The pos Experiment Controller: Reproducible & Portable Network Experiments

Henning Stubbe, Sebastian Gallenmüller, Georg Carle
Chair of Network Architectures and Services, Technical University of Munich, Germany
{stubbe|gallenmu|carle}@net.in.tum.de

***Abstract*—Network experiments are an essential tool to derive knowledge about network behavior. Yet, the experiment landscape is fractured, consisting of many specialized experiment enclaves, i.e., testbeds, with their own toolchain, limiting portability between different testbeds. Such fracturing hinders research, increasing the costs of replicating research across enclaves. This increase in cost also holds for the possible reuse of experiment parts across testbeds. Previous work introduced the pos framework and methodology, reducing the required effort for reproducible experiments. However, currently, this framework is available solely in a limited set of testbeds. In this work, we extend the proposed framework and, thus, build bridges between existing testbed islands. To showcase our approach and its feasibility, we recreate a well-known experiment on multiple testbeds and compare the results. These results indicate repeatability across testbeds. Concurrently, they highlight the flexibility gained through our approach and the need for reproducibility-aware experiment design.**

***Index Terms*—Repeatability, Reproducibility, Replicability, Network Experiment, Portability, plain orchestrating service (pos)**

I. Introduction

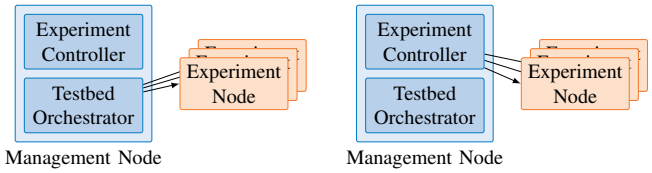
Though the indicating terminology might change over time, the underlying truth does not: the interconnection of applications is ever-increasing, as are the requirements associated with these applications. This truth holds regardless of the application's domain. Whether e-health, autonomous vehicles, or industry, to name a few, all these applications expect specific properties from the network. Often considered properties are throughput or latency. Once it is determined which applications should run, it becomes the network operator's task to provide a network infrastructure to satisfy the implicit demands on the network. A task that requires a thorough understanding of network device behavior and interplay.

Precise and accurate investigation of network device behavior is challenging. On one side, the continued increase in network line rate implies a reduced time budget for processing operations, such as reacting to network changes [1]. Moreover, increased network line rates result in the need for increasing measurement precision. On the other side, networked systems become more complex. While tempting, this complexity cannot be ignored in investigations, as component interactions may cause unwanted side effects. In other words, unwary simplification may result in inaccurate results. To tackle these challenges, researchers use designated test environments, called testbeds, to obtain new insights. Insights, in this context, are generally obtained and described via experiments. The various demands to investigate specific systems have led to an abundance of testbeds with varying scopes [2]. Independent of the chosen testbed, the value of the insights obtained depends on the ability of others to understand. When considering possible approaches to understanding previous work by others, recreating the described experiments is common. To describe the degree of experiment recreatability, ACM defined three levels: repeatability, reproducibility, and replicability [3]. Badges are awarded to papers depending on whether a paper is, e.g., reproducible [4]. A task that is easier described than performed [5]; even with careful investigation, small changes in the environment may result in deviations in the outcome. Despite these obstacles, reproducibility—and, ultimately, replicability—of results is needed to support the claims made.

The plain orchestrating service (pos) methodology [6] was devised to reduce the burden imposed on researchers by the requirement to design reproducible experiments. Through pos' "reproducibility by design", users are encouraged and aided to build only reproducible experiments. In order to achieve this, pos promotes a certain workflow. A prominent property of this workflow is that all experiment nodes are live-booted from a known set of images. I.e., it is discouraged to retain or assume any state on the experiment nodes before or after an experiment. This property allows anyone with access to the same testbed nodes, images, and experiment scripts to reproduce experiments.

Achieving reproducibility is, however, not an experimenter's highest honor. The latter is achieved through replicability. However, to make a pos-based experiment replicable, the experiment results must be reproducible first. I.e., with the same setup, the same experiment outcome must be achievable by a different team. Additionally, replicable experiments must be consistent across different experimental setups. While the pos methodology cannot provide an additional team to perform an experiment, pos' design allows it to be used in a multitude of environments. Thus, to enable researchers to determine if their experiment could achieve replicability, they would need to run their experiments in at least two pos-enabled environments. However, setting up different experiment infrastructures to speculatively improve their experiments again increases the burden on researchers. To mitigate this increased burden, preconfigured pos-supporting testbeds are needed in which researchers may run their experiments.

To address this need, we ported the pos framework to



(a) Step 1: Testbed orchestrator preparing experiment nodes

(b) Step 2: Experiment controller performing experiment

Fig. 1: Two components of the management node

two public testbeds: Chameleon [7] and Cloudlab [8]. By publishing our results, we aim to enable others to re-run their experiments in different testbeds, hence improving their research's quality. Ultimately, research using this approach empowers other groups to show the replicability of the knowledge obtained. This empowerment is aided by the fact that our approach allows anyone to execute an experiment without modification of the experiment code to function on multiple testbeds. In contrast, previous works required either modification of experiment code to interact with different testbeds or, in the case of geni-lib [9], only consider parts of the functionality offered by pos excluding, e.g., controlling the experiment itself. We show the feasibility of our approach by performing a sample experiment in both public testbeds. For comparison, we also conducted this experiment within two pos testbeds: one physical and one virtualized.

