh infrastructure. The reconfiguration action has its own complexity as it may occur at the testbed or to-be-tested system level. The testbed reconfiguration may consist in adding or removing hosts or changing resource constraints in order to emulate new network conditions. The to-be-tested system reconfiguration may change its parameters, version, or more often, the injected load.

When the entire experiment is finished, that is to say, when every combinations of the parameters have been tested, researchers can start the analysis of the data.

2.2 Towards a library for experiment-driven research

Based on the previous section, it is clear that implementing an artifact requires important development efforts to automatize and guarantee that the experiment campaign is executed completely and correctly. Without dedicated tools an experimenter is at risk of failing its evaluation by drawing its attention away from the main goal of the experiment. In this section, we identify opportunities for mutualizing practices and tools into a common library.

The first opportunity is related to the interactions with the testbeds. Most of the code that acquires resources is intended to be reused in each experimental campaign. It thus makes sense to consolidate it into a library. Furthermore, such a consolidation leads to the generalization of the way resources are exposed at the library level. This favors the development of testbed-agnostic artifacts. An independence that enables artifacts to be executed on different infrastructures (and so shared between different research groups).

The second opportunity where a library can help is related to the interactions with the to-be-tested system. There is a need to ease and automate its deployment as well as that of any third-party applications required for the study. For instance, to observe the to-be-tested system, it is mandatory to install applications capable of collecting logs and metrics, or tracing calls in processes. Outside academia, system administrators or DevOps have the expertise in this area. Solutions like Ansible [19] or Docker [5] have been broadly adopted. In comparison, the experimenter, often alone, has to gain these skills for the purpose of the study. This is unfortunately unrealistic due to the complexity of most modern software stacks. It is highly valuable therefore to offer the automatic installation of state-of-the art software stacks for experiment-driven research into a library.

A third opportunity is related to the code in charge of controlling errors that can occur during the execution of the experiment. When conducting experiments campaigns, in particular for distributed systems, failures such as transient connection errors or hardware issues are likely to happen. Transient connections can be treated in a generic manner across different experiments by offering a transparent retry logic. In contrast, hardware issues entail specific fixes depending on the situation, e.g., acquiring and configuring new resources in order to retry the action, raising an error, or silently ignoring the problem. Since failure should be considered as the norm, a library for distributed system experiments should offer remediation mechanisms.

The last opportunity is related to the scaling of the artifact. A monolithic artifact that significantly grows eventually reaches a spot that limits its maintainability. In such a case, separating the code into distinct high-level tasks such as steps of Fig. 1 is essential. Separation of concerns favors programming, testing and debugging in isolation [30]. But isolation is impossible to achieve for an artifact: each task uses side effects from the previous one. For instance, configuring the infrastructure requires resources to be ready first. This drastically slows down and makes the development frustrating at scale. To save the experimenter the burden of executing over and over again the entire workflow after a modification, a library should provide a mechanism to suspend the execution of the artifact on any task; then let the experimenter change this task; and finally resume the execution. The library should also promote idempotent side effects to favor iterative changes. Idempotence is defined in this work as the foundation for repeatable and robust automation [7], [12] —An idempotent task can be executed multiple times always yielding the same result, making it repeatable by design and removing the cumbersomeness of undoing side effects for the experimenter.

This is with all these motivations in mind that we develop the EnosLib library for experiment-driven research.

3 EnosLib key concepts

This section introduces the key concepts of EnosLib,[3] a library to elaborate artifacts that flexibly interact with testbeds and the to-be-tested system. For the sake of clarity, we give, first, an overview of these concepts through a common thread related to the evaluation of distributed Relational Database Management Systems (RDBMSes). We discuss, second, each concept in detail.

All along, code samples illustrate the library usage in a natural fashion for someone familiar with Python.[4]

3.1 Benchmarking distributed RDBMSes with EnosLib

As a common thread, we consider the development of an artifact that compares different distributed RDBMSes in terms of scalability (i.e., the load the distributed RDBMS can satisfy). The scalability depends a lot on the characteristics

3. https://gitlab.inria.fr/discovery/enoslib

4. This paper assumes Python3.7. For conciseness, it stylizes code and thus some examples may not run exactly as presented. Full, runnable examples are available at https://github.com/BeyondTheClouds/enoslib-paper/tree/tpds-2020.

```
1 def experiment(provider, configuration, delay, setup):
2   # Acquire resources on the testbed
3   infra = provider(configuration)
4   hosts, networks = infra.init()
5
6   # Install the distributed RDBMS and third party soft.
7   setup(rdbms=hosts["RDBMS"], sysbench=hosts["client"])
8   monitor(
9     monitored=hosts["RDBMS"], aggregator=hosts["client"][0])
10  # Configure the infrastructure
11  set_latency(hosts["RDBMS"], networks["RDBMS"], delay)
12
13  # Benchmark the distributed RDBMS
14  exec_sysbench(hosts["client"])
15
16  # Collect/backup all logs
17  with ansible_on(hosts) as playbook:
18    playbook.archive(path="/var/log", format="gz")
19    playbook.fetch(src="/var/log.tar.gz")
20
21  # Clean resources
22  infra.destroy()
```

Fig. 2: Implementation with ENOSLIB of an artifact for the evaluation of distributed RDBMSes. Instructions in **bold face** highlight the generic functions provided by ENOSLIB. This code follows the five steps of the workflow depicted in Section 2.

of the underlying infrastructure. For example, experimenting with five and then twenty-five nodes, or with ten and then a hundred miliseconds of round-trip time between nodes. To explore these scenarios, the artifact should be able to acquire and reconfigure resources between the different iterations. It should also collect several metrics such as the CPU, RAM and network consumption of each machine in order to observe the way the clients strain the distributed RDBMS servers. Eventually, this artifact has to be executed on different testbeds in order to ensure the replicability of the results.

Implementing the artifact

We implemented such an artifact for our own investigations [15]. An overview of the code is given by the `experiment` function in Fig. 2. Through this first example, we introduce ENOSLIB key *abstractions* and *functionalities*. Especially, we emphasize how they come together to help the experimenter in easily developing an artifact that explores various characteristics independently of the testbed.

Acquire resources (l. 2-4). The code gets resources on a testbed using an ENOSLIB functionality called `provider`. A `provider` takes a `configuration` as an argument that declares the expected infrastructure: the number of nodes, their characteristics (e.g., amount of RAM, CPU, NICs) and their roles (e.g., RDBMS, client). The `provider` acquires these resources from the testbed by calling the `init` method. It returns the two sets `hosts` and `networks` that describe the allocated resources as a response.

Each element of `hosts` is a python object that represents a node of the testbed. It offers a facility to run arbitrary SSH commands on it. That facility *hides* the underlying technology (e.g., bare-metal, virtual machine, etc.) therefore an experimenter can use it to develop the experiment independently of the testbed. Similarly, each element of `networks` models a network of the testbed.

Roles from the `configuration` label elements of the sets. They are string that indicate the expected behavior of resources. For instance, a host that is tagged with the RDBMS role is supposed to run RDBMS server. In like manner, a network that is tagged with the RDBMS role is supposed to serve traffic of RDBMS servers.

Configure soft./infra. (l. 6-11). The code prepares the infrastructure in order to be ready for the experiment. The `setup` function installs the database servers on `hosts` labeled with the RDBMS role. It also installs the sysbench[5] database benchmarking tool on hosts labeled with the client role, and configures them accordingly to the distributed RDBMS. The `monitor` function deploys a monitoring stack that collects CPU, RAM and network consumption of RDBMS hosts and aggregates results in the first client. Finally, the `set_latency` function enforces a `delay` between RDBMS hosts, and that delay only applies for packets that flow on the RDBMS network.

The `setup` function is specific to the experiment campaign and so its implementation is left to the experimenter. Conversely, `monitor` and `set_latency` functions are provided by default in ENOSLIB due to their common usage.

