nepi-ng: an efficient experiment control tool in R2lab

Thierry Parmentelat
Thierry Turletti
Walid Dabbous
Université Côte d'Azur, Inria
firstname.lastname@inria.fr

Mohamed Naoufal Mahfoudi
Université Côte d'Azur, Inria
mohamed-naoufal.mahfoudi@inria.fr

Francesco Bronzino
Inria, France
francesco.bronzino@inria.fr

ABSTRACT

Experimentation is an essential step for realistic evaluation of wireless network protocols. The evaluation methodology entails controllable environment conditions and a rigorous and efficient experiment control and orchestration for a variety of scenarios. Existing experiment control tools such as OMF often lack in efficiency in terms of resource management and rely on abstractions that hide the details about the wireless set-up. In this paper, we propose `nepi-ng`, an efficient experiment control tool that leverages job oriented programming model and efficient single-thread execution of parallel programs using `asyncio`. `nepi-ng` provides an efficient and modular fine grain synchronization mechanism for networking experiments with light software dependency footprint. We present and discuss our design choices and compare to the state of the art tools, mainly OMF.

KEYWORDS

Experimentation; Orchestration; Reproducibility; asynchronous programming; Python; `asyncio`.

1 INTRODUCTION

Validation and performance evaluation of new network protocols are done using different complementary approaches such as analytical modeling, simulation, experimentation or any combination of them [1]. In the wireless networking domain, modeling and simulation results may not be realistic enough, because the interaction between MAC and physical layers is complex to model due to the random behavior of the wireless environment. It is therefore essential for the adoption of the proposed wireless network protocol by the community and the industry to run experiments with real hardware and software components to get a meaningful evaluation. By choosing the experimental approach, the validation process requires the reproducibility of experiments. Thus, in order to ensure the reliability of the measurements and the inferred results, the evaluation methodology entails a rigorous experiment control and orchestration as well as the availability of artifacts [2]. While the realistic character of experiments is important for the overall validation process, wireless channel phenomena (e.g., interference, multipath), scalability issues and the variety of experiment scenarios, pose serious challenges for the experiment control efficiency [3].

Among the existing experiment control tools, OMF [4] happens to be the most deployed one in the major wireless networking testbeds, as it offers an efficient approach for measurement data collection and a straightforward control mechanism for experiment control. However, OMF lacks in efficiency in terms of resource management. Even though OMF proposes a layer of abstraction that simplifies the experiment setup, we argue that such approach can be detrimental to the overall control of the experiment, and adds other sources of uncertainty as to the actual experimental environment.

R2lab addresses precisely this controllability dimension. Being a remotely accessible wireless testbed located in an anechoic chamber, R2lab allows a fine-grained control over the wireless environment thanks to the limited number of fixed multipath sources (e.g., due to hardware metallic enclosure boxes) and the absence of outside interference. With `nepi-ng`, a tool for running and orchestrating network experiments, R2lab provides an ecosystem to run controlled experiments where the impact of unwanted variability is reduced to the minimum.

`nepi-ng` has been designed with the ambition to address the following high-level challenges:

- *Efficiency*: because an experiment is likely to be run over and over again with a combination of slightly different environments, it is important to remove all possible overhead, even when an experiment is remotely controlled, which is the desired usage model.
- *Light software dependency footprint*: the overall software dependencies should be kept to a strict minimum, so that the approach can be applicable in a wide variety of contexts and testbeds.
- *Modularity*: allowing pieces of code to be re-used or shared is highly desirable, like for any software development activity.

The plan of the rest of the paper is as follows. Section 2 presents the R2lab hardware together with its architecture and testbed management tools. In section 3, we lay down the paradigm proposed by `nepi-ng` as an orchestration

tool. Then in section 4, we present the state of the art, and in section 5 we discuss pros and cons of our approach. Section 6 concludes the paper.

2 OVERVIEW OF R2LAB

2.1 Hardware

The R2lab platform sits in an insulated anechoic chamber located in the basement of a building at Inria, Sophia Antipolis, France. Figure 1 shows a snapshot from inside the room. It gives access to regular computers with common Wi-Fi interfaces, a choice of Software-Defined-Radio (SDR) devices, and a couple of controllable commercial phones.

