NES: Towards lifecycle automation for emulation-based experimentation

Will Fantom[*], Eleanor Davies[*], Charalampos Rotsos[*], Paul Veitch[†], Stephen Cassidy[†], Nicholas Race[*]
[*]School of Computing and Communications, Lancaster University [†]BT Research
[*]{w.fantom, eleanor.davies, c.rotsos, n.race}@lancaster.ac.uk [†]{steve.cassidy, paul.veitch}@bt.com

***Abstract*—Network softwarization has revitalized the interest of the network community towards emulation as an effective mechanism for network experimentation. Relevant platforms automate the deployment of virtual network topologies on a host, providing users the ability to manually run experimental scenarios. Whilst this may suit prototyping, modern development and deployment practices such as CI/CD depend on fully automated testing processes, built around high-level testing APIs and abstracting the challenges involved with synchronizing complex node interaction scenarios. In this paper, we present Network Emulation System (NES): a cloud-native, and highly-parallelizable Network Emulation as a Service (NEaaS) platform designed from the ground up to facilitate codeless experiment specification and to automate network testing workflows in cloud CI/CD environments. We demonstrate that NES offers a 8x speed-up improvement in topology instantiation times in comparison to existing emulation platforms, and its life-cycle model can automate testing processes for complex service configurations using existing CI/CD platforms such as GitHub Actions.**

I. Introduction

Network programmability has drastically changed the way we operate network infrastructures, enabling unprecedented flexibility and programmability. This paradigm shift equally transformed the culture toward network testing and experimentation. Traditional experimentation methodologies rely on two approaches. On the one hand, network practitioners use network simulators and models to study the macroscopic behaviors of network systems [1], with imperfect accuracy and precision. On the other hand, network practitioners using small-scale testbeds with real network hardware to study the behavior of real devices in a configuration scenario. The two approaches remain complementary and offer different trade-offs between experimental scalability and realism.

In recent years, network emulation has emerged as an equally effective mechanism for network experimentation. Relevant platforms use control plane interfaces and data model standards to develop software components that emulate the behavior of real devices, at low link rates, and remain out-of-the-box compatible with unmodified off-the-shelf software. Furthermore, such platforms use the ability of modern OS network stacks to create virtual network topologies with user-defined link-level characteristics. As a result, an experimenter can emulate large network experiments in software and control the trade-off between experimental scalability and realism [2]. Furthermore, the adoption for Network Function Virtualization (NFV) technologies means that network vendors release an increasing number of software Virtual Machine (VM) appliances that replicate the characteristics of hardware devices. Such appliances are compatible with network emulation and can improve the realism of an emulated network experiment [3].

Inspired by these technological advances, the network community has developed several open-source network emulation platforms, like Mininet [4] and GNS3 [5], and open-source initiatives to promote experimental reproducibility [6], [7]. Furthermore, new management paradigms emerge, that exploit the flexibility of network emulation to improve network automation. For example, Network Developer Operations (NetDevOps) and Continuous Integration/Continuous Development (CI/CD) operationalize network testing, deployment, and monitoring, in an effort to achieve automation levels, typically found in the cloud-software domain. The success of NetDevOps depends on the ability to support multi-dimensional testing, and network emulation can realize suitable testing environments, facilitating integration tests for network configurations in a CI/CD pipeline [8]. Similarly, network twins propose the fusion of network models (simulation and emulation) with live network data as a way to predict the behavior and support the management decision processes. Several network vendors use network emulation to implement network twins of mobile and IoT systems [9], [10].

Unfortunately, emulation platforms cannot keep abreast with the increasing need for experimental automation. Firstly, modern CI/CD platforms, like GitHub Actions, require support for cloud-native execution. Experimental execution should support API design flexibility to align with varying CI/CD APIs and embrace packaging and isolation technologies, like containers. Secondly, emulation APIs remain topology-first, and low-level experimentation APIs introduce the need for custom host synchronization mechanisms to coordinate experimental execution. Automating experimental execution, essential for CI/CD, requires a fundamental rethink of experimental models in order to capture experimental stages and interactions, beyond the initial experimental setup. Finally, emulation platform architecture must increase the support for scalability and distributed execution. The time-based cost model in cloud infrastructures means that emulation platforms must exploit parallelization, in order to improve setup and execution times for individual experiments, while ensuring experimental isolation.

This paper argues that network emulation requires a redesign of models and execution environments, in order to meet the scalability and automation requirements of emerging network use cases. To address these challenges, we present an ex-

tensible experiment life-cycle model that allows practitioners to customize experimental execution beyond topology definitions and express component interdependence. To demonstrate the effectiveness of our model, we present Network Emulation System (NES), a cloud-native network emulation platform with support for a wide range of virtualization technologies, including containers, namespaces, and VNF appliances running on KVM. The contributions of this paper are the following:

- We present an extensible lifecycle model for network experiments, covering experimental setup and execution. The proposed model captures component interdependence and can improve execution time for experiments by parallelizing scenario execution.
- We present NES, a cloud-native emulation platform, with support for scalable and heterogeneous experiments. Users can define experimental topologies and scenarios in YAML.
- We demonstrate that the automation features present in NES can improve experimental execution by a significant margin in some use cases and can be integrated with popular CI/CD pipelines, like Github Actions.

In the rest of this paper, we discuss related network emulation research platforms (§ II) and elaborate on two emerging use-cases to identify the limitation of existing platforms (§ III). Furthermore, we present the design of NES(§ IV) and evaluate its performance in comparison to existing emulation platforms, as well as its ability to automate network testing (§ V). Finally, we conclude and discuss future directions (§ VI). NES and its components are open source and can be found at https://github.com/NEaaS.