The remainder of this work is structured as follows. Section II introduces existing concepts to operate testbeds. Our approach's architecture is described in Section III. After that, Section IV details the resulting implementation. We employ this implementation for the experiments discussed in Section V. Section VI concludes this paper.

II. Background and Related Work

In this section, we investigate different approaches to operating testbeds and executing experiments.

Testbed orchestration vs. experiment control: We identified two components implemented in frameworks to operate testbeds or run experiments, as shown in Figure 1. The first component we call the *testbed orchestrator*. This component is responsible for the management and preparation of the experimental environment. Based on the available resources and their reservation, the orchestrator initializes the specified resources and prepares them to become ready for experimentation. The second component we call the *experiment controller*. The main task of this component is the execution of the experiment workflow in a prepared environment. Typically, the experiment controller expects a particular structure, or even a domain-specific language, to define experiments. Based on this definition, experiments can involve the creation of measurement results, evaluations, or plots.

To make experiments reproducible, the orchestrator needs to create the state of the experiment environment reproducibly. Portability is ensured if an orchestrator creates the environment for the experiment controller across different platforms and the experiment controller performs the experiment according to a defined workflow. The following overview uses the previously introduced concepts to categorize existing approaches for testbed orchestration and experiment control.

A. Testbed Overview

Testbeds come in a wide variety, reflecting the different requirements of research domains. At the same time, testbed operators are conscious of the existence of other testbeds and, hence, strive to join forces for a common goal. These joint works resulted in successful cooperation, such as GENI [10], whose successor FABRIC [11] recently took over and has enabled access to experiment resources across a number of testbeds, including Chameleon and Cloudlab. Also, on the European side, there is ongoing work towards providing not only individual testbeds but also opening access to different testbeds to researchers. One manifestation of this work was FED4FIRE [12]. While this project is now concluded, the idea of a European testbed lives on in its successor, SLICES-RI [13].

a) Existing approaches for testbed orchestration: The geni-lib [9] is a library developed by the GENI initiative that was created to provide an interface to orchestrate different testbeds, such as CloudLab [8]. Testbed users use the API of this library to orchestrate the testbed. The Chameleon testbed [7] uses OpenStack [14] to manage the testbed resources. Users can use the OpenStack APIs to orchestrate this testbed. pos [6] is a testbed framework designed for the creation of reproducible experiments. Using live Linux images ensures that machine state is erased on reboot. In addition, pos users need to fully automate the orchestration process for a reproducible setup. Orchestration is handled via a command line interface (CLI), i.e., any application that can utilize the CLI will be able to use the pos orchestrator.

b) Existing approaches for experiment control: Along with the continuous need for experiments comes the community's striving for tools to simplify their handling. One representative of this is the *Common Workflow Language (CWL)* [15]. This programming language provides scaleable tooling for automated execution of experiments, implementing two standards: 1) details on available tools with their in- and outputs and 2) description of composition rules for known tools. CWL has several implementations using various platforms such as SSH-accessible systems, Kubernetes, AWS, or Azure to run experiments. It is a popular choice for processing data-driven experiments in life sciences [16]. CWL is well-suited to run purely software-based experiments that do not require specialized hardware or network topologies. While also focusing on the execution of experiments, the cOntrol and Management Framework (OMF) [17] approached this topic differently. Rakotoarivelo et al. developed a central controller accepting jobs from experiments to execute on the testbed's hardware. Using a Ruby-based domain-specific language (DSL) to describe experiments, OMF produces experiment outputs available via SQL or HTTP interfaces. NEPI [18] is a platform that provides different backends to

execute experiments: a physical testbed, a network emulator, and a network simulator. The NEPI framework provides an abstraction layer on top of its supported backends utilized by the experiments. Experiments use a Python API to control the experimental workflow. In recent years, Chameleon integrated services for experiment control [7]. The preferred interface for experiment specification are Jupyter notebooks and Chameleon's Jupyter hub; to store these experiment artifacts, researchers are encouraged to use Trovi. Jupyter notebooks aggregate all steps of the experiment workflow. Researchers are free to structure the experiment workflow according to their personal preferences. The pos framework [6] also integrates a component for experiment control. It relies not on a specific API or library for defining experiments. The user writes scripts that are executed on the allocated experiment nodes, and the scripts can be started via the pos CLI from the management node.

c) Benefits of the pos approach: In contrast to the previously mentioned frameworks, pos does not require users to learn and apply APIs or DSLs for orchestrating testbeds or describing experiments. Interaction with pos is kept minimal and handled via a CLI that can be utilized by experimenter-defined scripts. This ensures a high degree of freedom for its users while pos' properties ensure a reproducible execution of experiments.

B. Investigated Testbeds

For this paper, we want to focus on a specific type of testbed, targeting a similar domain and offering off-the-shelf server hardware: Chameleon, CloudLab, and pos. All three offer low-level access to resources, therefore, a high degree of freedom for the experimenter.