Benchmark (l. 13-14). The code evaluates the distributed RDBMS by calling the `exec_sysbench` function. Once again, the implementation of such a function is specific to the to-be-tested system and thus is left to the experimenter. In this case, it invokes the sysbench executable on each host labeled with the client role.

Collect metrics (l. 16-19). The code collects experiment logs with another feature of ENOSLIB that takes benefit of Ansible [19]. Thanks to this functionality, an experimenter can interact with the system in an effective manner.

Interacting with the system originally involved shell instructions over SSH. But errors easily slip in with shell (mostly due to the management of side effects) and optimizations are left to the developer. For example, to prevent the twice archiving and fetching of an unchanged large directory (as done in this step), the code would require to maintain and test a checksum of that directory.

Recently, DevOps tools, such as Ansible, have been proposed to facilitate interactions with the system by offering higher level modules that manage side effects and optimizations. Following the received wisdom in software development of not reinventing the wheel, ENOSLIB inherits from more than 2500 modules of Ansible to write a more expressive and robust code.

It is noteworthy that previous functions such as `setup` and `exec_sysbench` can benefit from these Ansible modules. As discussed later, the `monitor` function has been implemented in few lines thanks to it.

Destroy (l. 21-22). The code finally calls `destroy` so that the `provider` releases resources to the testbed.

Invoking the artifact

The distributed RDBMS experimental campaign can be achieved by calling the `experiment` function, multiple times, with proper arguments. For instance, to run the iteration on Chameleon [23] with a cluster of five RDBMS servers

5. https://github.com/akopytov/sysbench

and two sysbench clients, ten milliseconds of round-trip time, and the Galera[6] technology the experimenter executes:

```
experiment(chameleon, 5rdbms2sys-cham.yml, "10ms", setup_galera)
```

Before explaining in detail arguments such as `5rdbms2sys-cham.yml` and `setup_galera`, we want to emphasize the flexibility brought by ENOSLIB by discussing a few more examples of the `experiment` function. These additional calls aim to evaluate RDBMSes scalability over different testbeds and network constraints, solely by changing the arguments of the `experiment` function (in gray in the following to highlight differences).

The first benefits of ENOSLIB are offered by the resources abstraction and provider concretion. The resources abstraction (i.e., `hosts` and `networks`) enables experimenters to write a code that is independent of the concrete resources, while providers do the heavy lifting of acquiring resources on testbeds. In addition to avoiding experimenters to deal with the diversity of testbed APIs, the combination of the provider and resources abstractions enables the execution of the artifacts over different testbeds. This is a significant added value for experiment driven research as it allows researchers to validate the *replicability* of the results. For instance, the same `experiment` function runs on Grid'5000 [4] thanks to the `g5k` provider and the corresponding `5rdbms2sys-g5k.yml` configuration file:

```
experiment( g5k , 5rdbms2sys-g5k.yml , "10ms", setup_galera)
```

Moreover, by being agnostic regarding the concrete resources, it is possible to implement generic functions such as `monitor` and `set_latency` that could be reused across experimental campaigns to save experimenters the bother of writing them. The next execution enforces `"100ms"` of round-trip time between hosts of Grid'5000 whether they are connected through a physical network or a VLAN:

```
experiment(g5k, 5rdbms2sys-g5k.yml, "100ms" , setup_galera)
```

The role abstraction (i.e., label on nodes) hides the amount of hosts with a specific behavior, which produces a code that is independent of the number of resources. For instance, it allows the execution of the same function with a new configuration file `25rdbms10sys-g5k.yml` that defines twenty-five RDBMS servers and ten sysbench clients. The experimenter can thus easily test the *scalability* by increasing the number of sysbench and RDBMS hosts:

```
experiment(g5k, 25rdbms10sys-g5k.yml , "100ms", setup_galera)
```

Finally, roles also abstract the behavior of resources and permit to provide multiple implementations of one concept. For instance, the RDBMS role could be implemented by the aforementioned `setup_galera` function. Or, it could be implemented by a new `setup_CRDB` function that deploys and configures CockroachDB[7]. Being able to call the same function and vary the to-be-tested system lets the experimenter write its campaign code *once and for all*:

```
experiment(g5k, 25rdbms10sys-g5k.yml, "100ms", setup_CRDB )
```

This artifact could be extended with additional distributed RDBMSes by implementing new `setup` functions and become, ultimately, a defacto benchmark suite for RDBMSes.

Through this common thread we give an illustration of how ENOSLIB can help researchers consolidate their efforts in implementing advanced artifacts for each major topic. These new kind of benchmark suites cover the complete life cycle of an experiment campaign, allowing fair comparisons between existing and future proposals.

We describe ENOSLIB concepts in detail in the following.

3.2 Providers: Resources concretion from the testbed

Providers are components in ENOSLIB that are responsible for acquiring resources on existing testbeds. Concretely, a provider is a function that takes a declarative description of the expected resources as argument. It then requests these resources on the corresponding testbed, and finally returns a set of `hosts` and `networks` labeled with their roles. Providers hence help experimenters deal with the variety of the APIs of testbeds. ENOSLIB implements six providers in its current version (v6):

Vagrant provider starts machines and networks on the local machine thanks to Virtualbox or KVM. This provider is mainly used to implement and debug the artifact locally. (220 LoC)

OpenStack provider acquires resources on an OpenStack cloud. A variant for the Chameleon testbed [23] exists in ENOSLIB. (500 LoC)

Grid'5000 provider targets the large scale Grid'5000 infrastructure [4] which supports many features including Virtual LAN to isolate network communications. (1.5 KLoC)

VMonG5k provider deploys a virtualized infrastructure on top of Grid'5000 to segregate the resources and scale the experiment. (350 LoC)

Distem provider deploys a Distem cluster [38] that uses containers to emulate sophisticated topologies. (320 LoC)

FIT IoT-Lab targets wireless sensor devices and heterogeneous communicating objects on the FIT platform [2]. (1 KLoC)

The resources description follows a *declarative* style to make the desire of the experimenter clear about the concrete resources required by the artifact. The description is specific to each provider, but differs only slightly from one another. Roughly, it enumerates how many machines and networks of a specific hardware the experimenter expects, and assigns roles to each of them. For instance, Fig. 3 depicts the resources description for Vagrant (left) and Grid'5000 (right) of the aforementioned experiment (5 RDBMS servers and 2 sysbench clients). The Grid'5000 description is a bit more complex as it supports different network technologies.[8]

The declarative syntax favors sound experimental evaluations [29]. It first enables to conduct evaluations in different configuration to cover a representative sample of the space of the to-be-tested system by easily scaling out resources. For instance, increasing the `count` value of `RDBMS` in the description of Fig. 3 increases the number of resources dedicated to database servers. Similarly, augmenting the amount of `clients` implies a bigger number of sysbench

6. Galera is the active-active replication mechanism to manage a cluster of MariaDB/MySQL — https://galeracluster.com/products/

7. CockroachDB is a geo-distributed RDBMS built on distributed consensus — https://www.cockroachlabs.com/

8. For the sake of clarity, the code presents various configurations in YAML syntax throughout the paper. ENOSLIB also exposes a programmatic interface to build such configuration in Python.

```
$ cat 5rdbms2sys-vagrant.yml
---
env: generic/debian10
machines:
  - roles: [RDBMS]
    count: 5
    flavor: large
  - roles: [client]
    count: 2
    flavor: tiny
networks:
  - db-net
  - xp-net
```

```
$ cat 5rdbms2sys-g5k.yml
---
env: debian-10-x64
machines:
  - roles: [RDBMS]
    count: 5
    networks:[db-net,xp-net]
    cluster: parasilo
  - roles: [client]
    count: 2
    networks: [xp-net]
    cluster: paravance
networks:
  - id: db-net
    type: kavlan
    roles: [RDBMS]
  - id: xp-net
    type: kavlan
    roles: [experiment]
```

Fig. 3: Resources description for Vagrant (left) and Grid'5000 (right). Both descriptions get 5 nodes labeled `RDBMS`, 2 nodes labeled `client`, and put these nodes on `db-net` and `xp-net` networks. The Vagrant provider makes use of virtual machines and configures two private networks. The Grid'5000 provider gets and sets up bare-metal machines. It makes `RDBMS` nodes to have a NIC only for `db-net` and benefit from the VLAN L2 isolation offered by the testbed.

hosts and so puts more load on the RDBMS cluster. Second, the declarative syntax acts as a specification of the hardware setup and OS running on nodes in order to support the reproducibility of experiments.