Figure 1: R2lab room

The room size is about 90m^2, roughly 11m x 8m, although its shape is not a plain rectangle, as shown on Figure 2. This picture shows the ground plan layout of the nodes that are arranged in a grid with a spacing of about 1.0m and 1.15m in both horizontal directions, except for the two pillars in the room. This layout allows running various scenarios with wireless nodes that can be either line of sight, near-line-of-sight or non-line-of-sight.

It is insulated from the outside electromagnetic conditions by a Faraday cage and uses RF absorbers to drastically attenuate reflections on the copper foils.

The 37 wireless nodes are Icarus off-the-shelf computers* with CPU Intel® Core™ i7-2600, 8M Cache at 3.40 GHz, 8GB RAM, 240GB SSD.

Each node features two Wi-Fi MIMO NICs dedicated to experimentation: one Atheros AR9380 and one Intel 5300; each of these two models has pros and cons in terms of low-level hardware and software capabilities, so offering them both increase the spectrum of possible experiments.

*Icarus node: https://nitlab.inf.uth.gr/NITlab/.

Figure 2: R2lab topology

In addition to these two common cards, about half of the Icarus nodes are attached to an SDR device. In order to expose a physical substrate that is as diverse as possible in terms of possible experiments, various makes and models are present depending on the node, among several types of USRP's from Ettus, together with LTE USB dongles, and LoRa USB boards.

Each node also has 3 Gigabit Ethernet interfaces: one connects to a NITlab's Chassis Manager Card (CMC), that allows to manage power and reset the motherboard or a USB device; one is used by the testbed management framework for providing access; the last one is entirely left to the experiment - i.e. it is not managed at all by the testbed management software - and can come in handy for creating a wired data plane needed by the experiment, like e.g. a 5G infrastructure link. Note that on a few nodes, this interface is used to connect a USRP2 or N210 SDR device, as these models can only be connected through an RJ45 Gigabit interface.

Finally, two commercial phones (Nexus 5 and Moto E 4G) are also available right inside the chamber. Each one is connected through USB to a computer including convenience helpers to manage the phone remotely.

2.2 Architecture

The testbed architecture relies on a front-end gateway that allows to control and reach nodes through ssh; Note that you don't need to first connect to the gateway in order to run the experiments, but you can run the `nepi-ng` script from your own machine. This can be very convenient for experimenters and is not available in other testbeds such as ORBIT [5].

Two additional services are hosted in separate virtual machines, for (a) running a database and API, and (b) offering an all-purpose website†. The software used for controlling

†gateway on `faraday.inria.fr`; website on `r2lab.inria.fr`; API on `r2labapi.inria.fr`.

and managing nodes, including loading and saving images, is named `rhubarbe` [6] and is itself written in `nepi-ng`, which we describe in the next section.

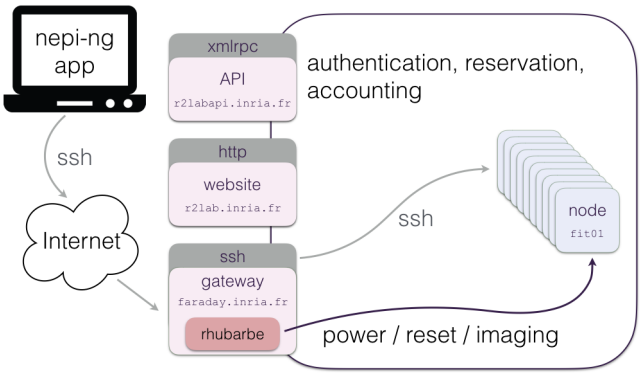

Figure 3: R2lab architecture

3 NEPI-NG

Proposed as an optional software companion to the R2lab testbed, `nepi-ng` is a modern experiment control tool, which allows R2lab users to script their experiments. As it primarily only relies on ssh connections, it can also fit many other uses if needed, as explained in section 5.1

3.1 Design choices

To answer the challenges mentioned in Section 1, the following design choices were made further down the path towards implementation.

As far as parallelism and synchronization are concerned, we have adopted a job-oriented programming model, where dependencies are explicit between jobs. In other words, each of the programming blocks, or jobs, is defined with an explicit list of jobs that it depends on. This allows for an explicit and visual representation of dependencies, as it will be illustrated below. This is typically in contrast to message-based synchronization, and this point is further discussed in section 5.