II. Background

The term network emulation describes a mechanism that allows a user to replicate aspects of a real network infrastructure in real-time on general-purpose CPUs. Originally, network emulation focused on accurately recreating packet-level characteristics on network traffic [11], [12] and replicating topologies with virtual routers in a single host [13]. In recent years, emulation realism has improved, with platforms like Cisco VIRL [3] allowing the inclusion of virtual hosts and VNF appliances in the form of VMs and namespaces and running large topologies entirely in software.

In recent years the increasing improvement in network emulation technologies, offers a plethora of emulation-based platforms to network experimenters, which improve fidelity on several experimental characteristics. Nonetheless, improving the fidelity of specific experimental characteristics influences the platform API and execution model. For example, GNS3, EVE-NG [14], and ContainerLab/VRNetLab [15] set as a key goal to increase support for device heterogeneity. This is achieved through a plugin model that abstracts access to virtualization technologies, and allows the creation of topologies with diverse node types, including Docker containers, Cisco virtual routers, and VNF appliances, in a single topology. Furthermore, users can share drivers for new network node types through open marketplaces, like the GNS3's appliance marketplace [16]. Alternatively, emulators such as Kathata [17] use pre-existing software registries that make a vast selection of software available to the emulator. In Kathara's case, this is Docker container registries such as Docker Hub. However, these platforms primarily focus on topology instantiation and lack support for automated experimental execution. Experimental execution is manual, and users can access virtual hosts via terminal through a GUI interface.

Another key design focus is emulation realism for specific network technologies. Scalion [18], for example, is a Tor emulation platform, built on top of the Shadow network emulator, and allows performance evaluation of Tor overlay applications at scale. Another popular platform in this class is Mininet, offering support for large-scale OpenFlow experimentation. Mininet defines an extensible host model, and several Mininet forks exist that enable support for new host types, like the Docker-based ContainerNet [19], as well as link technologies, like the Mininet-WiFi [20] platform. Mininet offers a Python Topology API, and a low-level execution API allows users to partially automate experimental execution via scripts, that sequentially execute commands on virtual hosts at runtime. The Mininet execution model is limited, experimenters must define custom synchronization mechanisms with reduced reusability. MeDICINE [21], is a network emulator that uses the NFV-MANO data model to realize experimental automation. The topology API is similar to Mininet and users can automate the configuration of virtual hosts using elements in the NFV-MANO host life-cycle model. Yet, although support for the NFV-MANO model certainly improves experimental automation, it lacks some key features, like experimental synchronization across nodes and event ordering.

Another key design goal for emulators is scalability. Relevant efforts utilize network tunneling protocols to distribute topologies across multiple emulation hosts, with Distrinet [22] and Mininet clusters [23] being prime platform examples. Alternatively, other approaches to scalability have seen smarter uses of modern virtualization technologies, as seen with Nokia Service Router Linux (SR Linux) [24], a containerized version of Nokia's routing platform. NES, combines components of both approaches to distribute experimental execution. Of great relevance to NES is Crystalnet [25], an Azure-centric emulation platform for cloud-based large-scale network experimentation. Crystalnet offers an experimental API that decouples topology creation from host configuration. Nonetheless, the specification of a common experimentation specification API is described as future work.

Table I compares the features of NES, the emulation platform presented in this paper, with that of other popular platforms. Whilst all provide the functionality required of network emulators, their specific design goals and supported features define the scenarios in which they offer the greatest benefit.

Recent research efforts explore the integration of network emulation in automated integration pipelines. Network Emulation-based Automated Testing (NEAT) [8] is a network

Emulator	NetNS	Container	VM	Cloud-Native	Automation	Experimental API	Distributed
Mininet [4]	✓	✘	✘	✓	✘	Python	✓
ContainerNet [19]	✓	✓	✘	✓	✘	Python	✓
GNS3 [5]	✓	✓	✓	✘	✘	✘	✓
ContainerLab [26]	✘	✓	✘	✘	✘	✘	✘
NES	✓	✓	✓	✓	✓	REST, WS	✓

TABLE I
A COMPARISON TABLE BETWEEN POPULAR OPEN-SOURCE NETWORK EMULATION PLATFORMS AND NES.

testing platform that uses the Mininet API and leverages containerization to perform automated testing in cloud CI/CD environments. However, the emergence of NEaaS [27] looks to be the most direct way for emulation to adopt the technologies required for the demands of modern workflows.

III. MOTIVATION

Network emulation use cases are plentiful, and include prototyping [28], [29], testing [30], and even teaching [31], [32]. However, as softwarization adoption increases across network infrastructures [2], network emulator architectures offer limited support for the automation required by operational workflows. In this section, we present two emerging use-cases for network emulation and define a set of desired functional requirements for relevant platforms.

DevOps and Continuous Integration/Continuous Delivery (CI/CD): The DevOps paradigm promotes mechanisms for autonomous testing, deployment, and monitoring, predominantly relying on CI/CD pipelines. The effectiveness of DevOps methods to support the rapid rollout of code changes in production with minimal operational disruption in cloud infrastructures has motivated the network community to explore adoption strategies. Network emulation offers a holistic mechanism for integration testing of network configuration in a CI/CD pipeline [8]. The emulator creates a testing environment similar to that in which a service will be deployed, taking into consideration the high degree of configuration interdependence of individual network functions in modern network services. Integration tests can deliver assurances regarding the compatibility between the configuration defined by the infrastructure operator and the component developer/configurator. However, existing emulation platforms depend on human interaction, and the limited experimental automation capabilities remain low-level, experiment-specific, and require development effort. To address these challenges, network emulators require API-driven testing automation, with support for a rich lifecycle model for experiments, beyond topology creation. These capabilities will improve both the completion time of experiments, by parallelizing the execution of independent processes, as well as, enable new cooperative testing approaches, like network test suite standardization.