In addition to help experimenters to deal with APIs of testbeds and favor reproducible evaluations, providers also have idempotent code. Running twice a provider with the same resources description has no extra effects on the infrastructure (no new machine will be started, OS deployed, nor network created). This frees experimenters from the burden of managing the state of testbed resources and potentially speeds up the multiple iterations made with the testbed during the development and execution of artifacts.

Finally, the number of Lines of Code (LoC) for each provider gives a general sense of the relative size to develop a new provider. The number of LoC does not capture libraries or dependencies outside of the repository: the size mostly varies depending on the existence of a SDK to request the testbed (OpenStack has one, Grid'5000 has not) and the features proposed by the testbed.

3.3 Resources: Abstract interactions with the testbed

Resources of a testbed are versatile. They include different *environments* (local machine, academic research testbeds, public and private clouds, etc.) and so, multiple *technologies* (bare-metal machines, virtual machines, containers, flat network, virtual – extensible – LAN, etc.)

To help the experimenter deal with this versatility, EnosLib defines a *resource model*. The resource model abstracts machines and networks to decouple them from the rest of the artifact. Developing an artifact on top of this abstraction hence presents two benefits for the experimenter. First, it helps researchers focus on the domain of the artifact by dissociating the code specifics to the experiment from the testbed [24]. Second, the artifact does not target any particular platform technology, making the artifact executable on any testbed (even those not anticipated by the library [3]).

In EnosLib, a machine resource is *anything* EnosLib *can access through SSH to and run shell commands on*, which is caught by the following `Host` data type:

```
@dataclass
class Host:
  address: str  # IP address of the unit.
  user: str     # Name of the SSH user...
                # ...that can access the unit.
  keyfile: str  # Path to the public SSH key file.
  port: int     # SSH port to connect to the unit.
  extra: Dict   # Network info (NIC, CIDR, ...).
```

From there, an experimenter can run arbitrary code on `Host`s to implement the experiment.

For instance, the code in Fig. 4 gives an excerpt of the `setup_galera` function from Sec. 3.1. It contextualizes a list of machines by simultaneously installing MariaDB and Galera services on all of them using the `run` function — a contextualization that is performed regardless of the underlying technology for a `Host`. In the same fashion, one can easily imagine the `setup_CRDB` function that installs CockroachDB and configures Sysbench accordingly.

The `Host` data type makes functions easy to share and reuse from one experiment to another. The `setup_galera` function could be reused in an artifact that, for example, evaluates a geo-distributed application built on top of a MariaDB Galera cluster. Another example of a reusable unit is the previous `monitor` function in Fig. 2, which installs a monitoring stack and collects metrics.

While we encourage to use the native EnosLib functions to perform remote actions on hosts, developers can export the `Host` data type and use alternatives (e.g., paramiko[9], Execo [21], or even another language thanks to the JSON exporter).

Similarly to machines, EnosLib abstracts networks. In a testbed, a network provides connectivity between machines. It could be of several technologies, for instance, private flat network, bridged network, virtual (extensible) LAN and for Internet Protocol version four (IPv4) and six (IPv6). EnosLib abstracts this in a `Network` object that contains the CIDR, gateway, DNS, and IP address range. On top of that object, the experimenter can develop generic functions. The `set_latency` in Fig. 2 is one example. It implements traffic shaping using NetEm [18] to model a non-trivial network topology. Another brief example is the code shown in Fig. 5. This code uses `Network` information and the Scapy packets

9. http://www.paramiko.org/

```
1 def setup_galera(rdbms: List[Host], sysbenchs: List[Host]):
2   # Install the MariaDB Galera cluster on `rdbms`
3   run("apt install -y mariadb galera", rdbms)
4   # Install Sysbench on `sysbenchs`
5   run("apt install -y sysbench", sysbenchs)
6   # Configure sysbench for the MariaDB Galera cluster
7   ...
```

Fig. 4: Excerpt of the `setup_galera` function from Sec. 3.1. It installs and configures MariaDB Galera cluster and Sysbench on a list of `Host`s: an abstract notion of unit of computation that can be bound to bare-metal machines, virtual machines, or containers. See Fig. 2, line 7 for a call.

```
def analyze_galera(nets: List[Network]):
  net_ipv4 = next(net for net in nets if net.version == 4)
  scapy.sniff(
    filter=f'net {net_ipv4.CIDR} and tcp and port 4567',
    prn=lambda packet: packet.summary())
```

Fig. 5: Analyzing of the Galera protocol on the `net_ipv4` networks. The function builds a specific `filter` with the CIDR of `net_ipv4` and prints TCP `packet`s that are related to the Galera communications.

manipulation library[10] to analyze the protocol of a MariaDB Galera cluster.

It is worth noting this snippet filters networks one step further to only consider the IPv4 network. Actually, ENOSLIB reuses the builtin Python capabilities to manipulate IPv4 and IPv6 addresses and networks.[11]

3.4 Roles: Abstract the number of resources

Abstracting machines and networks lets the experimenter execute arbitrary code independently of the technology. However, executing the same code on all `Host`s or `Network`s does not necessarily make sense. Developing an artifact often requires dividing machines and networks into groups that are contextualized differently. As depicted previously, an artifact that evaluates RDBMSes, for instance, requires to contextualize two kinds of machine. Machines that are the participants of the database cluster and machines that are the clients which stress the cluster.

To put resources in groups, ENOSLIB introduces a notion of role: a label on `Host`s and `Network`s. Thanks to roles, an experimenter can filter out resources in order to define code that will be only executed on some of them. This makes it possible to break down the artifact into *coherent units* responsible for one behavior.

Getting machines and networks of a specific role is as simple as using the Python indexing notation. For instance, the following snippet returns the list of `Host`s and `Network`s labeled with the RDBMS role.

```
rdbms_hosts: List[Host] = hosts["RDBMS"]
rdbms_nets: List[Network] = networks["RDBMS"]
```

Resulting lists could then be passed to functions as for example `setup_galera` and `analyze_galera` from Fig. 4 and 5, such as in the following snippet:

```
setup_galera(rdbms_hosts, hosts["client"])
analyze_galera(rdbms_nets)
```

This shows that roles *decouple* the amount of resources involved. For instance, the `setup_galera` function remains the same with 5 or 25 RDBMS `Host`s. The experimenter can thus easily execute its artifact in different configuration.

3.5 Modules: Effective interactions with the to-be-tested system

Until now, examples use the `run` function to implement the artifact. A straightforward function that performs shell commands on hosts. However, developing an artifact requires a lot of efforts to ensure that the experimental campaign progresses correctly, as pointed out in Sec. 2. The `run` presents some limitations to achieve this easily such as the lack of idempotent side effects, or the need for the experimenter to implement optimizations. To facilitate the programming of artifacts, ENOSLIB inherits from Ansible modules[12].

An Ansible module is a piece of code that executes operations on machines such as starting a service, installing a package or copying a configuration file. More generally, a module declares an intent of the DevOps/experimenter and its executions then ensures that machines are in the state that fulfills this intent. For example, executing the `apt` module with MariaDB and Galera on some machines ensures that MariaDB Galera cluster is installed, or installs it otherwise. This makes the module repeatable. It guarantees to keep the resources in the desired state, even during the iterative process of the artifact development that executes the module multiple times.