On a similar note, we made a second design choice, which is to keep the number of abstractions as low as possible, and to avoid defining new ones when it is not strictly necessary. This is a subtler point, which may be better illustrated through an analogy in the completely different domain of data visualization. In that field, a lot of tools offer abstractions like histograms or boxplots or else. As a result, creating a real-life figure is made easier at first, as compared with having to deal with the gory details of, say, a histogram. However it is unclear if this added-value still holds when additional decorations are required, as it often involves calls to an endless string of specific and hard to remember features, typically for tweaking subfigures, tickers and legends, titles, etc. On the other hand `d3.js` [7], arguably one of the most successful visualization libraries today, has gone a completely different path, in that the underlying basic objects, namely SVG elements, are fully exposed to the programmer user; the added value of `d3.js` is to foster a general workflow between data and graphical elements, and not to try and hide them behind abstractions[‡]. Our design choice for `nepi-ng` is comparable: it offers a small number of low-level abstractions like nodes and commands, that are very closely related to ssh internals, but does not attempt to intrude in the domain of, for example, wireless setup. We argue that the actual details of how a wireless device is setup are too important to a wireless experiment, for them to remain implicit, i.e., deferred to an experiment controller tool or third party library. The added value of `nepi-ng` in this area is instead focused on providing an efficient paradigm for orchestrating this setup, and a clean way to write these gory details using the most appropriate tool - often a plain shell script.

3.2 Experiment programming model

3.2.1 A simple example. As a first illustration of how these design decisions shape a `nepi-ng` experiment, let us consider Figure 4, which depicts the logical ordering of a basic and typical experiment: using two nodes as one sender and one receiver, we want to configure wireless devices on both ends, and then record data at the receiving end.

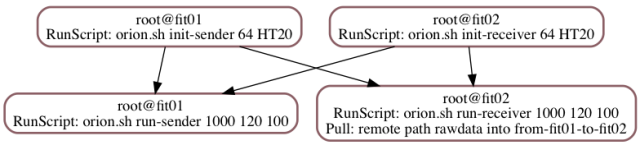

Figure 4: Simplest send-receive experiment

This figure was automatically derived from the Python code that implements a real-life experiment, designed as part of another work [8] that deals with Orientation Estimation. It should give the reader an intuitive grasp on the programming model. In the rest of this section, we describe in further details the various concepts at work in this example and beyond.

3.2.2 Jobs and schedulers. At the heart of `nepi-ng` are the notions of jobs and schedulers. Quite usually, a job describes a sequential portion of a program, and a scheduler is a set of jobs, that are linked together with a *requires* relationship;

[‡]Although of course some second-tier tools do offer such abstractions on top of `d3.js`.

when job b requires job a, this naturally means that b cannot start until a is done. This is illustrated graphically with an arrow $a \longrightarrow b$. A job can have any number of requirements, but the model imposes for the graph to be acyclic. This means that once a job is completed, it will never run again. At least one scheduler is needed to run any job, so that the example from Figure 4 involves 4 jobs - the rounded corner boxes - in one scheduler, although this top-level scheduler is not rendered graphically.

3.2.3 Single-threaded concurrency using `asyncio`. `nepi-ng` relies entirely on Python, and in particular on its asynchronous programming model. Starting with version 3.5, Python proposes a coroutine-based paradigm, leveraged in the standard `asyncio` [9] library, as well as a few other non standard ones like `curio` [10] and `trio` [11]. This innovative programming style allows for **single-threaded** execution of otherwise parallel programs, and provides a radically different solution for race conditions.

`nepi-ng`'s mechanism of schedulers and jobs merely adds the notion of time dependencies between coroutines. Please note that `nepi-ng` is not a Python library in itself, it is actually the union of two libraries; this first set of functionalities is implemented in the `asynciojobs` [12] library.

3.2.4 ssh-oriented jobs. The second half of `nepi-ng` is called `apssh` for asynchronous parallel ssh [13]. The objective here is to expose a few primitives for creating job objects that support remote execution through ssh. This is done primarily by combining three categories of Python objects, as illustrated on Figure 5.