1) Chameleon: Operating since 2015, the Chameleon testbed [7] provides an experiment platform for researchers. Designed for a broad set of experiment domains, this testbed provides networked compute on a heterogeneous device set. The available hardware is listed on the testbed's webpage. The desired resources, e.g., networks and machines, can be reserved for a given interval, assuming sufficient availability. Another limiting factor when reserving is that resource allocation requires a virtual currency; each project accepted on the testbed has a limited amount of said currency to spend. During the reserved interval, researchers can configure their resources, e.g., the disk image to be used or how nodes are connected, and conduct their experiments.

A recent addition to Chameleon is Trovi, a service hosting Jupyter-notebook-based experiments and artifacts. The motivation for Trovi and Jupyter notebooks is the attempt to provide a repository for collecting, publishing, and sharing different experiments. Other researchers can easily access Trovi to reproduce existing experiments or to derive their own experiments.

2) Cloudlab: Cloudlab [8] is built on Emulab [19], a purpose-built testbed management software. Designed as a federated testbed, Cloudlab spans across multiple sites in the USA. While sites vary in the kind of offered hardware, there is a tendency towards commercial off-the-shelf servers. Each type of hardware is generally available several times, thus enabling experiments among the same kind of hardware. Similar to Chameleon, connections between machines are implemented via testbed-configurable data center switches. As with servers, the kind of switch differs between sites.

Experiment setups in Cloudlab can be configured either via an XML description or via a Python script, which, based on provided libraries, allows to generate this XML description. Part of this setup description are, e.g., the number of machines required, their type, and disk image, or the intra-experiment links. Additionally, optional information, such as experiment documentation or commands to execute on boot, can be configured this way. The resulting experiment description can be shared with other researchers and simplifies recreating the same experiment setup as needed.

To conduct an experiment, researchers can either search for a suitable time slot by entering their requirements or try to immediately begin experimenting. In the latter case, though, more evolved reservations are more likely to be infeasible, as many resource types are well-utilized. Part of the experiment configuration associated with a reservation is, e.g., the disk image to use on the reserved nodes. Once allocated for the researcher, machines will be configured as instructed; this configuration commonly includes tasks such as deploying SSH keys or mounting a persistent NFS share. Before the allocation terminates, the researcher may, provided sufficient availability, request an extension of their experiment. After the experiment, nodes are reimaged with a default configuration and made inaccessible.

3) pos: Dedicated Deployment: The pos framework consists of two components: a testbed orchestrator and an experiment controller [6]. During the past years, pos-managed testbeds were used for research and teaching at the Technical University of Munich. Only a virtualized testbed has been publicly available with a limited amount of resources.

To organize access to testbed resources between multiple researchers, the pos deployment supports a calendar-based resource reservation and allocation. Researchers can note their intention to use resources during any interval. In their reserved interval, experiments can be conducted by applying the pos methodology. That is, researchers may allocate their reserved nodes, configure arbitrary images to be live-booted, and make use of other features of the pos framework to implement their experiment.

4) pos: Virtualized Deployment: The virtualized deployment of pos considered in this work is similar to the dedicated deployment (cf. Section II-B3). The only difference between virtualized and dedicated deployment is the type of the experiment nodes: the dedicated deployment uses bare-metal servers; in contrast, in the virtualized deployment virtual machines (VMs) are used. Each physical machine present in the dedicated deployment is represented by a VM running; all VMs run on a single physical host. One benefit of this approach is the reduced number of physical machines required to host the testbed and perform experiments [20].

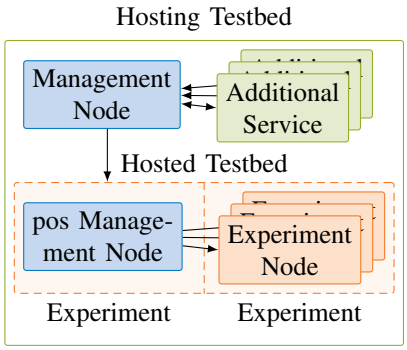


Fig. 2: High-level testbed architecture: the pos management node controls pos-controlled experiment nodes

III. Architecture

The high-level architecture of pos follows a two-layer design, consisting of the testbed orchestrator as the lower layer and the experiment controller running on top (cf. Figure 1). The testbed orchestrator contains all testbed-specific functionality, the experiment controller only utilizes the interface provided by the testbed orchestrator. To achieve portability, only the testbed orchestrator needs to be modified when porting the pos framework to other testbeds; the experiment controller remains unchanged. With the experiment controller left untouched, experiments using the pos experiment controller do not need to be modified. Preserving the experiment scripts reduces the workload for experimenters, as they do not need to adapt their experiment code to new testbeds. At the same time, the reproducibility-by-design is conserved, as testbed orchestration and experiment controller ensure that all properties of the pos methodology are maintained.

Also, apart from portability, another benefit of our approach comes to mind: experiment metadata. To automatically enrich experiment results with information about the experiment environment, e.g., network topology and hardware involved, pos captures and stores said information for each experiment conducted. While performed automatically for scenarios where pos acts as testbed orchestrator and experiment controller, this capability is extensible to other testbeds providing the required information in a machine-readable fashion.

Looking at commonalities between scientific testbeds, we find that such settings feature in particular: 1) a set of entities available for experiments, 2) a management facility to control entities available for experiments, and 3) optional additional services providing supplementing functionality to users, e.g., data storage for experiment results. Based on the premise of limited assumptions, only the former two can be considered when designing the pos experiment controller. However, the use of pos should not prevent the use of features present in the hosting testbed.