To run modules in Ansible, DevOps have to declare them in YAML files. ENOSLIB can read these YAML files and execute them on `Host`s with a specific role. This lets experimenters to benefit from the Ansible catalogs providing complex software stacks and extra modules (e.g., Kubernetes[13]).

However, writing maintainable and reusable YAML files implies to be proficient in Ansible: It requires to use a specific file organization and dedicated concepts.[14] To lower that barrier for experimenter, ENOSLIB seamlessly integrates with Ansible in a novel manner. It wraps the Ansible API and lets the experimenter write on-the-fly YAML files in Python functions. In contradiction to Ansible dedicated concepts, a function is a natural and common programming pattern to modularize and reuse code.

The code in Fig. 6 presents such a function. It collects metrics on `monitored` machines and stores results on `aggregator` machines. It relies on the Ansible `docker_container` module that declares to run a container on some `Host`s. During the execution, the `docker_container` module actually starts the container if it does not exist; runs it if it is stopped or suspended; and does nothing if it is already running. In comparison to shell commands, modules makes the code more expressive, efficient and less fragile. In comparison to the original Ansible, the code sticks to common programming patterns and does not require to learn new concepts to be reused.

4 ENOSLIB'S TOOLS OF THE TRADE

To make the development of artifacts easier (with respect to Section 2.2), ENOSLIB builds *services*, *tasks* and *tasks composition* upon the four previous concepts. *Services* deliver ready-to-use software stacks that are relevant across different experiments. These include network emulation, load generation and monitoring tools among others. *Tasks* are functions of the artifact that allow the execution to be suspended and resumed for flexible interactions. *Tasks*

10. https://scapy.readthedocs.io/
11. https://docs.python.org/3/library/ipaddress.html
12. https://docs.ansible.com/ansible/latest/modules
13. https://github.com/kubernetes-sigs/kubespray, https://galaxy.ansible.com/community/kubernetes
14. https://docs.ansible.com/ansible/2.3/playbooks_best_practices.html

```
1 def monitor(monitored: List[Host], aggregator: Host):
2   with ansible_on(monitored) as playbook:
3     # Expand Telegraf conf file on `monitored' Hosts
4     playbook.template(
5       src="telegraf.conf.j2", dest="telegraf.conf")
6     # Start Telegraf on `monitored' Hosts to collect data
7     playbook.docker_container(
8       image="telegraf", state="started",
9       volumes=["telegraf.conf:/etc/telegraf.conf"])
10
11  with ansible_on(aggregator) as playbook:
12    # Start InfluxDB on `aggregator' Hosts to store data
13    playbook.docker_container(
14      image="influxdb", state="started")
```

Fig. 6: Definition of a reusable function for monitoring. The code targets hosts of two roles: the `monitored` hosts to collect metrics on, and the `aggregator` hosts to store metrics to. The ENOSLIB construction `with ansible_on(hosts) as playbook` lets the experimenter to define an on-the-fly playbook that executes Ansible modules on specific hosts. See Fig. 2, lines 8-9 for a call example.

```
$ cat rdbms-with-tc.yml
 1 ---
 2 machines:
 3   - cluster: parasilo
 4     count: 3
 5     networks: [db-net,
 6       xp-net]
 7     roles: [RDBMS, site1]
 8   - cluster: parasilo
 9     count: 3
10     networks: [db-net,
11       xp-net]
12     roles: [RDBMS, site2]
13 networks:
14   - id: db-net
15     type: kavlan
16     roles: [RDBMS]
17   - id: xp-net
18     type: kavlan
19     roles: [experiment]
20 tc:
21   enable: True
22   default_delay: 75ms
23   default_rate: 1gbit
24   default_loss: 3%
25   groups: [site1,site2]
26   constraints:
27    - src: site1
28      dst: site1
29      network: db-net
30      delay: 5ms
31    - src: site2
32      dst: site2
33      network: db-net
34      delay: 5ms
35   except: [xp-net]
```

Fig. 7: Resource description with a netem specification – This configuration puts RDBMSes in two logical sites (i.e., `site1`, `site2`). The `tc` key (l .20) configures the netem service to ensure 150ms of RTT, 1Gbit/s bandwidth and 3% packet loss between those sites. Default constraints can be overridden (l .26); a delay of 10ms is observed between machines inside a given site. The `except` key (l .35) excludes the `xp-net` from the default constraints so the traffic shaping does not impact services that use this network.

composition provides additional mechanisms for remediation in case of failures.

4.1 ENOSLIB services

A service is a high-level construction that validates the resource, role and module abstractions to provide reusable "units of behavior". It is instantiated through a Python function call that enforces its configuration as a side effect. Concretely, a service bootstraps classical software stacks and hides the low-level details of their deployment to the experimenter (although experimenters feed the service with specific inputs described in its interface). Services in ENOSLIB are idempotent (thanks to their modules-based implementation) and agnostic to the testbed (by means of the resource abstraction). At the time of writing of this article, ENOSLIB is released with a few services, including Docker, Netem and an observability stack.

Docker service

Containerized deployment has become a standard way to deploy software on an infrastructure. Experimentation-wise, using containers is considered a good practice as they are reusable and versioned [5]. Moreover, all components of the deployed application are accessible through a unified API, which eases its life cycle management.

The docker service of ENOSLIB installs a local registry to cache container images. ENOSLIB supports short-life and long-life registry. In the former case, the registry only exists for the experimentation duration. This accelerates the deployment. In the latter case, the registry survives the experiment. This ensures stable binaries across different iterations of the experimentation for replicability.

Netem service

Geographically distributed studies (e.g., Fog/Edge computing) require to reproduce the observed network characteristics (e.g., delay, rate) between the real hosts. Emulating the network characteristics of the links is often a method of choice among the experimenters [18]. This method allows to easily sweep over a large set of link parameters (i.e., latency, bandwidth, packet loss) which would not be possible on a classical tesbed without emulation, even distributed.

The netem service of ENOSLIB transforms high-level constraints expressed between roles into lower-level constraints between IP addresses of the machines. These constraints are translated into traffic control commands.[15] ENOSLIB then orchestrates and applies these commands on `Hosts` to implement the traffic shaping. This service can model *complex* network constraints. For instance, Fig. 7 puts databases in logical sites and applies different constraints inside and between sites. Network conditions can be changed dynamically with updated constraints. This enables, for example, to emulate the partitioning of the network with a 100% packet loss (equivalent to a cut link). The `set_latency` function of Fig. 2 relies on the netem service. The function restricts the service to the specification of a default delay between all nodes.

Observability services

Getting insight from the to-be-tested system is crucial especially with complex stacks. Currently, ENOSLIB provides two services for this purpose: monitoring and network analyzer. These services let experimenters observe or collect information about the running or finished experiments.

The monitoring service of ENOSLIB deploys a *TIG stack*[16] to collect and present metrics to an experimenter. It enables a fine tuning of collected metrics (e.g., CPU, RAM, disk IO, opened TCP connections) and how they are sent to a store (i.e., collection interval, network to use for transfer). The service provides a `backup` method to get snapshots of collected data. This is particularly useful for postmortem

15. https://lartc.org/
16. Telegraf, InfluxDB, Grafana

analyses. The `monitor` function of Fig. 6 is an excerpt of that service.

Similarly, the network analyzer service of ENOSLIB allows a *deep inspection of the network* using Skydive.[17] It visualizes and inspects the network components of the infrastructure (e.g., software bridges, openflow rules). Traffic can be forged or captured on demand through the graphical interface or programmatically. Therefore, this service is a good choice for Software Defined Network analyses.