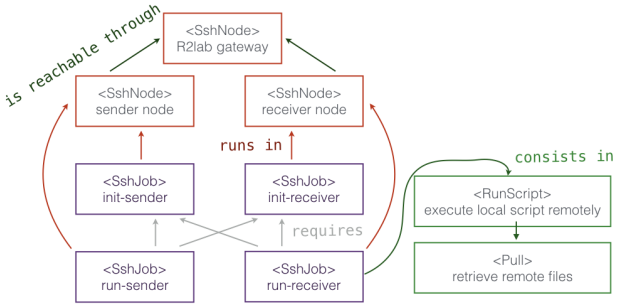

Figure 5: `nepi-ng` objects and their relationships

(a) Node objects are all of the *SshNode* class; they describe ssh connections and can be nested to materialize 2-hop connections, like is frequently needed in networking testbeds where actual resources are only reachable through a gateway host. (b) Command objects come in a few flavours, like *Run* to invoke a remote command as with regular ssh, *RunScript* for running a local script remotely, as well as *Push* and *Pull* for file transfers. (c) Finally, *SshJob* instances allow tying these two dimensions, namely what needs to be done and on what node, into a single *SshJob* object that is suitable to depend on other jobs and to be scheduled.

One central property of this menagerie of objects is that by design, at most one ssh connection is created for each *SshNode* object, and all the commands attached to that instance share that connection. This is a crucial point as far as performance is concerned, as it allows running drastically faster than implementations that rely on a separate ssh client process.

A glimpse at an example `nepi-ng` code is given on Figure 9 in appendix, that puts all these pieces together in a fragment that implements the workflow of Figure 4, and where `nepi-ng` entities have been outlined in purple. In this real-life experience, the experimenter has chosen to write the actual body of the 4 individual jobs in a separate shell script; an extract is given on Figure 10. This is an approach that can be recommended, as it allows to cleanly separate on the one hand the overall logic that belongs in the `nepi-ng` script, and on the other hand the gory details, which are best expressed in other languages or tools; here a shell script is used to unload and reload the iwl-wifi driver for controlling the Intel 5300 agn cards. The script enables the raw packet injection mode by setting up the wireless cards to work in the monitor mode as well as by configuring them to operate on the same frequency and bandwidth.

It is important to note that using a companion script allows keeping all these details explicit, and thus contributes to making the experiment more reproducible by others and/or in other experimental setups.

3.2.5 Semantics of schedulers. Coming back to the synchronization mechanisms offered by schedulers and jobs, let us now be more explicit on the actual semantics of these objects. As mentioned already, the dependency graph within a scheduler's jobs must be acyclic. When executing a scheduler, all its jobs with no requirement are started, and as they complete, jobs downstream can be started in turn once all they requirements are completed.

Naturally a scheduler completes when all its jobs have completed. However this simple mechanism is not always flexible enough. Considering the *send and receive* example of Figure 4, although the sender process will easily know exactly when to stop, it is not the case on the receiver end. So a first approach is to write a receiver that estimates the experiment duration based on some context - e.g., number and frequency of packets involved in the experiment - and stops on its own after that time.

Although a workable approach, this is not entirely satisfactory: it would be safer if we could instead use synchronization here again. As much as stating "b requires a" allows us to synchronize *the beginning* of b with the end of a, this

mechanism seems unable to let us synchronize *the end* of a job - here the receiver - with *the end* of another one - the sender.

In order to address this kind of needs, `nepi-ng` offers more advanced mechanisms. The first one is the notion of so-called *forever* jobs. A *forever* job is one that is *not waited for*, but that instead gets canceled by the scheduler when all its regular jobs have completed. Figure 6 shows our initial logic, where the receiver end is defined as a *forever* job - which is outlined with a **dotted border**.

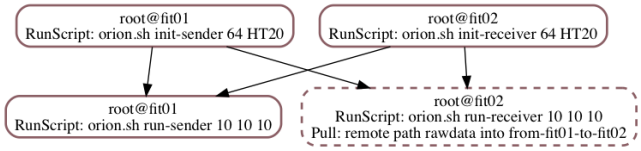

Figure 6: Send-receive with a *forever* job

This refinement of the scenario would not work as expected though; its behavior would indeed be for the lower-right job to be canceled once the sender is done, but that would also mean that no data collection could take place.