A high-level overview of the architecture of the pos framework is depicted in Figure 2. There, a differentiation between the hosting and the hosted testbed is made. The former refers to the testbed providing the physical infrastructure and means to orchestrate it. Moreover, the hosting testbed may, as mentioned, feature additional services surpassing the baseline requirements imposed by our approach. The hosted testbed in Figure 2 refers to the testbed that features are used to perform the actual experiment, i.e., the pos experiment controller. Embedded in the hosting testbed the pos framework provides a subset of the hosting testbed's available functionality, ultimately providing an abstraction of it. Inside this hosted testbed's abstraction, there is, again, a differentiation of resources. On the one hand, the pos experiment controller, itself a managing entity. And, on the other hand, one or multiple experiment nodes, provided by the hosting testbed, are managed through the pos experiment controller. Note that both the managing and the managed experiment nodes may appear indistinguishable in their type to the hosting testbed's management node. Additionally, depending on the hosting testbed's design, nodes used by the hosted testbed may stem from one or multiple experiments of the hosting testbed. For example, the pos experiment controller and experiment nodes managed by it may be part of different experiments on the hosting testbed, cf. Figure 2.

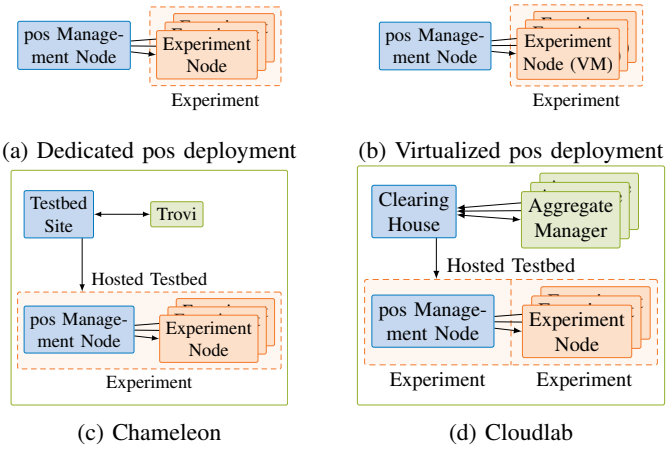
Running the pos framework in a testbed relies on the availability of certain functionality to be exposed via an API, namely: 1) the ability to configure the power state of an experiment node, i.e., turning it off and on again, and 2) the means to control the boot process of experiment nodes, e.g., by allowing network boot or customizing the disk image to boot. With these requirements met, the pos framework can be ported to a testbed.

To summarize, to host the pos experiment controller inside a variety of testbeds, a minimal set of functions, akin to all observed testbeds and required to allow hosting our approach, was determined. Building on this, the pos framework was extended to enable interfacing with a dedicated as well as representative existing testbeds. Details on the implementations will be provided subsequently. The architecture of the pos experiment controller is summarized in Figure 2 and prominently isolates the hosted from the hosting testbed.

IV. Implementation

As indicated previously, we implemented our approach for a selection of testbeds. This selection is based on the experiences reported by Nussbaum [2]; he surveyed available testbeds with a focus on Chameleon, Cloudlab, and Grid'5000 [21]. Given that the latter two support GENI [10], we base our implementation on this API, thus achieving compatibility with both testbeds. Support for Chameleon is achieved by adding support for OpenStack's API to pos. Consequently, the presented approach is not limited to the chosen testbeds.

While the pos methodology is impartial to the language and tools used to conduct experiments, researchers are not. Interactive and visual tools such as Jupyter [22] enjoy great popularity. In a previous work, Demchenko et al. [23] investigate reproducible research and tools suitable for this task. They show how the pos methodology and Jupyter notebooks can be combined to this effect. Building on this, this section shows



(a) Dedicated pos deployment (b) Virtualized pos deployment

(c) Chameleon (d) Cloudlab

Fig. 3: Implementation details depend on the hosting testbed

how Jupyter notebooks can be integrated with the proposed pos experiment controller.

The remainder of this section provides information about implementation considerations for the respective testbeds. As each implementation follows the overarching architecture, this description is focused on particularities related to the individual testbeds. An overview of the different implementation approaches is given in Figure 3.

A. *pos: Dedicated Deployment*

In the dedicated deployment, cf. Figure 3a, the hosting testbed is non-existent and the pos experiment controller is solely responsible for managing the infrastructure. This implies increased control of the experiment infrastructure. At the same time, it also imposes a larger burden on the experimenter. As an example, the hosting testbeds may provide additional recovery features of integration of long-term data storage or user authentication. Such features are not provided by the pos experiment controller. But, as indicated, the benefit of accepting this burden is a tighter control of the infrastructure. For example, some hosting testbeds may only provide virtualized resources, thus limiting the experimenter's control over resources and potentially subjecting their experiments to the actions of others.

While a pos testbed typically relies on a CLI to run experiments, Jupyter notebook-based interaction is also feasible. To run a Jupyter notebook-based experiment on the dedicated pos testbed, the experimenter first needs to provision Jupyter in a Python virtual environment on the testbed's management server. Once the Jupyter notebook is installed and running, the experimenter needs to make this instance accessible, e.g., by forwarding the respective port via SSH. After this setup phase, experimenters can interact with pos either via the CLI or an optional Python library.