Emulation-based Network Twins: Network twin technologies combine system models with real-time production data, in order to predict the behavior of an infrastructure, support decision-making, as well as generate realistic data for AI/ML model training. The research community has explored the application of network twins on a range of operational processes, including prognostic maintenance [33] and planning [34]. Network twins predominantly utilize simulation to manage computational scalability, with potential negative consequences on experimental precision. Network emulation has been proposed as a complementary mechanism to model-based network twins [9], [10], offering improved realism by integrating real production software in a network topology. Controlling the trade-offs between experimental scalability and fidelity is essential for network twins. Emulation mechanisms offer built-in mechanism to facilitate vertical scaling, in order to support experiments of varying size. In parallel, API-driven architectures allow easy integration of external components, like monitoring, with emulation components.

IV. DESIGN

In order to support new application domains, we identify three key design goals for network emulation. Firstly, network emulation platforms must adopt cloud-native design and offer cloud execution environments by-design, like Docker and Kubernetes. In addition to creating a container image with the emulator binary and all required resources, an emulator platform should support strong experimental isolation mechanisms and flexible API integration. An experimenter should be able to execute the same experimental files seamlessly on a laptop, as well as on a large multi-cloud environment. Secondly, emulation platforms must rely on new modeling approaches that can equally automate topology creation and experimental execution and can model complex entity interactions. Modeling entity interactions can improve experimental scallability and allow the platform to parallelize experimental execution while ensuring correctness by ordering interaction events. Thirdly, automation should ensure platform extensibility, and support experimental automation for a wide range of node types.

In this section we present the design of NES, a NEaaS platform, and discuss how the platform supports the aforementioned properties.

Cloud-native network emulation: The term Cloud-native describes services that can be decomposed into re-usable components, capable of integrating into any cloud execution environment. Enabling Cloud-native support in network emulation opens up a wide-range of benefits, including easy vertical scale in experimental execution and easy execution of service specifications in a wide range of environments (e.g. personal laptop, test harness, deployment environment). Platforms like Kathara and ContainerLab provide partial cloud-native emulation support, since their node support is limited to Docker containers, and lack support for VM-based node types, like vendor VNF appliances.

The NES architecture, depicted in Figure 1, adopts a client-server architecture, thus allowing easy distribution of network experiments in multi-host environments. NES is split in two components: the NES client and the server-side NES (SNES).

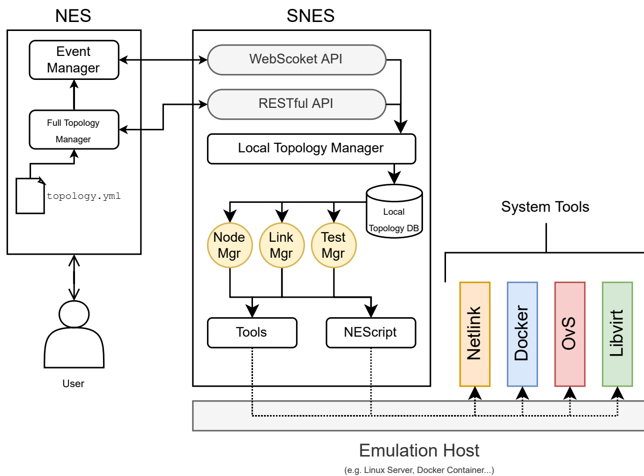

Fig. 1. Architecture of the SNES emulation server application

The NES client holds the user-provided definition of a topology, along with all the assets required to build the topology, such as VM images, and effectively controls the execution of experiments. SNES is distributed as a Docker container, consisting of the NES binary and libraries and runtimes to facilitate OS virtualization, including namespaces, containers, netlink support, using libnl, and KVM VMs. The binary SNES exposes a low-level emulation API, which abstracts virtualization control and allows NES clients to remotely instantiate and manage topologies along with their constituent nodes and links. A user can co-locate the client and the SNES container on a single host, to run a local experiment, or instantiate the complete or a part of the network topology on a remote SNES host.

Figure 2 depicts a sample API endpoint snippet of the SNES RESTful API. The API allows for network nodes and links to be created, destroyed and otherwise managed at any point. SNES offers additionally a websocket-based API for event monitoring. This eliminates the possibility of an unexpected HTTP request timeout for long-running events, like VM node creation. The notion of sensible defaults is also adopted in the API design, allowing for a very minimal configuration in typical use cases, yet allowing for fine-grain control where necessary.

One the key benefits of the client-server NES architecture is the ability to seamlessly scale execution across multiple emulation hosts to support large resource-intensive topologies and improve experimental realism. This ability builds on two mechanisms: offering an extensive experimental management API; and allowing links to exist across multiple SNES instances. Effectively, the SNES API decouples the operational state of a topology element, from its runtime state and allows NES state to distribute across multiple hosts. The NES client stores the operational state of all topology elements, while the runtime details of an element are managed by the SNES instance. For example, a Docker container node will have a topology node ID managed by the client, however, the running

POST **/api/node/create**

Create a node in SNES. Example showing an abridged version of a docker node (nokia SRL) with a post start script.