4.2 ENOSLIB tasks

Tasks, in ENOSLIB, enable iterative development of the artifact. They are designed to allow experimenters to repeat their executions until a desired state is reached. During the development phase, an experimenter must often stop the experiment execution (e.g., to fix it) before continuing at the same breakpoint. Unfortunately, this is not possible in a normal Python program. If experimenters stop the execution when a specific point is reached, then they lose the execution state leading to it (hidden by the Python runtime environment).

ENOSLIB offers a Python decorator to reify the state of an experiment, so every task involved in the execution workflow is filled by a data structure that represents the execution state of that experiment, called *environment*. Fig. 8 shows the use of that decorator (i.e., `@task`) that automatically adds the environment as a parameter (i.e., `env`) of any function. Executing a task could modify the environment and every modification is always saved on the experimenter's computer even if the task exits abnormally. Thanks to this mechanism, an experimenter can recall a stopped task with its execution state, and restart the experimentation where it was stopped. Note that ENOSLIB does not make any assumption on what should be stored in the environment. Instead it is the artifact responsibility to determine the relevant states.

17. https://github.com/skydive-project/skydive

```
$ cat acquire_resouces.py
@task def acquire_resources(..., env):
  # Acquire resources on the testbed (e.g. 1.2-4 Fig. 2)
  ...
  # Save RDBMS hosts in the environment
  env["RDBMS"] = hosts["RDBMS"]

$ python acquire_resouces.py

$ cat configure.py
@task def configure(..., env):
  # Read RDMBS hosts from the environment
  rdbms_hosts = env["RDBMS"]
  ...

$ python configure.py  # Access RDBMS from the previous run
```

Fig. 8: Code of a task-based experimentation. A task is a Python function decorated with the `@task` annotation. The decorator is responsible for getting the current environment state (serialized on the computer) and injecting it as a task argument (`env`). The `@task` annotation ensures that the environment is always serialized even if the function exists abnormally. In this example the `acquire_resources` function saves the RDBMS `hosts` returned by the providers. Other tasks can thus access them during subsequent runs.

4.3 Tasks composition

In ENOSLIB, tasks are commonly associated to a command-line interface. For instance, experimenters can iterate on a provisioning task from the command-line before executing a benchmark. As a consequence, this mechanism allows to debug the experimental code in an incremental way. Tasks exposed in the command-line interface can also be combined from shell scripts which is useful for fast prototyping.

With the ultimate goal of automating an experimental campaign, failures must be handled correctly and thus more control is often required. Upon failures, different actions can be envisioned: retry the action, raise an error or silently ignore the issue. ENOSLIB runs remote actions on the `Hosts` (see Sec. 3.5) using a retry logic that will silently hide transient errors to the artifact. But when the failure persists (either due to connection or software errors), ENOSLIB will raise a structured error that lets the artifact decide what actions need to be taken.

The experimenters may define the deployment of some components as critical, so the experiment cannot proceed if they fail. On the contrary, the failure of non-critical components should not jeopardize the whole experiment. In that case, ENOSLIB offers a better remediation of the execution than shell script thanks to Python exception mechanism.

5 CASE STUDIES

Various artifacts have been developed on top of ENOSLIB to conduct experimental campaigns [8], [9], [10], [13], [15], [26], [33], [37], [41]. We describe in more detail two of them. The first artifact focuses on the evaluation of different communication buses over a large geo-distributed deployment. It illustrates the use of tasks and their flexibility to handle a large set of parameters. The second one demonstrates the relevance of the resources abstraction to replicate deployment and benchmarks of OpenStack over different testbeds with the same artifact.

5.1 Benchmarking communication middleware

In this paragraph, we present an artifact that has been developed to perform an extensive study of the scalability and locality of a RPC communication middleware in edge infrastructures [33]. This study evaluated a communication middleware (the to-be-tested system) in various configurations (broker and router based). The scalability evaluation was achieved by increasing the number of connected agents. The locality evaluation was based on measuring the proportion of messages delivered locally when both the communication middleware and the agents are dispersed. For conciseness, we only focused on the scalability aspects and more precisely on the artifact we built to explore a large set of parameters and we invite readers to refer to the aforementioned publication for further information.

Fig. 9 lists the different parameters to study the scalability of the communication middleware. Two types of entities are managed: the clients (who initiate the remote calls) and the communication agents (as *Com. agents*, which handle

Parameter	Values
Clients	1 000 2 000 4 000 6 000 8 000 10 000
Com. agents	1 broker, 1 router, 3 brokers, 3 routers, 5 brokers, 5 routers
Request pattern	`Anycast`, `Unicast`
Request type	`rpc-call`, `rpc-cast`

Fig. 9: Description of a possible set of parameters for evaluating a communication middleware. In total 144 parameters are explored. Each parameter corresponds to a cartesian product of the above values.

```
p = next(parameters)
while p:
  try:
    deploy(p.clients, p.com_agents)
    bench(p.req_pattern, p.req_type)
    backup()
    done(p)
  except Exception as e:
    skip(p)
  finally:
    destroy()
```

Fig. 10: Experimental workflow. The exact code has been shortened for the sake of conciseness. This workflow allows researchers to iterate over the set of parameters.

the calls or transfer them to the target). For this purpose, we developed the ombt-orchestrator artifact.[18] This artifact defines four EnosLib tasks: `deploy`, `bench`, `backup`, and `destroy`. `deploy` interacts with the testbed (getting the resources) and with the to-be-tested system (deploying the communication middleware, the desired number of clients and servers). Additionally, a monitoring stack is used to track system metrics during the experiment. Then `bench` performs the benchmark given the request pattern and type. `backup` retrieves all the metrics and logs of the current run. Finally, `destroy` restores the execution environment in a clean state before proceeding to the next iteration. Since the deployment is fully containerized, we deem that removing all containers and associated volumes is sufficient.

The artifact (see Fig. 10) combines these tasks in a workflow that iterates over previously introduced parameters (see Fig. 9). It relies on the existence of a *sweeper* (e.g., execo sweeper [21]) that persists the information whether the current iteration is successful (`done`) or needs to be retried (`skip`) on a later run. The benefit of using EnosLib tasks is twofold. First it allows to better organize the code. Second the state of the experiment is implicitly saved between the tasks, meaning that the exact same code can be run again to handle the previously missing parameters (e.g., due to an aborted execution).

5.2 OpenStack WANwide on multiple testbeds

In 2017, a collaboration between the University of Chicago and the French Institute for Research in Computer Science (Inria) demonstrated the possibility to perform replicable experiments between the Chameleon [23] and Grid'5000 [4] testbeds using the same EnosLib-based artifact [8].

The experiment evaluated the behavior of OpenStack[19] on top of a Wide Area Network. From a bird's eye view,

18. https://github.com/msimonin/ombt-orchestrator
19. https://www.openstack.org/

```
provider: chameleon

topology:
  grp1:
    haswell:
      control: 1
      network: 1
  grp2:
    haswell:
      compute: 10
```

```
provider: g5k

topology:
  grp1:
    parasilo:
      control: 1
      network: 1
  grp2:
    parasilo:
      compute: 10
```

Fig. 11: OpenStack resources description on Chameleon (left) and Grid'5000 (right). Those descriptions further abstract the underlying resources and are translated in EnosLib's description at runtime. Groups (`grp1` & `grp2`) will be used for network emulation. `haswell` and `parasilo` refer to cluster names on the corresponding plaforms. Finally `control`, `compute` and `network` are reserved keywords for the OpenStack deployment.

OpenStack is composed of control services running in a central place and compute services enabling to horizontally scale resources. It is a large modular system with more than 100 services that can be enabled or disabled if need be. That makes OpenStack a system complicated to deploy.

To ease the deployment of and stress a geo-distributed OpenStack (the to-be-tested system), we developed the Enos[20] artifact [9]. This artifact is built on top of EnosLib resources abstraction. It also delegates to the netem service the emulation of network constraints between sites. Thus, the code that deploys, configures and stresses the geo-distributed OpenStack is the same regardless of the testbed. Executing the artifact on Chameleon and Grid'5000 is then just a matter of slightly modifying the description for the provider such as in Fig. 11 (i.e., only 3 different lines).

