A NEAT way to test-driven network management

Will Fantom, Paul Alcock, Ben Simms, Charalampos Rotsos, Nicholas Race
School of Computing and Communications, Lancaster University
{w.fantom, p.alcock1, b.simms, c.rotsos, n.race}@lancaster.ac.uk

Abstract—**The increasing softwarization of network infrastructures introduces an important challenge for network configuration. On the one hand, the growth of the network configuration space as a result of new device types and the expanding inter-dependence of network service components, increases the network configuration complexity. On the other hand, new service deployment architectures lack mechanisms to validate the impact of service configuration on network resilience. Network operators need to adopt new mechanisms to validate and verify network configuration changes, inspired by popular Continuous Integration/Continuous Development (CI/CD) mechanisms. This paper introduces Network Emulation-based Automated Testing (NEAT), an automated testing framework for network configuration. NEAT allows network managers to define network topologies and tests through YAML files and run realistic network topologies and tests. Furthermore, network managers can control the fidelity of their network tests and bound the execution time of testing suites, as well as exploit parallelization of modern servers to speedup test execution.**

Index Terms—**Network Verification, CI/CD, NetDevOps**

I. Introduction

Network Softwarization [1] promotes the adoption of software techniques in building, configuring, and managing network infrastructures, thus simplifying and automating network management. Nonetheless, the adoption of network softwarization is hindered by the increased configuration complexity and the lack of automated and holistic testing. Traditionally, network operators use network testing labs with real network devices to validate network configurations in a controlled environment. In parallel, network administrators inject time-based checks and rollback statements in device configuration, to automatically discard invalid configurations (e.g. disconnection). These human-centric approaches lack support for the dynamism of new network services. Recent research efforts use theoretical network models in conjunction with mathematical tools like model checkers [2], theorem proof [3], or SAT solvers [4] to prove the properties of a network under a specific configuration and a traffic pattern. Such efforts can capture a wide range of network policies and detect several types of network misconfigurations and bugs. Nonetheless, their effectiveness is as good as their model. New middlebox features require modeling updates and the scope of verification is bounded by the scope of the underlying models. In parallel, modeling higher network layers increases complexity drastically.

As an alternate approach, network emulation and simulation platforms offer a healthy dose of generalization by allowing the replication of a wide range of real-world scenarios and network applications and expands the scope of formal method verification. Both approaches can be used synergistically to improve testing coverage. On the one hand, it is easier to add new device types and applications in a network emulation platform, an essential feature to realistically replicate the heterogeneity of network infrastructures. On the other hand, the primary users of such platforms, network managers, tend to be familiar with APIs and tools available to setup testing environments on an emulation platform. Nonetheless, existing network emulation mechanisms lack critical features to support network testing. Firstly, effective network testing requires mechanisms to precisely replicate the device heterogeneity in a network infrastructure. Secondly, existing emulation API cannot capture complex testing scenarios. Finally, there is a need to implement mechanisms to integrate network testing, such as device configuration scripts, with CI/CD pipelines in order to fully exploit the potential of network automation.

In this paper we argue that automated testing is essential for exploiting the benefits of softwarization and programmability. Alongside network verification methods, network operators require emulation platforms with the ability to run automated testing suites. To meet these goals we present NEAT, an automated network testing platform with built-in support for network testing suites and enhanced device emulation realism. In summary, the contributions of this paper are:

- We present how NEAT allows network managers to create network test scenarios with custom network configurations, run asynchronous network tests and collect detailed test logs.
- We describe a set of Mininet extensions offering support for several new devices types, including Docker containers, KVM/XEN and unikernel VMs, as well as CISCO VIRL and JunOS images. In addition, a RESTful API allows asynchronous execution of test scenarios.
- We demonstrate the ability of our system to control trade-offs between testing precision and execution time.

In the remainder of this paper we discuss related network testing efforts (§ II) and present the architecture of our NEAT platform (§ III). Furthermore, we evaluate the scalability of our NEAT implementation (§ IV) and discuss future directions(§ V). NEAT is available under an open-source license at `https://github.com/ng-cdi/neat`.

II. Background

Standardization efforts for network management, like the ETSI MANO model, have motivated the development of automated testing platforms for network services. Peuster *et al.* [5] developed an early benchmarking platform for Service

Function Chains (SFC) targeting the Sonata orchestrator. The system allows users to measure the performance of individual NFV instances using off-the-shelf tools like ping and iperf. Similarly, the H2020 5GTango project developed one of the first CI/CD pipelines for network service descriptors as part of their architecture [6]. Tests are defined using the NFV-MANO model and the SDK uses a Virtual Infrastructure Manager (VIM) to execute them. Existing service testing mechanisms offer partial coverage, designed to test the service components configuration, and lack the ability to validate the service in conjunction with the wider network infrastructure configuration.

Testing standardization has been an active field of exploration for standard definition bodies. Efforts include the ETSI TTCN-3 WG [7], developing a platform-agnostic testing language, the ITU SG11 WG [8], developing protocol compatibility test suites, and the IEEE Future Networks Initiative (FNI) Testbeds WG [9], developing testbed federation standards. Relevant efforts provide an excellent source of use-cases and testing requirement analysis.