Body application/json

```
1 "name" : "egNode",
2 "appliance" : "docker",
3 "hooks": {
4 "post-start": "setupScript",
5 ...
6 },
7 "config" {
8 "image": "ghcr.io/nokia/...",
9 ...
```

Fig. 2. Snippet of the create node API endpoint of SNES

container and associated runtime details will be managed by the SNES instance and the local Docker engine. Furthermore, link models in NES allow users to adopt a range of connectivity technologies to interconnect multi-host deployments. As a result, an experiment can use a VXLAN link type to interconnect topologies running on different hosts, on environments that permit direct UDP connectivity. Alternatively, a L3 VPN link in conjunction with the GRE tunneling protocol can establish Ethernet connectivity between hosts where NATs and firewalls impose restrictions. Our NES implementation uses Netlink vEth pairs for regular local links, and provides VXLAN and Wireguard/GRE support for WAN links.

NES Topology Model: Although a challenge in and of itself given the growing heterogeneity of modern networks, extensibility is also deeply tied to scalability. Existing emulation modeling efforts have a fundamental trade-off between ease of use and experimental generality. Platform designs, like Mininet, target a single virtualization technology, in order to allow easier experiment development and automation, whilst platform designs, like GNS3, adopt a plugin model that allows for easy integration of several element types, and frequently sacrifice experimental automation or flexibility.

Towards this challenge, NES adopts the later approach, but enforces a rich set of interactions for plugin realizations. Figure 3 shows the three base plugin specifications in the NES model: tool, link, and appliance. At the root of the NES plugins system are *tools*, compile-time plugins that provide internal access for SNES to interact with system tools, essential for emulation. The NES source code offers built-in tool support for several popular emulation technologies, including Netlink, Docker, OpenVSwitch, and Libvirt. Building atop the *tools*, *appliance* and *link* runtime plugins create and manage the components of a topology, such as the veth link plugin and the Docker container appliance plugin. Existing plugins in NES allow for the creation of highly heterogeneous topologies,

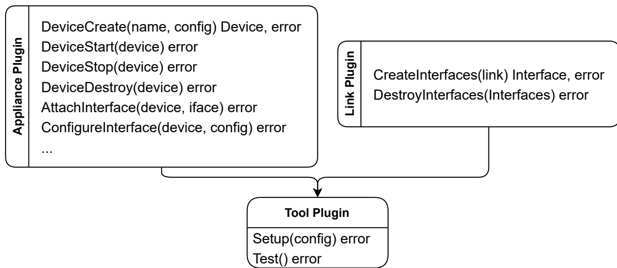

Fig. 3. The NES plugin model consisting of three primary components: *Tools*, *Appliances* and *Links*.

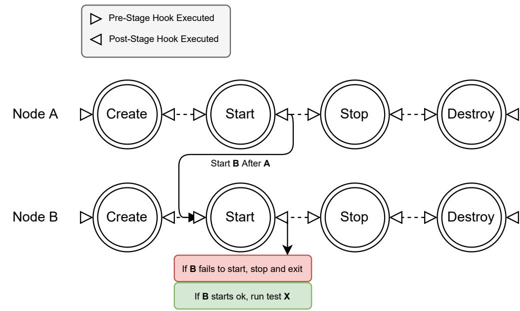

Fig. 4. Lifecycle for automating node hook execution in NES topologies

capable of running emulated hardware devices, such as CISCO IOS devices, and software switches with SDN support, like OpenFlow and P4.

The complexity that can often arise with plugin systems is mitigated in NES by maintaining a simple topology model. An SNES node has a type, which maps into a type of a runtime appliance, and it carries a set of configuration metadata, including node-specific parameters. This slim mode model design allows the inclusion of real hardware devices in a topology, by developing a set of interaction scripts that configure the device and monitor its operation.

Life-Cycle Automation: Directly addressing shortcomings in currently available emulation platforms, NES adopts a life-cycle model for topology components to improve automation support. In doing so, NES adopts a novel topology design model, that differs from ones seen in platforms such as Mininet and GNS3. This model acts as the medium for topology definition, the driver for the API design, and the core approach for automation. Other benefits of a life-cycle model for emulation can also be found, perhaps the most pronounced being parallelism, greatly improving performance of startup times in some topologies.

To start with, nodes in topologies get 4 life-cycle events shown in Figure 4: creation, startup, stop, and destruction. At each one of these a pre and post hook can be provided, executing logic on either the relevant node or the emulation host. These events can be used by an experimenter to control the configuration parameters of an instance at different stages of an experimental run. For example, the relatively simple life-cycle can be leveraged to emulate the impact of updating a network function after a specified portion of the topology has passed the startup stage. This model also allows experimenters to declare node event dependencies, since using this experiment model creates a directed graph between node events. This in turn, provides a global ordering of node events, as well as allowing the platform to detect opportunities for parallelization. Whilst this is useful for automating topologies in general, some use-cases leverage this feature more than others. In a scenario where a topology contains a lightweight container-based firewall application, and VM-based router that has a long instantiation time, so long as the firewall does not depend on the router to have started before starting itself, tests specific to the firewall can be executed whilst the router instantiates. This may allow for failed tests to be identified prior to fully starting the topology. This conforms well with the costing model in cloud CI/CD environments, which typically are time-based thus shorter test execution times reduce CI/CD costs.