Fig. 12 is an excerpt of the results of the experimentation on these two testbeds. The experimentation execution produces results with the same trend on both testbeds. This validates the development of a complex artifact that achieves replicable experimental research thanks to EnosLib.

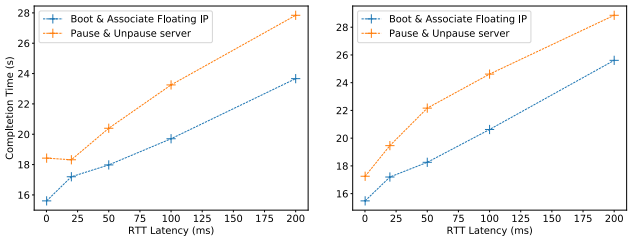


Fig. 12: Results of the execution of `Pause & Unpause server` and `Boot & Associate Floating IP` benchmarks on Chameleon (left) and Grid'5000 (right). The former starts a virtual machine (VM), pauses it and resumes it. The latter associates a public IP to a newly created VM. The average of the execution time is taken over 20 runs in warm cache.

20. https://enos.readthedocs.io/en/stable/provider/openstack.html

6 Related work

EnosLib in the DevOps tools landscape

Nowadays, an experimentation requires two distinct entities. First, the testbed where the experimentation runs. Second, the configuration management system that orchestrates remote actions on the targeted hosts. DevOps tools (Ansible, Puppet[21], Chef[22]) only help in the second case since they often assume that the infrastructure is ready. Here, ready means that the machines are up and the network connectivity is configured. However, conducting experiments on a testbed requires advanced operations such as copying SSH keys or setting up network cards in specific VLANS. It is the purpose of EnosLib's provider to fill this gap.

Similarly to EnosLib, Juju[23] integrates the management of the infrastructure. Juju does so thanks to a Domain Specific Language (DSL) as usual in DevOps tools. Commonly, a DSL targets a specific domain and so is extremely powerful but also limited to develop applications of that domain [20]. For instance, Juju is extremely powerful for configuring infrastructures, but it comes short for experimenting. In contrast, EnosLib is a library for the Python general-purpose language. Hence, experimenters are free to design useful code for their to-be-tested system. And they still can rely on DevOps tools and their DSL thanks to Python bindings.[24]

Moreover, DevOps tools abstract and control the execution in order to add more built-in behavior. Getting out of this control is a tedious and sometimes impossible task. EnosLib, being a library, inverts this control. It offers modules and services as *ready-to-use* facilities for the experiments. Experimenters can choose to use them or not. But EnosLib never locks down the execution in a specific framework. Rather, it lets experimenters in charge of it. EnosLib is thus able to carry out the diversity of experimental campaigns. Finally, automation tools are biased toward the cloud use case and operational considerations. Being cloud oriented, they benefit from the flexibility brought by cloud infrastructures in the sense the platform can be adapted (on demand) to the needs: for instance a private network can be requested with a given set of -known beforehand- IPs. However, not all experimental research can be achieved on cloud infrastructures. Some require to be executed on bare-metal platforms where the code must be adapted to the infrastructure constraints. Hence, the goal of EnosLib is to propose such abstractions and mechanisms.

EnosLib in the reproducibity landscape

Different initiatives have been presented to help developers and researchers investigate complex distributed software systems targeting repeatability, replicability, or variability. Some of them pave the way towards a set of common conventions to ensure an experiment to be easily re-executed and validated. Some other are practical implementations to help researchers build experimental artifacts. We propose now to review those we deem relevant in our context.

Popper [22] helps researchers make their experiments repeatable. It is an end-to-end set of conventions that allows to continuously (as in software continuous testing) validate an experiment result. EnosLib is a strong candidate to help experimenters implement the conventions in the field of distributed applications experiments.

Reprozip [11] is a tool to make experiments replicable by creating packages of the experiments that can then be reused. A Package is made replicable automatically and transparently by tracing at runtime the underlying system calls. This approach is generic but targets single machine experiments. EnosLib use case differs in the distributed nature of the to-be-tested system. EnosLib does not provide a native way of packaging an experiment apart from backing up entirely the filesystem involved in the experiment.

DataMill [14] proposes to automate the variations of hidden factors. As a matter of fact, hidden factors such as filesystem types or compiler optimizations can influence the experimental results. Datamill thus consists in an infrastructure of workers to run the replicas of the artifact, and a set of conventions to create deployable packages. Changing factors can be autodetected by the workers or specified by the end-user using a plugin system. Datamill targets mainly single machine experiments and is originally designed for computational experiments. In the distributed context, the same ideas apply and a library to enforce those factors could leverage EnosLib to enforce specific configurations.

DeFog [28] is a benchmarking suite designed to help developers to benchmark their applications on the cloud, the edge or both. It automates the build and deployment of the application and ensures, using containers, that the environment is similar accross all nodes. The automatization leaves less choice to an experimenter developing his artifact fit for its needs.

Kameleon [35] provides reconstructability of software appliances. Authors describe four criteria for improving user productivity: (1) easiness, (2) support during the build process, (3) container diversity, and (4) shareability. EnosLib shares the same vision by providing a library that can be easily integrated in any Python program, offering facilities to iterate on the experimental code (idempotency, task checkpoints), replicability across different architecture, and shareability provided by the Python packaging system. Additionally, Kameleon and EnosLib can be used in a complementary way by letting EnosLib deploy Kameleon appliances on a given infrastructure.

While being less expressive in term of possible workflow descriptions than Buchert et al. proposition [6], EnosLib shares the idea that idempotency is a crucial property for the workflow execution. In this paper, authors envision checkpointing as a way to restart the workflow where it last failed with the benefit of being (theoretically) agnostic to the application. EnosLib relies on a lighter but effective application-specific way of resuming the execution of a task.

Finally, EnosLib allows to gather in a single Python library different components identified in Desprez et al. proposition [16] from the resource management and basic remote execution to the management of the experimental

21. https://puppet.com/
22. https://www.chef.io/
23. https://jaas.ai/
24. To list a few Python bindings of DevOps tools:
- Ansible – https://github.com/ansible/ansible/tree/devel/lib/ansible/cli
- Chef – https://github.com/coderanger/pychef
- Juju – https://github.com/juju/python-libjuju
- Terraform – https://github.com/beelit94/python-terraform

workflow. This offers repeatibility and replicability to the experimenter. Moreover, ENOSLIB can serve as a building block to provide automatic variability of an experimental setup and packaging of a distributed experiment as it is done for computational experiments.

7 LESSONS LEARNED

Started in late 2016, ENOSLIB has significantly evolved along with the adoption of new users[25]. From the Enos solution [9], our team has developed an experimenters-centric library: evolutions are driven by the experimenter's use-cases, theirs needs and their contributions. In this section, we discuss three lessons we learned while developing, maintaining and disseminating ENOSLIB and one more general lesson on the development of experimental artifacts.

L1: *Documentation and code examples are key aspects of the project*. It has evolved over time and converged nowadays in a two levels documentation: a user documentation and an API documentation. The user documentation contains plenty of small, self-content and *ready-to-adapt*, snippets of code. These numerous snippets are thus very beneficial as they consist of minimal viable examples than can be composed by the users when crafting a new experiment. They also act as good functional tests. The API documentation contains both more background and technical details about the implementation.

L2: *User experience is decisive in the adoption of the library*. Besides the concepts discussed in this article, we continuously add features that ease the experimenter's life. Some of them, like automatic retries (e.g., deployments of base Linux image), transparent caching (e.g., of remote REST API calls), overcommitting resource allocations or integration with interactive environments (e.g., Jupyter) improve greatly the user experience. Following [43], programmatic interfaces to build ENOSLIB's objects (e.g., Configuration) is proposed and favored as this makes the code less error-prone and allows to raise self-explainable errors. Although these features can be seen as secondary compared to the main concepts of ENOSLIB, we have observed that they also contribute to the adoption of the library.