To solve this problem, we need another feature of `nepi-ng`, which is the ability to create so-called *nested* schedulers. As a matter of fact, a scheduler object is a job in itself, and so can it be inserted at a higher level in order to create hierarchical schedulers. This is illustrated on Figure 7. In this iteration of the same scenario, the sender and receiver jobs are the only two members of a nested scheduler - represented with sharp corners. With this scenario, the receiver job does not need to estimate its duration: it can simply run forever since it will be canceled when the sender job is over. If needed, inserting a safety delay - e.g. to account for all the packets to reach the receiver - can simply be done as an extra step at the end of the sender job.

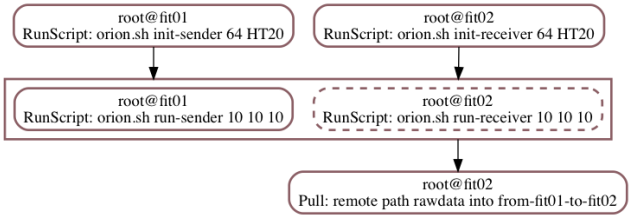

Figure 7: Send-receive with a nested scheduler

Nested schedulers also drastically improve code reusability; typically, a real-life experiment needs to support daily operations like loading a specific image on selected nodes, and turning off the rest of the testbed for for avoiding unintentional interference. Thanks to nested schedulers, it is possible to write helper tools that can for example decorate the scheduler of Figure 7 and embed it into a higher level one that takes care of those operations, resulting in the scheduler depicted on Figure 8.

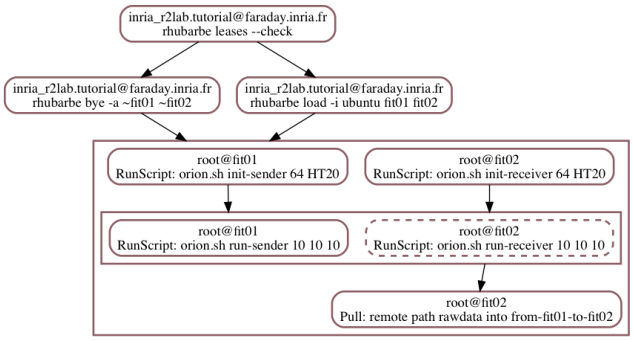

Figure 8: Nesting helps reusability

3.2.6 Data flow. The jobs model currently does not support any kind of data flow; that is to say, one could imagine exposing, as an input to a given job, the results of its required jobs. We have considered, but rejected this option, as it resulted in extra complexity for a very low added value, at least in our application domain: our needs are typically to cleanly manage event-based and IO-bound orchestrations, we are not addressing parallel computing, where such a mechanism would on the contrary be of paramount importance.

Supporting data flow propagation would require in particular some mechanism to allow a programmer to bind incoming values - the ones produced by its upstream jobs - to its programming namespace; this would imply some form of naming within jobs, which is otherwise not needed. To make things worse, a real-life scenario typically supports a variable number of nodes: e.g., the same scenario can be used with a flock of 3 or 15 nodes. This means that when writing a given job, one does not know, nor most of the time care about, how many upstream jobs we have.

3.2.7 Control flow. On the other hand, there is a limited form of control flow, in the sense that any exception raised inside a job triggers an abrupt termination of the whole scheduler; in situations where this behavior is not desirable, a job can be marked as non critical, in which case the job gets canceled, but the overall scheduler proceeds.

3.2.8 Resource control. Finally, a scheduler can be subject to limitations, in terms of either a predefined timeout, or in terms of a maximal number of concurrent tasks, which can be helpful for example when dealing with a large number of nodes, which is a frequent situation with larger experimental testbeds like PlanetLab.

4 STATE OF THE ART

The following section summarizes the approaches to control management employed in two of the most widely used research testbeds: ORBIT and Emulab.

ORBIT. The ORBIT testbed [14] is a two-tier wireless network emulator/field trial designed to achieve reproducible experimentation. Its main facility is the radio grid testbed that uses a 20x20 two-dimensional grid of programmable radio nodes.