Platforms, such as Mininet, offer built-in scripted tests such as PingAll, where all nodes attempt to ping all other nodes. This does rely on the somewhat rigid node model used, as if other node appliances are used, the means to execute a ping from a node may be different than a Netns Exec. Similarly, ContainerNet makes assumptions of VMs in its LibVirt branch, expecting all VMs to have a running SSH daemon. Whilst not an outlandish assumption, this approach bakes some fixed operational requirements in the execution model for the node type, which reduces support for heterogeneity. NES adopts a flexible and slim testing approach, that can support a wide range of execution models to favor extensibility. Specifically, the platform offers NEScript, a scripting framework to describe test actions on a node. Using Go text templating, complicated scripts can be created in just a few lines. Furthermore, each NEScript can specify an interpreter, thus not assuming nodes have a shell and allowing languages like Bash and Python to be used. Finally, this system defines a loose interface for script execution, enabling the script to be executed via a range of mediums including SSH, Docker Engine, Netns Exec, and locally. In parallel, the service depends on a lightweight interface, and adding support for additional services is straightforward. NEScript is open-source and can be found on GitHub (https://github.com/willfantom/nescript).

```
iperf_out=$(iperf3 -J -c 127.0.0.1 | jq -c)
echo "::set-output name=bw type=json::"$iperf_out
```

Listing 1. NEScript Example

```
bw.end.sum_received.bits_per_second >= 10000000000
```

Listing 2. NEScript Expression Example

Extending this, NES explicitly defines *tests* as a component of the topology, each containing its own life-cycle and dependencies. Much like the node life-cycle model, tests can have dependencies against node and link events. For example, a test create event can depend on the start events of specific nodes and links, which allows tests to be executed as soon as the nodes and link have completed their configuration.

Furthermore, in order to improve testing automation for network emulation, NEScript also provides the means to evaluate testing outputs. Using print statements similar to that seen in GitHub Action's *::set-output::*, NEScripts can set string, integers and json output that can then be parsed via the expr [35] module. As a result, tests in NES can be executed as soon as possible, contain complex logic simplified via Go templates, be executed via a medium appropriate for the node appliance, and be evaluated beyond a simple exit code. For example, the short NEScript in listing 1 can ensure that the measured bandwidth of a topology is greater than or equal to 10Gbps with the expression in listing 2. Finally, NEScripts can use the NES template capabilities to support dynamic data setting at runtime, and thus a single NEScript can be reused by multiple nodes even where node specific data is needed.

Other Considerations: Whilst the core design decisions of NES have been discussed, the complete design is not limited to this. The use of Go for the implementation allows both the SNES and NES applications to be distributed as single binaries, a distance away from the complex install process of some other emulators. Whilst tools such as Docker and Libvirt are still expected to be installed on the host machine, the base SNES binary can run basic topologies using Linux bridges and network namespaces, typically no further install procedures on many UNIX systems. Also, the Go compiler can target multiple CPU architectures, including x86 and ARM. In conjunction with the multi-host emulation setup, this allows specific node appliances that run on differing architectures to be used in the same topology, using WAN links for interconnectivity.

Finally, the SNES platform offers a Dockerfile that can generate a Docker container image, especially useful in short-lived cloud environments. The resulting container image contains all tools to create a NES topology, including Docker-in-Docker and Libvirt in Docker. Coming in at less than 300MB, this container image is also an ideal way to get up-and-running quickly with network emulation with NES. Although not intended as a secure isolation method due to the privileges required for interacting with certain system tools, this acts as the most suitable deployment approach for cloud-environments in most circumstances.

V. Evaluation

In this section, we evaluate the performance of the NES emulator. Network emulation platforms depends on system virtualization tools, and thus, their performance is influenced by the technologies it relies on. Regardless, it is not uncommon for literature around network emulation platforms to discuss metrics describing performance characteristics such as topology instantiation times. Our analysis evaluates the scalability of NES and compares the topology instantiation performance between NES and other open-source network emulators. All experiments are executed on a dual-socket DELL server (2xIntel E5-2697, 32G RAM, Ubuntu 22.04) with all the latest stable software versions at the time of writing. Finally, we demonstrate the flexibility of the NES platform by presenting a network test automation use-case for NES on a hosted CI platform.

A. Scalability Evaluation

A key design principle for NES is extensibility, both in terms of supporting the level of network heterogeneity available in today's networks, as well as, to enable user-controller scalability. The appliance plugin system presented in NES allows for many different system tools to be used to create emulated devices. The choice of appliance can be impacted by the user from both a resource perspective, looking to run a topology in given resource constraints, or from a fidelity point of view, looking to create a network providing the highest degree of realism possible. In Figure 5 we show how different node types can impact NES startup times on topologies of varying size. The results highlight that appliances such as Linux bridges and network namespaces have short setup times in comparison even to lightweight Alpine-based Docker images.

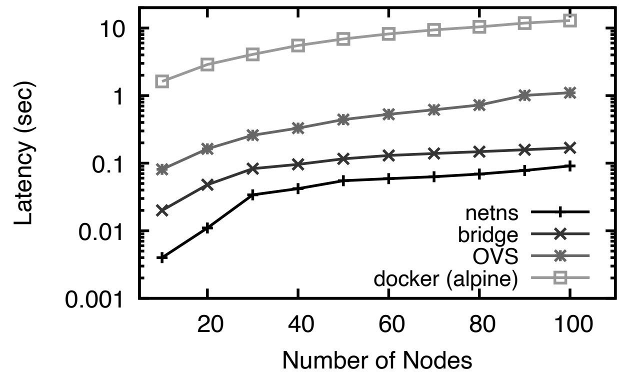

Fig. 5. Comparison of the times required to create a topology consisting of a varying number of node instances using the NES platform.