L3: *Users turn-over is high*. Users usually work only on a limited timeframe on experimental activities. As a consequence it is often hard for them to upstream changes to the code (or report bugs) in this period of time. This is why we pro-actively help the users develop their artifacts and deliver bug fixes/enhancements in a timely manner. The ultimate goal of this process is to capture the various experimenter's practices (before loosing them) and deliver them, through library's new components, to the whole community.

L4: *A tradeoff between artifact genericity vs complexity should be found*. As shown in 5.2, ENOSLIB helps in building replicable artifacts across different testbeds. Porting from one testbed to another is facilitated if they offer similar characteristics: base Linux distribution (e.g., Debian/Ubuntu) or system architecture (x86). Fully decoupling the artifact code from the underlying distribution and architecture is close to impossible, often unnecessary and may lead to a very complex code. It's rather the experimenter's responsability to ensure that the overall artifact is consistent and to find the good trade-off between genericity and complexity. For this purpose, ENOSLIB's *modules* can be adapted to fit some of these characteristics (e.g., `yum` module targets RHEL while `apt` targets Debian systems). Also some of the ENOSLIB's *services* are gradually updated to target a wider range of systems when deem relevant. For instance the monitoring service supports both x86 and ARMv7 based nodes to cover the diversity of hardware available on Grid'5000 and FIT IoT-Lab infrastructures.

Finally note that the library is under a GPLV3 licence and most evolutions are discussed freely (for further information see https://gitlab.inria.fr/discovery/enoslib).

8 CONCLUSION

In this article, we introduced ENOSLIB, a dedicated library for experiment-driven research in distributed computing. ENOSLIB proposes various abstractions to help researchers implement extensible and replicable artifacts. To make the development phase easier, ENOSLIB modifies the Python execution model with a richer task granularity offering an efficient mean to develop an artifact in an iterative manner. It also integrates state-of-the-art services that are useful in most experiment driven activities such as network traffic shaping, container technologies, monitoring stacks, and so on. For the execution, providers, resource models and roles abstractions allow the development of experimental artifacts in an agnostic manner avoiding any lock-in with the underlying testbed. This is a significant added value as it allows researchers to validate the replicability of the results as strongly promoted in top-ranked conferences. At the time of the drafting this article, ENOSLIB implements six providers (Vagrant, Grid'5000, Vmon, OpenStack, Distem and FIT IoT-Lab) that enables researchers to acquire resources and execute the artifact on such testbeds transparently.

Based on the success of the library within the OpenStack community[26], we recently released a similar engine for Kubernetes[27].This engine has has been already used to conduct experiment-driven activities [25]. In parallel, new experimental engines leveraging ENOSLIB have been implemented for other software stacks such as the InterPlanetary File System (IPFS) or the blockchain technology from other research groups. Although they are more ad-hoc artifacts, these initiatives[28] are important because they can lead to the integration of new extensions in ENOSLIB. As an example, the Locust HTTP load injection has been recently integrated. These extensions could be wider than the state-of-the-practice tools and cover for instance experiment methodologies [27], [29], [34], enabling experimenters to get additional guarantees about the artifact. Those effective and possible integrations show the flexibility brought by a library approach in comparison to a framework that

25. https://discovery.gitlabpages.inria.fr/enoslib/theyuseit.html

26. https://superuser.openstack.org/articles/collaborations-cross-industries-openstack-neutron-and-discovery-open-science-initiative/

27. https://pypi.org/project/enos-kubernetes/

28. https://discovery.gitlabpages.inria.fr/enoslib/theyuseit.html

could limit the operations that can be performed by experimenters.

Our long term goal is to gather academic and industrial actors of our community around EnosLib in order to benefit from development efforts of experimental artifacts made by any research group. If tools such as Yahoo Cloud Serving Benchmark have been contributing to our community for the past ten year significantly, we claim that our community could go one step further by pooling our efforts to deliver not only the benchmark but also the complete artifact that covers the whole cycle of the experimental campaign.

Acknowledgements

All developments related to EnosLib have been supported by Inria and Orange Labs in the context of the Discovery Open Science initiative. Experiments were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations.

The authors would like to thank all the EnosLib contributors and the users for the feedback and encouragements.

References

[1] ACM Artifact Review and Badging. https://www.acm.org/publications/policies/artifact-review-badging, Apr. 2018.

[2] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne. FIT IoT-LAB: A Large Scale Open Experimental IoT Testbed. In *IEEE World Forum on Internet of Things (IEEE WF-IoT)*, Milan, Italy, Dec. 2015.

[3] J. Aldrich. The power of interoperability: Why objects are inevitable. In *Onward! 2013*, Indianapolis (IN), USA, Oct. 2013.

[4] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lebre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In *CLOSER 2012*, Porto, Portugal, Apr. 2012.

[5] C. Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operative Systems Review*, 49(1):71–79, 2015.

[6] T. Buchert, L. Nussbaum, and J. Gustedt. A workflow-inspired, modular and robust approach to experiments in distributed systems. In *CCGRID 2014*, Chicago (IL), USA, May 2014.

[7] M. Burgess. Testable system administration. *Commun. ACM*, 54(3):44–49, 2011.

[8] R.-A. Cherrueau, A. Lebre, and P. Riteau. Toward Fog, Edge, and NFV Deployments: Evaluating OpenStack WANwide. OpenStack Summit, Boston (MA) USA, https://www.openstack.org/videos/summits/boston-2017/toward-fog-edge-and-nfv-deployments-evaluating-openstack-wanwide, May 2017.

[9] R.-A. Cherrueau, D. Pertin, A. Simonet, A. Lebre, and M. Simonin. Toward a holistic framework for conducting scientific evaluations of openstack. In *CCGRID 2017*, Madrid, Spain, May 2017.

[10] R.-A. Cherrueau, M. Simonin, and A. van Kempen. EnosStack: A LAMP-like stack for the experimenter. In *INFOCOM Workshops*, Honolulu (HI), USA, Apr. 2018.

[11] F. Chirigati, R. Rampin, D. Shasha, and J. Freire. ReproZip: Computational reproducibility with ease. In *SIGMOD'16*, San Francisco (CA), USA, July 2016.

[12] A. L. Couch and Y. Sun. On the algebraic structure of convergence. In M. Brunner and A. Keller, editors, *Self-Managing Distributed Systems, 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, DSOM 2003, Heidelberg, Germany, October 20-22, 2003, Proceedings*, volume 2867 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2003.

[13] H. Coullon, C. Perez, and D. Pertin. Production deployment tools for iaases: An overall model and survey. In *FiCloud*, Prague, Czech Republic, august 2017.

[14] A. B. de Oliveira, J.-C. Petkovich, T. Reidemeister, and S. Fischmeister. DataMill: Rigorous performance evaluation made easy. In *ICPE'13*, Prague, Czech Republic, Apr. 2013.

[15] M. Delavergne, R.-A. Cherrueau, and A. Lebre. Keystone in the context of fog/edge massively distributed clouds. OpenStack Summit, Vancounver, Canada, https://www.openstack.org/videos/summits/vancouver-2018/keystone-in-the-context-of-fogedge-massively-distributed-clouds, May 2018.

[16] F. Desprez, G. Fox, E. Jeannot, K. Keahey, M. Kozuch, D. Margery, P. Neyron, L. Nussbaum, C. Pérez, O. Richard, W. Smith, G. Von Laszewski, and J. Vöckler. Supporting Experimental Computer Science. Research Report RR-8035, INRIA, July 2012.

[17] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS'19*, Providence (RI), USA, Apr. 2019.

[18] S. Hemminger. Network emulation with NetEm. In *LCA 2005*, Canberra, Australia, Apr. 2005.