B. *Cloudlab*

Neither Cloudlab nor pos are designed with a specific experiment script format in mind. On the contrary, both testbed and methodology are deliberately open in terms of options available to their users. Thus, generally, any approach to deploy the pos experiment controller would be feasible. For consistent deployment across testbeds, we decided to implement a Jupyter notebook-based deployment.

As highlighted in Section II-B2, to start an experiment on Cloudlab for the deployment of the pos experiment controller, a description of the experiment is required. I.e., among others, the number of nodes and their type, their connections, and disk image. For the pos experiment controller, a dedicated experiment was created. Using a Debian bullseye cloud image [24], we make use of Debian's cloud-init [25] capabilities to deploy Jupyter and setup scripts to install the pos experiment controller to the experiment node. Once Cloudlab provisioned the experiment node and cloud-init's configuration has concluded, a Jupyter instance is available. To set up the pos experiment controller, the experimenter executes cells of one of the provisioned setup scripts, a Jupyter notebook. Since the pos experiment controller needs to impersonate the experimenter when interacting with Cloudlab, the experimenter will be asked to provide means to authenticate with Cloudlab. Subsequent steps of this notebook will deploy pos using Ansible. Finally, the deployment instantiates a second Cloudlab experiment. Unless configured otherwise, two nodes are instantiated and integrated into pos. With pos configured and the sample topology in place, the second deployed Jupyter notebook containing the sample experiment can be executed. Figure 3d summarizes this implementation approach.

C. *pos: Virtualized Deployment*

The deployment of pos in the virtualized testbed differs from the dedicated deployment described in Section IV-A. E.g., access to the testbed is granted via a web-shell. Despite that, the implementation of the virtualized and the dedicated deployment do not differ as the features of pos used are unchanged; this similarity is shown in Figure 3b. However, due to the change in access to the testbed, the workflow changes slightly: instead of running a Jupyter server on the management node, we suggest converting the involved notebooks to Python scripts and subsequently executing those.

D. *Chameleon*

Jupyter notebooks [22] are at the center of Chameleon's workflow. While the use of notebooks is not required, their use is encouraged through the availability of APIs, examples, and documentation on their integration. Given that a core concept of the pos experiment controller is its embed-ability into different testbed contexts, its implementation takes this into account. Specifically, the pos experiment controller provides a Python API to control experiments. As a result, the interaction with the controller is well suited for Jupyter notebooks those most prominent execution engine, called kernel, is for Python.

To run an experiment on Chameleon, a Jupyter notebook should be provided on Chameleon's experiment script sharing platform Trovi. Thus, our implementation, as shown in Figure 3c, follows this approach and consists of a Jupyter notebook to interact with Chameleon. The provided Jupyter

notebook attempts to allocate all resources, required for later operation, in Chameleon. I.e., one node featuring the pos experiment controller, the pos management node, and other nodes used to conduct experiments with. E.g., for the experiment discussed later, apart from the pos management node, two experiment nodes are requested to function as device under test and load generator.

After the required resources have been provided by Chameleon, the implementation is otherwise unable to proceed, management and experiment nodes are configured. In particular, the nodes' disk images are set. While the management node is configured to run on Debian bullseye, the experiment nodes are configured to boot iPXE [26].

We selected the former due to its proven reputation as a stable and well-tested platform. Besides, the selection of a PXE booting image is needed to apply the pos methodology. As mentioned, this methodology includes live-booting the experiment node's operating system. To this end, PXE is used to serve the desired operating system to the experiment nodes. Even though numerous devices support PXE booting, its use often requires BIOS reconfiguration. BIOS reconfiguration is not supported on all hosted testbeds we encountered and may suffer from partial or flaky PXE implementations. Therefore, experiment nodes boot the iPXE PXE implementation and, thus, mitigate these issues.

With the management node booted, the provided Jupyter notebook will continue to set up a pos experiment controller on the same. This setup step relies on Ansible [27] and, apart from installing the software itself, ensures preconditions for conducting experiments, such as the availability of bootable images, are met.

Once the setup script concludes, a brief functionality test is performed, to validate the success of the deployment. With this test succeeding, the desired experiment may be performed.

The provided Jupyter notebook concludes with instructions to tear down the setup, once all work has been performed. We encourage releasing experiment resources instead of relying on automatic experiment termination.

The specialization of the high-level architecture depicted in Figure 2 for the implementation in Chameleon is shown in Figure 3c. Of particular note is the interaction with Trovi as an additional service. Trovi hosts the Jupyter notebook describing the experiment. Another specialization is the way each testbed allocates resources and how to map the pos experiments onto the hosting testbed: In Chameleon, for integration of the pos experiment controller, both management and experiment nodes can be part of a single experiment. On Cloudlab, we use two separate experiments for management node and the experiment nodes.

V. Evaluation

To show the usability of the developed pos experiment controller, we revisit a previously conducted pos experiment [6] on the three supported testbeds: our local pos deployment, the Chameleon, and the Cloudlab testbed. The considered experiment investigates the achievable throughput of a device under test (DuT), when subjected to constant bit rate traffic by a load generator (LoadGen). Evaluation of the experiment observations is done thereafter by an evaluator script. Management of the experiment is done by the pos experiment controller. Both, DuT and LoadGen run on a Debian buster live system. Figure 4 depicts the overall experiment setup. There, the test traffic is generated by MoonGen [28] on the load generator.