Experiment realization on ORBIT is managed through the ORBIT Management Framework (OMF) [4]. First developed for ORBIT (but now used in a diverse set of testbeds, e.g. GENI [15], NITOS [16], w-iLab.t [17]), OMF's architecture is divided into three planes: the control plane, the measurement plane - handled through the ORBIT Measurement Library (OML), and the management plane. The key role of the Control Plane is to provide researchers with tools and methods to systematically develop and orchestrate their experiments.

A domain specific language - the OMF Experiment Description Language, OEDL - allows an experimenter to write an Experiment Description including resource requirements, their initial configuration, and a state machine describing the time/event-triggered actions required to realize the experiment. An experiment description is composed mainly of two parts: 1) the resource requirements and configurations (e.g. IP address to be used for an interface); 2) task descriptions, which are essentially contained in a state-machine that enumerates the different events, states, and associated tasks to perform with the resources in order to realize the experiment.

Emulab. The Emulab testbed [18] was first designed as an experimental emulation platform for distributed systems and networks. For this reason, the general design of the Emulab control software (simply referred to as Emulab software) differs from OMF based on its key requirements.

To support dynamic experiment control, Emulab uses an event system to extend the notion of signals across sets of nodes and links. This facility closely mirrors the style of event schedulers found in network simulators. Just as with simulation, experimenters are allowed to manipulate link characteristics at prescribed times, so they can dynamically change latencies, bandwidths, and loss rates on emulated links. Node configuration is driven by the nodes themselves, but entirely controlled by state stored centrally in a centralized database. Emulab nodes load the state from this central system to achieve distributed self-configuration, which includes obtaining host names, loading the disk image, and executing startup scripts.

Since its inception, the Emulab testbed has integrated into its architecture a diverse set of resources that allow for experimentation with 802.11 Wireless and Software-Defined Radios. Moreover, the control software is now deployed at more than 36 other locations.

Next iterations. The recent emphasis given to the standardization of the fifth generation of wireless standards has pushed towards a renewed interest in developing city scale testbeds that integrate radically new technologies in real world urban environments. In the US, the NSF funded PAWR program [19] will in the close future support the development of two testbeds: COSMOS [20] and POWDER [21]. These new experimental architectures build on the control software developed for the ORBIT (OMF) and Emulab (Emulab software) testbeds to support the wider set of technologies and scenarios introduced.

5 DISCUSSION

5.1 Wider uses

Because it relies only on ssh, `nepi-ng` can be used on a large variety of contexts and substrates, even outside of a purely experimental context. In particular, it has been successfully used to write experiments on different other testbeds, including PlanetLab [22] and ORBIT [5]. For instance, a by-product of `nepi-ng` is a command line tool used to run the same command on a large number of nodes, which we routinely employ for the daily operations of PlanetLab Europe. As far as ORBIT is concerned, preliminary attempts have proven encouraging, although the SFTP subsystem did not appear to be enabled between the testbed's gateway and the outside Internet, which hampered file transfers, and thus prevented us from running real scale experiments on this testbed.

5.2 On synchronization

As was described in 3.2, the only synchronization mechanism offered in `nepi-ng` relies on the *requires* relationship between jobs, as well as *forever* jobs combined with sub-schedulers. This is in contrast with other parallel programming techniques, and especially with message-passing techniques, which are for example the primary tool available within OMF for achieving synchronization.

Message-passing admittedly is more powerful than a simple dependency graph, as it allows for instance to create synchronization points anywhere inside a program, while the dependency model can only deal with beginning and end of programs. However in our context, which is not so much compute-oriented as it is IO-oriented, this is not a very meaningful limitation; in particular, remember that networking protocols are in essence precisely about dealing with such fine-grained synchronization by themselves, which means that the programs orchestrated by an experiment control

tool already have built-in mechanisms for fine-grained synchronization.

On the other hand, if one wants to take full advantage of message-passing's flexibility, it is a requirement to have messages delivered right on the nodes where a running program can react on their occurrence, and it is not enough to have messages propagate back only up to the controller program. This implies that some sort of message transport infrastructure be available on the testbed, and this is typically the sort of constraint that we wanted to avoid as per our initial requirement of a light software dependency footprint.