The results demonstrate the built-in ability of NES to parallelize node creation and improve setup times for large topologies. Notably, simple node types with short setup times, like Netlink-based nodes, gain small benefits from parallelization, and the topology setup time is driven by the synchronization model of the Netlink service. Nonetheless, Docker containers see a much greater benefit from parallelism, and a 100-node topology exhibits an 8x increase on instantiation time, versus a 10-node container topology. These startup time gains are further pronounced for application containers, like the Ryu OpenFlow controller. NES experimenters can mix-and-match any node type typically found in a modern network environment and control the trade-off between precision and scalability.

The built-in parallelization in NES achieves faster topology setup times in comparison to most open-source network emulators, when using the same underlying virtualization technologies. Figure 7 present the topology setup time of a simple topology realized using NES and two popular emulators that support a subset of the same underlying tools as NES: Mininet

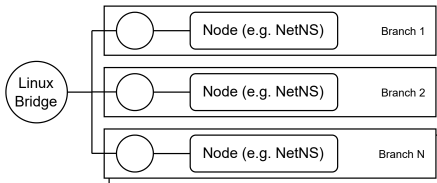

Fig. 6. Star topology used to compare NES startup times with other emulators

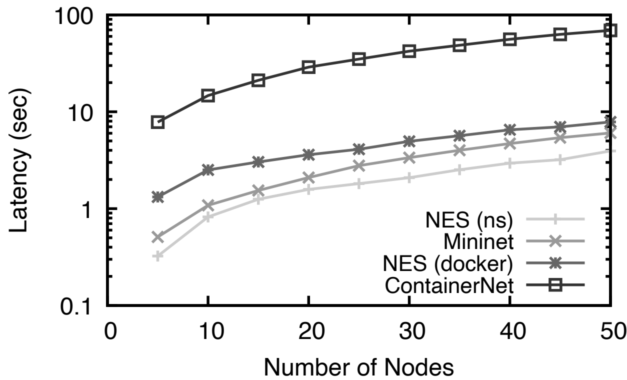

Fig. 7. Star topology startup time with NES, Mininet, and ContainerNet

(namespaces) and ContainerNet (Docker containers). Each experiment creates a star topology, depicted in Figure 6, for a varying number of star branches and Figure 7 presents the total topology creation time for each configuration. As netlink-based nodes are not benefited by the parallelism brought about by the NES lifecycle model, the NES topology instantiation is only marginally faster than Mininet, which uses network namespaces. However, the NES lifecycle model can achieve increased parallelization when instatiating docker-based topologies and outperforms ContainerNet.

```
1 {{ range .Nodes }}
2   {{if (ne .ID 0) }}
3       ping -c 1 {{ (index .Iface 0).IP }}
4   {{end}}
5 {{ end }}
```

Listing 3. A basic 'PingAll' NEScript

Given the extensive feature set of NES, having comparative or improved performance over other popular platforms is significant. It also shows how automation-focused features can be added with little impact on the overall performance, and in some cases actually improves key performance metrics. However, the components of NES that enable this automation can incur performance costs. For example, the scripting system (NEScript) used in NES node hooks and tests has to be compiled whilst the emulation session is underway in order to use dynamic topology data. One specific example is the *PingAll* script, where a node pings all other nodes in the topology. Listing 3 shows what such a NEScript looks like prior to being compiled by the Go text template system. Fortunately, this cost is remarkably low even when dealing with large sets of data. Figure 8 shows the cost of compiling the *PingAll* script with a data set of varying size, both with

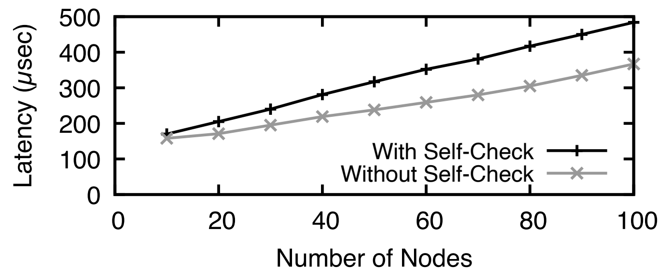

Fig. 8. Impact of dataset size on compile time of 'PingAll' NEScript

and with the self-check if statement. Whilst this can give insight to the compilation cost of each operation, it generally demonstrates a very low cost, even with very large data sets.

B. Testing Automation Evaluation

To show how NES can integrate the benefits of network emulation into autonomous workflows, we present a testing framework for a network service. The network design is from the source code of ContainerNet and consists of a simple Service Function Chain (SFC) containing several clients, a proxy, a firewall, an IDS, an OpenFlow switch and a controller, and a web content server. ContainerNet uses this topology as a tutorial to demonstrate how experimenters can performs a series of connectivity and performance test from the user nodes via the interactive CLIs. In this section, we present a NES configuration that can create a similar topology and perform automated tests in a hosted CI environment.

```
1  nodes:
2      - name: "client1"
3        appliance: "namespace"
4        host: local
5        interfaces:
6          - "h1eth0"
7      - name: "s1"
8        appliance: "ovs"
9  ...
10     - name: "proxy"
11       appliance: "docker"
12       config:
13         image: "ubuntu/squid"
14 ...
15     - name: "client2"
16 ...
17       hooks:
18         post_start: #add httpie,hping3 via nescript
19 links:
20     - type: "veth"
21       interfaces:
22         h1eth0:
23             hw: "72:A6:A3:69:AD:54"
24             ip: "10.0.0.1"
25 ...
```

Listing 4. NES Topology Definition Example

Firstly, to create the topology in NES, we define the nodes and links in the YAML format as shown abridged in Listing 4. However, as many cloud CI environments, particularly hosted ones, do not have nested virtualization support, we adapt the ContainerNet scenario. In order to fully instantiate the topology in GitHub Actions, we replace LibVirt nodes with equivalent Docker container nodes. The resulting Github Action configuration uses the Docker-in-Docker NES image, and

the same environment can run both in the cloud environment and in a local host. Here it is also worth noting, that thanks to the parallelized startup of nodes in NES, the topology instantiation time is reduced compared to ContainerNet, a particularly useful metric in pay-per-minute hosted CI platforms.