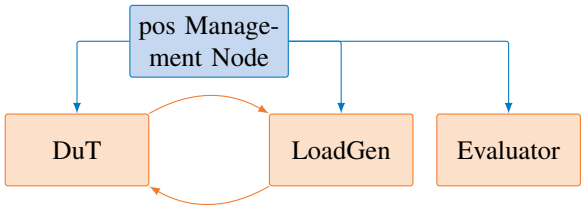


Fig. 4: Experiment setup

The nature of the traffic is varied between different experiment rounds depending on two parameters. The first parameter is the packet size, here, either $64\,\mathrm{B}$ or $1450\,\mathrm{B}$. Ranging from $10\,\mathrm{kpps}$ to $3000\,\mathrm{kpps}$, the second parameter is the requested packet rate. The packet size is chosen to reflect both the smallest and the largest possible packet size transmittable without fragmentation on any of the investigated testbeds. In particular, on Chameleon, the usual MTU of $1500\,\mathrm{B}$ is not available, possibly due to the use of VXLAN as a tunneling protocol between the running virtual machines.

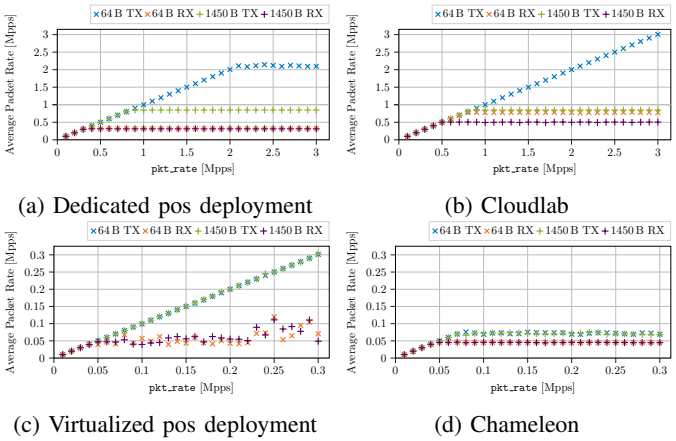
From the load generator, the generated traffic is sent to the DuT. The latter is configured to act as a Linux-based forwarder, i.e., traffic received on the ingress interface is emitted unchanged on the egress interface. Packets from the DuT's egress then arrive again at the load generator.

Experiments are described via and were conducted from Jupyter notebooks [22]. While the pos framework, and, thus also, the pos experiment controller is agnostic to the language used to contain experiment instructions, there is a noticeable preference for this format by, e.g., the Chameleon community.

Figure 5 summarizes the measurement results obtained from executing the same experiment scripts on multiple testbeds. The individual results are discussed in more detail hereafter.

A. *pos: Dedicated Deployment*

For the dedicated deployment, the experiment was conducted on two dedicated machines. A DuT featuring an Intel Xeon E5-2640 v2 running at $2.0\,\mathrm{GHz}$ with $32\,\mathrm{GB}$ of memory. As load generator, we used an Intel Xeon E5-2640 v2 with $16\,\mathrm{GB}$ RAM. Both DuT and LoadGen were equipped with a $10\,\mathrm{Gbps}$ Intel X540-AT2. Results in Figure 5a indicate a linear relationship between requested and received traffic for generated packet rates below $0.5\,\mathrm{Mpps}$. There, for every investigated combination of the parameter, the DuT was able to forward the traffic such that the load generator received any packets sent. For larger packet rates, however, results are more diverse. Here, the load generator's transmitted rate continues to grow linearly until approximately $2\,\mathrm{Mpps}$ and



(a) Dedicated pos deployment (b) Cloudlab

(c) Virtualized pos deployment (d) Chameleon

Fig. 5: Measurement result comparison: same experiment on different testbeds—two bare-metal (top) and two virtualized

1 Mpps for packets of size 64 B and 1450 B, respectively. Afterward, despite requesting higher packet rates, the load generator is unable to comply. In contrast to that, the packet transmitted back from the DuT, independent of the packet size, never surpasses a rate of just below 0.5 Mpps.

B. Cloudlab

The Cloudlab experiment used two machines of type `c220g2` for the LoadGen and DuT. These nodes feature an Intel Xeon E5-2660 v3, running at 2.2 GHz, as well as 160 GB RAM each. Moreover, this node type has dual-port Intel X520 NICs, which were used as experiment interfaces. Similar to the dedicated deployment, the results in Figure 5b show a linear relationship between requested and received traffic. I.e., with the given experiment, no deviation of the received traffic from the requested traffic is noticeable.

C. pos: Virtualized Deployment

In the virtualized setting, the VMs are hosted on a system equipped with two Intel Xeon Silver 4214 12-core CPUs running at 2.2 GHz and with 384 GB RAM. Apart from the pos management node providing both orchestrator and controller, two experiment nodes, provided by virtualized machines, are provisioned on this host system. Each of the experiment nodes is assigned four cores and 7.4 GB of RAM. The virtualized machines are connected via two virtual links provided by the KVM-based hypervisor.

For both investigated packet sizes, the experiment results, as depicted in Figure 5c, look alike. The load generator is, for all investigated packet rates, able to provide the requested load. As a result, the observed transmit rates grow linearly. The packet rate received from the DuT behaves differently. While a linear increase is observable for packet rates up to 0.05 Mpps, for larger packet rates, the received rate remains roughly constant.