[19] L. Hochstein and R. Moser. *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way*. O'Reilly Media, 2nd edition, 2017.

[20] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society, June 1998.

[21] M. Imbert, L. Pouilloux, J. Rouzaud-Cornabas, A. Lebre, and T. Hirofuchi. Using the EXECO toolkit to perform automatic and reproducible cloud experiments. In *CloudCom 2013*, Bristol, UK, Dec. 2013.

[22] I. Jimenez, M. Sevilla, N. Watkins, C. Maltzahn, J. Lofstead, K. Mohror, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. The Popper convention: Making reproducible systems evaluation practical. In *IPDPSW 2017*, Lake Buena Vista (FL), USA, May 2017.

[23] K. Keahey, P. Riteau, D. Stanzione, T. Cockerill, J. Mambretti, P. Rad, and P. Ruth. Chameleon: A scalable production testbed for computer science research. In *Contemporary High Performance Computing: From Petascale toward Exascale*, volume 3, pages 124–148. 2019.

[24] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA'87*, Orlando (FL), USA, Oct. 1987.

[25] K. Manaouil and A. Lebre. Kubernetes wanwide: a deployment scenario to expose and use edge computing resources? In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 193–197. IEEE, 2021.

[26] V. Marangozova-Martin, N. de Palma, and A. El Rheddane. Multi-level elasticity for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2326–2337, 2019.

[27] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci. Taming performance variability. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 409–425, 2018.

[28] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese. Defog: fog computing benchmarks. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 47–58, 2019.

[29] A. V. Papadopoulos, L. Versluis, A. Bauer, N. Herbst, J. von Kistowski, A. Ali-Eldin, C. L. Abad, J. N. Amaral, P. Tuma, and A. Iosup. Methodological principles for reproducible performance evaluation in cloud computing. In M. Felderer, W. Hasselbring, R. Rabiser, and R. Jung, editors, *Software Engineering 2020, Fachtagung des GI-Fachbereichs Softwaretechnik, 24.-28. Februar 2020, Innsbruck, Austria*, volume P-300 of *LNI*, pages 93–94. Gesellschaft für Informatik e.V., 2020.

[30] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[31] R. Ricci, E. Eide, and C. Team. Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications. *; login:: the magazine of USENIX & SAGE*, 39(6):36–38, 2014.

[32] J. Rojas Balderrama and L. Miguel. Identify potential bottlenecks in the neutron api, plan a performance testing and more. OpenStack Superuser blog, https://superuser.openstack.org/articles/collaborations-cross-industries-openstack-neutron-and-discovery-open-science-initiative/, 2020.

[33] J. Rojas Balderrama and M. Simonin. Scalability and locality awareness of remote procedure calls: An experimental study in edge infrastructures. In *CloudCom 2018*, Nicosia, Cyprus, Dec. 2018.

[34] D. Rosendo, P. Silva, M. Simonin, A. Costan, and G. Antoniu. E2clab: Exploring the computing continuum through repeatable, replicable and reproducible edge-to-cloud experiments. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 176–186. IEEE, 2020.

[35] C. Ruiz, S. Harrache, M. Mercier, and O. Richard. Reconstructable software appliances with kameleon. *ACM SIGOPS Operating Systems Review*, 49(1):80–89, 2015.

[36] G. K. Sandve, A. Nekrutenko, J. Taylor, and H. E. Ten simple rules for reproducible computational research. *PLoS Computational Biology*, 9(10):e1003285, 2013.

[37] D. E. Sarmiento, A. Chari, L. Nussbaum, and A. Lebre. Multi-site connectivity for edge infrastructures - diminet:distributed module for inter-site networking. In *CCGRID 2020 (to appear)*, Melbourne, Australia, May 2020.

[38] L. Sarzyniec, T. Buchert, E. Jeanvoine, and L. Nussbaum. Design and evaluation of a virtual experimental environment for distributed systems. In *PDP'13*, Belfast, UK, Apr. 2013.

[39] C. Séguin, E. Caron, and S. Dubus. SeeDep: Deploying reproducible application topologies on cloud platform. In *CLOSER 2019*, Heraklion, Crete, Greece, May 2019.

[40] R. Shambaugh, A. Weiss, and A. Guha. Rehearsal: A configuration verification tool for Puppet. In *PLDI'16*, Santa Barbara (CA), USA, June 2016.

[41] M. Simonin, D. Belova, and A. Shaposhnikov. Chasing 1000 nodes scale. OpenStack Summit, Barcelona, Spain, https://www.openstack.org/videos/summits/barcelona-2016/chasing-1000-nodes-scale, Oct. 2016.

[42] M. H. R. R. L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX ATC'08*, Boston (MA), USA, June 2008.

[43] C. Tang, T. Kooburat, P. Venkatachalam, A. Chander, Z. Wen, A. Narayanan, P. Dowell, and R. Karl. Holistic configuration management at facebook. In E. L. Miller and S. Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 328–343. ACM, 2015.

[44] G. Von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Voeckler, R. J. Figueiredo, J. Fortes, et al. Design of the FutureGrid experiment management framework. In *GCE Workshop 2010*, New Orleans (LA), USA, nov 2010.

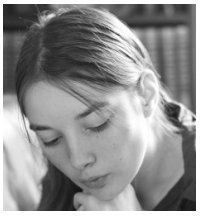
Alexandre van Kempen is currently a post-doctoral fellow in the STACK team, at Inria Rennes-Bretagne Atlantique. He received his Ph.D. degree in Computer Science from the Universite of Rennes in 2013. His research interests cover distributed systems, with a main focus on storage systems, erasure coding and fog/edge paradigms.

Adrien Lebre is a full professor at IMT Atlantique (France), head of the STACK research group, and PI of the Open Science Discovery Initiative. He holds a Ph.D. from Grenoble Institute of Technologies and a habilitation from University of Nantes. His activities focus on large-scale distributed systems, their design, compositional properties and efficient implementation. Since 2015, his activities have been mainly focusing on the Edge Computing paradigm, in particular in the OpenStack ecosystem.

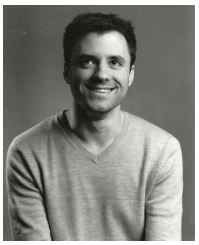
Dimitri Pertin is a post-doctoral fellow at the Laboratory of Digital Sciences of Nantes (France). He received a MEng in systems and networks; and a PhD in computer science, both from the University of Nantes. His research deals with distributed application deployment and reconfiguration on dynamic and massively distributed infrastructures.

Ronan-Alexandre Cherrueau is a research engineer at Inria (France). He received the PhD degree from the École des Mines de Nantes, France in 2016, where he defined a functional language to write cloud application that preserves users' privacy. His work focuses on massively distributed cloud computing infrastructures for the edge computing. His research interests consider programming language and distributed applications.

Javier Rojas Balderrama is a research engineer at Inria (France). His research interests include distributed computing, neuroinformatics, and computational semantics. He received a MSc in cognitive sciences from the Grenoble Institute of Technology, and a PhD in computer science from the Nice Sophia Antipolis University, France.



Marie Delavergne is a PhD candidate at Inria (France). She received a Master in software architecture from the University of Nantes in France, in 2018. Her research interests are massively distributed cloud computing infrastructures for edge computing. She is the main developer of Juice.



Anthony Simonet received a PhD degree from the École Normale Supérieure de Lyon, France, in 2015. He is now research scientist at IExec (France). Previously, he was post-doctoral associate with the Rutgers Discovery Informatics Institute in New Jersey, USA. His research spans over multiple domains including distributed data management, models and middleware for hybrid distributed computing infrastructures, high performance computing and energy.

Matthieu Simonin is a permanent research engineer at Inria Rennes–Bretagne Atlantique, France. He is involved in various activities around distributed systems validation, big data and stream processing analysis. He is the maintainer of ENOSLIB.