5.3 On abstractions

As stated earlier, we foster a low-level, abstraction-less model for the design of experiments, in contrast for example with OMF's declarative-based mechanisms.

We argue that abstractions convey quite some implicit information. For example, in the `nepi-ng` code illustrated on Figures 9 and 10, the companion shell scripts provide a low-level radio interface initialization sequence that directly manages the Linux driver. Under a formalism like OEDL, a similar effect is obtained through an abstraction, where attributes are set on a Ruby object that represents a node, and that the OMF runtime takes care of implementing.

Because no abstraction layer sits between the `nepi-ng` code and its companion scripts, all the details are immediately accessible to the reader, and we argue that this is an asset in terms of reproducibility.

On Figure 10 for example, we can see that the experiment needs to create a monitoring interface that allow raw packet injection. This mode allows for a fine grained control over the transmitted packets because it doesn't suffer from inconsistencies in the measurements due to the native rate adaptation mechanism. By adopting this approach, we have more consistent measurements of the CSI (channel state information).

Furthermore, proponents of abstractions argue that they make an experiment more reproducible because its description is less dependent on the underlying testbed. This remains in our experience to be established; imagine that we want to compare the results obtained by executing the same experiment on top of two different testbeds. Then, either the physical substrates are similar, in which case the abstractions do not help a lot, or they do exhibit substantial differences, in which case the details of these differences *must be* understood by the experimenter, for a correct interpretation of results.

In conclusion, we are convinced that abstractions in this area can be more harmful than helpful, as they require work to define and implement, do not significantly accelerate the experiment design and implementation cycle, but do substantially obfuscate the actual details of the setup at work during the experiment.

5.4 Impact of asynchronous programming

It is worth outlining that, although the internals of `nepi-ng` heavily rely on the asynchronous programming paradigm of `asyncio`, this can be considered as an implementation detail from an experimenter's point of view; advanced users can take advantage of `asyncio` if need be, but as far as newcomers are concerned, there is no need to be aware of coroutines or event loops.

On the other hand, `asyncio` proves to be extremely efficient for our needs, with the additional benefit of removing the necessity of dealing with concurrent access, i.e., locks and other exclusion mechanisms, thanks to its single-threaded execution model.

6 CONCLUSION

In this paper, we present a tool for controlling and orchestrating network experiments based on a powerful and efficient model that permits a concurrent control of multiple nodes using `asyncio`. This tool allows for a clear separation between the scenario orchestration and the details of the experiments on top of providing the necessary tools that allow code re-usability.

ACKNOWLEDGMENTS

This work and the R2lab testbed are funded by the French ANR through the "Investments for the Future" Program under grants ANR-11-LABX-0031-01 (LABEX UCN@Sophia) and ANR-10-EQPX-0031-01 (EQUIPEX FIT).

REFERENCES

[1] Young-Hwan Kim, Alina Quereilhac, et al. Enabling iterative development and reproducible evaluation of network protocols. *Computer Networks*, 63:238–250, 2014.

[2] Mohamed Naoufal Mahfoudi, Thierry Turletti, et al. Lessons Learned while Trying to Reproduce the OpenRF Experiment. In *ACM SIGCOMM Reproducibility Workshop*, volume 41, pages 21 – 23, LA, USA, August 2017.

[3] Cristian Tala, Luciano Ahumada, et al. Guidelines for the accurate design of empirical studies in wireless networks. In *IEEE TridentCom*, pages 208–222, 2011.

[4] Thierry Rakotoarivelo, Maximilian Ott, et al. OMF: a control and management framework for networking testbeds. *ACM OSR*, 43(4):54–59, January 2010.

[5] Open-Access Research Testbed for Next-Generation Wireless Networks (ORBIT). https://orbit-lab.org.

[6] R2lab Testbed Management Framework. https://github.com/parmentelat/rhubarbe.

[7] D3: Data-driven documents. https://d3js.org/.

[8] Mohamed Naoufal Mahfoudi, Thierry Turletti, et al. ORION: Orientation Estimation Using Commodity Wi-Fi. In *IEEE Workshop on ANLN*,

pages 1033–1038, Paris, France, May 2017.

[9] Why use asyncio ? http://asyncio.readthedocs.io/en/latest/why_asyncio.html.