With the topology defined, the tests can now be added to the same topology file, coupled with the relevant NEScripts alongside. Defining these tests in NES is trivial, simply requiring a name, a list of nodes to execute the tests from, the NEScript to execute, and a set of expressions that will be used to evaluate the success or failure of the test. For this demonstration, the topology is tested by checking the HTTP response from the web content server from the clients, ensuring HTTP status OK is returned, and testing the firewall by sending packets that should be blocked, in this case ICMP. Listing 5 shows how these tests are added to the topology file, and Listing 6 shows the NEScript that can be used in order to get the HTTP status code of the web server from any client in the topology.

```
1  tests:
2    - name: "curl web server"
3      nodes:
4        - "client1"
5        - "client2"
6      script: # curl web server and store status
7      expressions:
8        - "status == 200"
9    - name: "test firewall"
10     nodes: ["client1"]
11     script: | # send known blocked packets (icmp)
12     expressions:
13       - "packet_loss == 100"
```

Listing 5. NES Tests Definition Example

```
1 status=$(curl --silent \
2   --output /dev/null \
3   --write-out "%{http_code}" \
4   {{ Nodes.Server.IP }} )
5 echo "::set-output name=status type=int::"$status
```

Listing 6. NEScript to capture HTTP status code

Once the topology file and scripts for the experiment have been created, these can be mounted to the NES Docker-in-Docker image in order to be executed. The resulting container is used in a GitHub actions file and pushed to a remote repository. Provided the test is successful, the action will pass. However, if the test fails for any reason, NES outputs a non-zero exit code so most hosted CI platforms will consider the run failed. More information regarding test failure is outputted in the logs. This action can be associated with a repository events such as a push, tag, or release, and during service changes (*e.g.,* modify the OpenFlow controller code or the proxy configuration) the emulation-based test will automatically execute and show the results. This example use case requires a report 7 seconds to instatiate the test topology and run tests on GitHub actions, with NES itself reporting 4.6 seconds to create the topology and 0.95 seconds to run and evaluate the tests. Although introducing more resource-intensive nodes, or even scaling up the size of the topology may increase the required time, the overall execution time is well within the norm for a CI pipeline, especially considering other tests can be run in parallel, such as style checking and unit tests.

VI. Conclusion

Network programmability has drastically evolved the operational capabilities of network infrastructures. Network emulation has emerged as a key technology to improve operational assurances in new automated and autonomic network management paradigms. Unfortunately, existing emulation-based experimentation platforms have not kept abreast with the need for cloud-native, API-driven and automated operational requirements. NES is a network emulation platform, with built-in support for heterogeneous network topologies and experimental automation, beyond topology creation. A key approach to improve experimental automation is the definition of an extensible life-cycle model for experimental components which can capture complex node interdependencies and topology events. Furthermore, NEScript hooks and tests leverage dynamic topology data to allow for well-defined and repeatable automated emulation workflows, all whilst keeping the scope of script capabilities wide. Although NES shares many of the same network virtualization technologies with existing emulation platforms, its ability to parallelize experimental execution provides improved scalability and fast experiment completion times. Changes to the current toolstack such as these are necessary for the growing autonomy of modern networking workflows, from NetDevOps practices to fully autonomous updates via intent-based networking.

In terms of future development, we intend to implement additional plugins that integrate NES with advanced management APIs, including OpenStack and Kubernetes, to support the deployment of complex topologies. This would allow NES to increase topology fidelity and even incorporate some components of production-like infrastructure, where the context is appropriate. Furthermore, to improve experimental efficiency we plan to explore how node, test and link dependencies can be utilized to assist experimental scheduling and resource allocation mechanisms. For example, NES can schedule interconnected nodes on cores that belong to the same CPU socket, that reducing the impact of NUMA architectures on experimental performance and reduce noise neighbor effects. Finally, although using hosted CI environments such as GitHub Actions and GitLab CI can be a quick and easy way to begin adopting modernized software workflows, the platforms often have limits such as job co-location and nested virtualization limits. However, having a multi-tenant NES server would allow NES to operate as its own job manager and be hosted elsewhere, thus bypassing the prior mentioned limitations.

VII. Acknowledgment

The authors gratefully acknowledge the support of the Next Generation Converged Digital Infrastructure (NG-CDI) Prosperity Partnership project funded by UK's EPSRC and British Telecom plc.

REFERENCES

[1] R. Jain, "A survey of network simulation tools: Current status and future developments," https://www.cse.wustl.edu/~jain/cse567-08/ftp/simtools/index.html, 2008.

[2] C. Rotsos, D. King, A. Farshad, J. Bird, L. Fawcett, N. Georgalas, M. Gunkel, K. Shiomoto, A. Wang, A. Mauthe *et al.*, "Network service orchestration standardization: A technology survey," *Computer Standards & Interfaces*, vol. 54, pp. 203–215, 2017.