D. Chameleon

Depending on the site used, the Chameleon testbed offers both physical and virtualized resources. For this experiments, we used two virtualized `m1.xlarge` instances in KVM@TACC as DuT and LoadGen, respectively. Such an instance amounts to an 8-core Intel Haswell CPU running at 2.3 GHz with 16 GB of memory and VirtIO network devices for the experiment. In contrast to the dedicated deployment, the results in this setting are more diverse.

Looking at the 64 B case, a linear relationship between requested and received traffic is visible for packet rates up to 0.05 Mpps. For larger packet rates, the load generator continues to generate faithfully, up until about 0.075 Mpps. After that point, the generated packet rate remains, roughly, constant. Though the number of generated packets keeps increasing, after the mentioned point at 0.05 Mpps, the LoadGen fails to receive more than approximately 0.05 Mpps for higher rates. A similar behavior can be observed for the 1450 B case.

Regardless of the packet size, the observations are likely due to the same underlying effects. While initially, the DuT is capable of handling every packet, at some rate, this changes. Thus, after this point, 0.05 Mpps, the load generator only receives the subset of traffic the DuT was able to process; this marks the CPU bottleneck. For even higher rates, the network begins to falter. Hence, while capable, the load generator is unable to increase pressure on the DuT. A subsequent test with `iperf3` [29] validates this begin of the network bottleneck at about 520 Mbit/s.

E. Experiment Result Comparison

We conduct a simple throughput measurement on four different testbeds and with four different device configurations. Even though the experiment devices appear comparable in terms of CPU and memory specifications, the results differ. While for both the dedicated pos deployment as well as the Cloudlab testbed, the DuT was able to fulfill the forwarding demand imposed by the load generator, this was not the case in the Chameleon testbed configuration. However, it is important to highlight that this is not a shortcoming of the Chameleon testbed itself. The observed differences in packet generation performance between the Chameleon testbed and the other two testbeds are likely due to the difference in maximum CPU clock speed. While the machines selected in latter cases offer up to 3.3 GHz, the environment in the former is limited to 2.0 GHz. Since the packet generation in this experiment is single threaded, this reduced CPU clock rate also limits the load generator's capability. Additionally, running the experiment virtualized is unlikely to improve performance. Finally, the exact properties of the connection between the load generator and the DuT are unknown. Though MoonGen, due to the DPDK implementation [30], will report the link as 10 Gbps, neither it nor the Linux kernel or OpenStack have information on the link speed. Additional tests with `iperf3` suggest a link bandwidth of about 520 Mbit/s.

VI. Conclusion

The contiguous influx of new network applications with ever-tightening demands calls for a repeated revision of research methods and tooling. One answer to this call is

presented in this paper: the pos experiment controller. By lifting the pos framework and methodology from fixed testbed deployments to a solution that is deployable across different testbeds, researchers are enabled to move from repeatable to likely-replicable experiments. Further, the use of the pos experiment controller reduces researchers' dependency on single testbeds and, therefore, eases collaboration. After presenting the general architecture of the pos experiment controller, cf. Section III, we discussed its implementation in Section IV. To demonstrate how experiments can be conducted using the pos experiment controller, we performed and evaluated a simple measurement in Section V. The code used to conduct the experiments is available online [31].

Availability of the pos experiment controller and its implementation for selected testbeds enables two new challenges to pursue: 1) enhancing the implementation to be deployable in further testbeds, and 2) applying the pos framework to new and existing experiments to afterward evaluate their replicability.

Acknowledgment

This work is partially funded by the German Research Foundation (HyperNIC, grant no. CA595/13-1) and by the European Union's Horizon 2020 research and innovation programme (grant agreement no. SLICES-PP 101079774 and SLICE-SC 101008468). The German Federal Ministry of Education and Research (BMBF) supported our work under the projects 6G-life (16KISK002) and 6G-ANNA (16KISK107). Additional funding was received by the Bavarian Ministry of Economic Affairs, Regional Development and Energy as part of the project 6G Future Lab Bavaria. We would like to express our gratitude for the fruitful exchanges with Simon Schäffner whose support helped shape this work. Additionally, we would like to thank the Cloudlab and Chameleon testbeds supported by the National Science Foundation for this research opportunity.

References

[1] H. Stubbe, S. Gallenmüller, M. Simon, E. Hauser, D. Scholz, and G. Carle, "Keeping Up to Date With P4Runtime: An Analysis of Data Plane Updates on P4 Switches," in *IFIP Networking*, 2023.

[2] L. Nussbaum, "Testbeds Support for Reproducible Research," in *Proceedings of the Reproducibility Workshop*, 2017.

[3] Association for Computing Machinery, "Artifact Review and Badging - Current," 2023. [Online]. Available: https://www.acm.org/publications/policies/artifact-review-and-badging-current

[4] D. Saucez, L. Iannone, and O. Bonaventure, "Evaluating the artifacts of SIGCOMM papers," *ACM SIGCOMM Computer Communication Review*, vol. 49, no. 2, 2019.