[10] Curio, a Python library for concurrent I/O and systems programming. https://curio.readthedocs.io/.

[11] Trio: async programming for humans and snake people. http://trio.readthedocs.io/.

[12] asynciojobs: add time dependencies between coroutines. https://asynciojobs.readthedocs.io/.

[13] apssh: asynchronous parallel ssh. https://apssh.readthedocs.io/.

[14] Dipankar Raychaudhuri, Ivan Seskar, et al. Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols. In *IEEE WCNC*, volume 3, pages 1664–1669, 2005.

[15] Mark Berman, Jeffrey S Chase, et al. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[16] Katerina Pechlivanidou, Kostas Katsalis, et al. Nitos testbed: A cloud based wireless experimentation facility. In *ITC 2014*, pages 1–6. IEEE, 2014.

[17] Stefan Bouckaert, Wim Vandenberghe, et al. The w-ilab. t testbed. In *International Conference on Testbeds and Research Infrastructures*, pages 145–154. Springer, 2010.

[18] Brian White, Jay Lepreau, et al. An integrated experimental environment for distributed systems and networks. *ACM OSR*, 36(SI):255–270, 2002.

[19] Platforms for Advanced Wireless Research (PAWR). https://www.advancedwireless.org/.

[20] Cloud Enhanced Open Software Defined Mobile Wireless Testbed for City-Scale Deployment (COSMOS). http://cosmos-lab.org/.

[21] Powder (the Platform for Open Wireless Data-driven Experimental Research). https://powderwireless.net/.

[22] PlanetLab, an open platform for developing, deploying and accessing planetary-scale services. https://planet-lab.eu.

```python
from asynciojobs import Scheduler
from apssh import SshNode, SshJob, RunScript, Pull

GATEWAY = "faraday.inria.fr"
SLICE = "inria_r2lab.tutorial"

# this local script contains the gory details
AUXILIARY_SCRIPT = "./orion.sh"

def send_receive(sendername, receivername, packets, size, period):

    local_trace = "from-{}-to-{}".format(sendername, receivername)

    # the proxy to enter faraday
    r2lab_gateway = SshNode(hostname=GATEWAY, username=SLICE)

    # sender and receiver nodes - reachable through 2-hop ssh connection
    sender = SshNode(gateway=r2lab_gateway, hostname=sendername)
    receiver = SshNode(gateway=r2lab_gateway, hostname=receivername)

    # one initialization job per node
    init_sender = SshJob(
        node=sender,
        command=RunScript(AUXILIARY_SCRIPT, "init-sender", 64, "HT20"))
    init_receiver = SshJob(
        node=receiver,
        command=RunScript(AUXILIARY_SCRIPT, "init-receiver", 64, "HT20"))

    # ditto for actually running the experiment
    run_sender = SshJob(
        node=sender,
        command=RunScript(AUXILIARY_SCRIPT, "run-sender", packets, size, period),
        required=(init_sender, init_receiver),
    )
    run_receiver = SshJob(
        node=receiver,
        commands=[
            RunScript(AUXILIARY_SCRIPT, "run-receiver", packets, size, period),
            Pull(remotepaths='rawdata', localpath=local_trace),
        ],
        required=(init_sender, init_receiver),
    )

    # create an Scheduler object that will orchestrate this scenario
    return Scheduler(init_sender, init_receiver,
                     run_sender, run_receiver,
                     timeout=120)

experiment = send_receive("fit01", "fit02", 1000, 120, 100)
experiment.run()
```

Figure 9: `nepi-ng` code for Figure 4

```
function init-sender() {
    ### 2 arguments are required
    channel=$1; shift       # e.g. 64
    bandwidth=$1; shift     # e.g. HT20

    # unload any wireless driver
    # useful when the experiment is restarted
    modprobe -r iwlwifi mac80211 cfg80211
    # load our driver
    modprobe iwlwifi debug=0x40000

    wlan=$(wait-for-interface-on-driver iwlwifi)

    # create the monitor interface
    iw dev $wlan interface add mon0 type monitor
    # bring it up
    ip link set dev mon0 up
    # init monitor interface
    iw mon0 set channel $channel $bandwidth
}
```

Figure 10: Extract of the auxiliary shell script