[3] J. Obstfeld, S. Knight, E. Kern, Q. S. Wang, T. Bryan, and D. Bourque, "VIRL: the virtual internet routing lab," in *SIGCOMM*, 2014, pp. 577–578.

[4] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[5] J. C. Neumann, *The book of GNS3: build virtual network labs using Cisco, Juniper, and more*. No Starch Press, 2015.

[6] L. Saino, C. Cocora, and G. Pavlou, "A toolchain for simplifying network simulation setup," in *SimuToolss*, 2013.

[7] "mininext," https://github.com/USC-NSL/miniNExT/.

[8] W. Fantom, P. Alcock, B. Simms, C. Rotsos, and N. Race, "A NEAT way to test-driven network management," in *IEEE/IFIP NOMS*, 2022, pp. 1–5.

[9] Ericson, "Digital twins catalyst reflections from digital transformation world," https://www.ericsson.com/en/blog/2019/6/digital-twins-catalyst-booth-reflections-from-digital-transformation-world.

[10] Spirent, "Simplifying 5g with the network digital twin," https://federalnewsnetwork.com/wp-content/uploads/2020/08/Spirent_5G_Network_Digital_Twin.pdf.

[11] L. Rizzo, "Dummynet: A simple approach to the evaluation of network protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 27, no. 1, p. 31–41, jan 1997.

[12] S. Hemminger *et al.*, "Network emulation with NetEm," in *Linux conf au*, vol. 5. Citeseer, 2005.

[13] M. Carson and D. Santay, "NIST-Net: A Linux-Based Network Emulation Tool," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, p. 111–126, jul 2003.

[14] "Eve-NG Emulator." [Online]. Available: https://www.eve-ng.net/

[15] "vrnetlab." [Online]. Available: https://github.com/vrnetlab/vrnetlab

[16] "Gns3 appliance marketplace." [Online]. Available: https://gns3.com/marketplace/featured

[17] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Di Battista, "Kathará: A container-based framework for implementing network function virtualization and software defined networks," in *IEEE/IFIP NOMS*, 2018, pp. 1–9.

[18] R. Jansen and N. J. Hopper, "Shadow: Running tor in a box for accurate and efficient experimentation," 2011.

[19] M. Peuster, J. Kampmeyer, and H. Karl, "Containernet 2.0: A rapid prototyping platform for hybrid service function chains," in *IEEE NetSoft*. IEEE, 2018, pp. 335–337.

[20] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-WiFi: Emulating software-defined wireless networks," in *IEEE CNSM*, 2015, pp. 384–389.

[21] M. Peuster, H. Karl, and S. Van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *IEEE NFV-SDN*, 2016, pp. 148–153.

[22] G. Di Lena, A. Tomassilli, D. Saucez, F. Giroire, T. Turletti, and C. Lac, "Distrinet: A mininet implementation for the cloud," *ACM SIGCOMM Computer Communication Review*, vol. 51, no. 1, pp. 2–9, 2021.

[23] B. Lantz and B. O'Connor, "A Mininet-Based Virtual Testbed for Distributed SDN Development," in *ACM SIGCOMM*, 2015, p. 365–366.

[24] Nokia, "Nokia service router linux," Nov 2022. [Online]. Available: https://documentation.nokia.com/srlinux/22-11/SR_Linux_Book_Files/pdf/Product_Overview_22.11.pdf

[25] H. Liu, Y. Zhu, J. Padhye, J. Cao, S. Tallapragada, N. Lopes, A. Rybalchenko, G. Lu, and L. Yuan, "CrystalNet: Faithfully Emulating Large Production Networks," in *ACM SOSP*, October 2017, pp. 599–613.

[26] "containerlab." [Online]. Available: https://github.com/srl-labs/containerlab

[27] J. Lai, J. Tian, K. Zhang, Z. Yang, and D. Jiang, "Network emulation as a service (NEAAS): towards a cloud-based network emulation platform," *Mobile Networks and Applications*, vol. 26, no. 2, pp. 766–780, 2021.

[28] A. Coutinho, F. Greve, C. Prazeres, and J. Cardoso, "Fogbed: A rapid-prototyping emulation environment for fog computing," in *IEEE ICC*, 2018, pp. 1–7.

[29] L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "SCOPE: An open and softwarized prototyping platform for NextG systems," in *19th Annual International Conference on Mobile Systems, Applications, and Services*, 2021, pp. 415–426.

[30] D. Salopek, V. Vasić, M. Zec, M. Mikuc, M. Vašarević, and V. Končar, "A network testbed for commercial telecommunications product testing," in *IEEE SoftCOM*, 2014, pp. 372–377.

[31] T.-Y. Huang, V. Jeyakumar, B. Lantz, N. Feamster, K. Winstein, and A. Sivaraman, "Teaching computer networking with mininet (tutorial)," in *ACM SIGCOMM*, 2014.

[32] M. Pizzonia and M. Rimondini, "Netkit: network emulation for education," *Software: Practice and Experience*, vol. 46, no. 2, pp. 133–165, 2016.

[33] I. Errandonea, S. Beltrán, and S. Arrizabalaga, "Digital twin for maintenance: A literature review," *Computers in Industry*, vol. 123, p. 103316, 2020.

[34] R. Dong, C. She, W. Hardjawana, Y. Li, and B. Vucetic, "Deep Learning for Hybrid 5G Services in Mobile Edge Computing Systems: Learn From a Digital Twin," *IEEE Transactions on Wireless Communications*, vol. 18, no. 10, pp. 4692–4707, 2019.

[35] "Expr." [Online]. Available: http://github.com/antonmedv/expr