[5] N. Zilberman, "An Artifact Evaluation of NDP," *ACM SIGCOMM Computer Communication Review*, vol. 50, no. 2, 2020. [Online]. Available: http://doi.org/10.1145/3402413.3402418

[6] S. Gallenmüller, D. Scholz, H. Stubbe, and G. Carle, "The pos framework: a methodology and toolchain for reproducible network experiments," in *CoNEXT '21*. Association for Computing Machinery, 2021. [Online]. Available: http://doi.org/10.1145/3485983.3494841

[7] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbah, A. Rocha, and J. Stubbs, "Lessons Learned from the Chameleon Testbed," 2020. [Online]. Available: https://www.usenix.org/conference/atc20/presentation/keahey

[8] D. Duplyakin, R. Ricci, A. Maricq, G. Wong, J. Duerig, E. Eide, L. Stoller, M. Hibler, D. Johnson, K. Webb, A. Akella, K. Wang, G. Ricart, L. Landweber, C. Elliott, M. Zink, E. Cecchet, S. Kar, and P. Mishra, "The Design and Operation of CloudLab," in *Proceedings of the USENIX ATC*, 2019.

[9] geni-lib, "Welcome to geni-lib's documentation!" [Online]. Available: https://geni-lib.readthedocs.io/en/latest/

[10] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "GENI: A federated testbed for innovative network experiments," *Computer Networks*, vol. 61, 2014.

[11] I. Baldin, A. Nikolich, J. Griffioen, I. I. S. Monga, K.-C. Wang, T. Lehman, and P. Ruth, "FABRIC: A National-Scale Programmable Experimental Network Infrastructure," *IEEE Internet Computing*, vol. 23, no. 6, 2019.

[12] L. Nussbaum, "An overview of Fed4FIRE testbeds – and beyond?" 2019. [Online]. Available: https://inria.hal.science/hal-02401738

[13] S. Fdida, N. Makris, T. Korakis, R. Bruno, A. Passarella, P. Andreou, B. Belter, C. Crettaz, W. Dabbous, Y. Demchenko, and R. Knopp, "SLICES, a scientific instrument for the networking community," *Computer Communications*, vol. 193, 2022.

[14] "Open Source Cloud Computing Infrastructure." [Online]. Available: https://www.openstack.org/

[15] M. R. Crusoe, S. Abeln, A. Iosup, P. Amstutz, J. Chilton, N. Tijanić, H. Ménager, S. Soiland-Reyes, B. Gavrilović, C. Goble, and T. C. Community, "Methods included: standardizing computational reuse and portability with the Common Workflow Language," *Communications of the ACM*, vol. 65, no. 6, 2022.

[16] CWL, "CWL User Gallery." [Online]. Available: https://www.commonwl.org/gallery/

[17] T. Rakotoarivelo, M. Ott, G. Jourjon, and I. Seskar, "OMF: a control and management framework for networking testbeds," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, 2010.

[18] A. Quereilhac, M. Lacage, C. Freire, T. Turletti, and W. Dabbous, "NEPI: An integration framework for Network Experimentation," in *SoftCOM*, 2011.

[19] F. Hermenier and R. Ricci, "How to Build a Better Testbed: Lessons from a Decade of Network Experiments on Emulab," in *Testbeds and Research Infrastructure. Development of Networks and Communities*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 2012.

[20] S. Gallenmüller, E. Hauser, and G. Carle, "Prototyping Prototyping Facilities: Developing and Bootstrapping Testbeds," in *IFIP Networking*, 2022, iSSN: 1861-2288.

[21] D. Balouek, A. C. Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Perez, F. Quesnel, C. Rohr, and L. Sarzyniec, "Adding Virtualization Capabilities to the Grid'5000 Testbed," in *CLOSER*, 2013.

[22] B. E. Granger and F. Pérez, "Jupyter: Thinking and Storytelling With Code and Data," *Computing in Science & Engineering*, vol. 23, no. 2, 2021.

[23] Y. Demchenko, S. Gallenmüller, S. Fdida, P. Andreou, C. Crettaz, and M. Kirkeng, "Experimental Research Reproducibility and Experiment Workflow Management," in *COMSNETS*, 2023.

[24] "Debian Official Cloud Images." [Online]. Available: https://cloud.debian.org/images/cloud/

[25] Canonical, "cloud-init - The standard for customising cloud instances," 2023. [Online]. Available: https://cloud-init.io/

[26] M. Brown, "iPXE - open source boot firmware," Nov. 2023. [Online]. Available: https://ipxe.org/

[27] Red Hat, "Ansible is Simple IT Automation." [Online]. Available: https://www.ansible.com

[28] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Proceedings of the 2015 IMC*. Association for Computing Machinery, 2015. [Online]. Available: http://doi.org/10.1145/2815675.2815692

[29] J. Dugan, S. Elliott, B. A. Mah, J. Poskanzer, and K. Prabhu, "iPerf - The TCP, UDP and SCTP network bandwidth measurement tool," 2023. [Online]. Available: https://iperf.fr/

[30] I. Dyukov and M. Coquelin, "net/virtio: support Virtio link speed feature · DPDK/dpdk@1357b4b," 2020. [Online]. Available: https://github.com/DPDK/dpdk/commit/1357b4b36246da9dd36d21754234dc306d51f7b4

[31] H. Stubbe, S. Gallenmüller, and G. Carle, "hstubbe/wons2024: The pos Experiment Controller: Reproducible & Portable Network Experiments," Dec. 2023. [Online]. Available: https://github.com/hstubbe/wons2024