A key enabler for testing automation is network emulation. Mininet [10] is a popular emulation platform used widely to execute large-scale OpenFlow experiments. Several Mininet extensions improve the scalability [11], network technology support [12], and integration with a wider range of network control technologies [13]. Unfortunately, the Mininet API is primarily designed to capture network topology properties and simple linear execution scenarios, and lacks built-in supports for asynchronous and parallel operations, and automatic testing. GNS3 [14] is an open source network emulator, providing enhanced network device realism. Users can construct rich topologies with both legacy and programmable device types, like Cisco and Juniper routers. The platform uses predominantly hypervisor-based virtualization to support device model diversity, which reduces scalability for complex experimental scenarios. We believe that network configuration testing requires an emulation platform that combines large-scale topologies with the ability to precisely emulate specific network device types. Fulfilling these two goals with fixed computational resources is a trade-off and topology APIs should allow users to explicitly control the level of fidelity required by their topology and thus optimize the testing environment. Furthermore, existing platforms lack primitives to support test specification, including asynchronous process execution and time-based events.

III. Network Emulation-based Automated Testing

NEAT is a network testing automation system with the goal of furthering network DevOps via enhancing CI/CD workflows. To achieve this, NEAT aims to fulfill the following objectives. Firstly, the test suites should allow users to control the level of realism and support emulated network topologies of varying complexity. Secondly, operators and developers should be able to define repeatable test suites and the data produced from these tests should be easily evaluated by testing policies (e.g. binary output). Finally, all of these

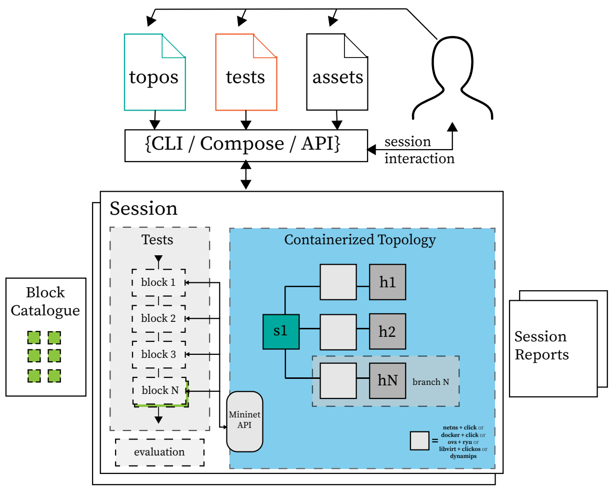

Fig. 1: The NEAT architecture, running a single test scenario against a policy update.

objectives should be capable of local execution by developers, but also allow integration with popular CI/CD systems to support large tests with high resource demands which cannot be satisfied locally. All of these objectives should limit the need for coding to a minimum. In this section, we discuss the design of our network testing architecture and discuss the details of our prototype implementation.

Network Emulation: An essential functionality to enable the NEAT testing capability is an emulation mechanism which can model heterogeneous network devices with high-fidelity, whilst simultaneously having the capacity to run large topologies. In parallel, in order to improve the usability, a language-agnostic interaction API for tests would allow users to seamlessly develop tests using their preferred programming language. Finally, the platform should allow for parallel instance execution to speed up test execution. To meet these goals, NEAT uses the Mininet emulation platform due to its proclivity for extensibility [15], for which we developed a series of extensions.

The first of our extensions increases the support for heterogeneity in Mininet. Specifically, we added a series of new device types to support VM and containers in the form of LibVirt and Docker respectively. This design choice enables support for several pre-built network applications, including Docker-based web-servers, JunOS and Cisco router images, and unikernel appliances. To maintain backwards-compatibility and simplify topology design, our extensions use class inheritance and are compatible with the Mininet topology API. Furthermore, our node type extensions offer testing support for legacy Cisco images via DynaMIPS [16], without the need for real hardware. By supporting an array of network devices on top of the Mininet base, we can use Mininet to manage the whole topology, dynamically attaching device interfaces and connections so that defining a complete testing environment is as simple as including a topology file.

Furthermore, an essential feature for a testing platform is support for parallel program execution and time-based actions, in a language-agnostic way. The existing Mininet interaction

Listing 1: Sample test configuration for a network function test suite

```
topologies:
  - name: clickos-loadbalancer
    topology: cdn-topo
    assets: ["$(pwd)/click-images"]
    libvirt: true
    post_start_script: ./install_routers.sh
blocks:
  - name: h1 connect h4
    variant: ping
    topologies: [clickos-loadbalancer]
    mutables: { sender: h1, target: h4, count: 5}
    expressions:
      - Sent == Received
```

mediums, the CLI and the Python API, unfortunately lack this capability. To address this challenge, we developed a RESTful API to Mininet which exposes topological information and enables asynchronous program execution and log collection. Using the RESTful API, NEAT users can separate test scenarios from the topology and execute them independently.

Finally, the NEAT emulator is designed to run as a Docker container, thus ensuring test isolation. However, as some device types require LibVirt and Xen support we have built support to run certain topology components outside of the Docker context. To isolate parallel tests containing LibVirt nodes, we adopted the following runtime configuration. Firstly, we use a shared folder between the NEAT container and the host, containing required VM images and XML definitions. Secondly, hooks have been added to the VM creation process in LibVirt which detect when a VM is created by a container, in order to expose the VMs interfaces to the container's network namespace. Interface name collision is eliminated by including the parent container's ID.

Automating Testing: The architecture of NEAT is presented in Figure 1. The core of the system is the *session manager* module. A session is an isolated set of topologies and network tests defined by user interaction from either the RESTful API or though a local CLI. Tests are defined using a YAML format and a sample YAML test is depicted in Listing 1. NEAT proceeds to manage the session, providing means for session interaction and result reporting.

A NEAT YAML files contains a series of named topologies and a sequence of testing blocks. A NEAT topology consists of a Mininet topology along with any node assets (e.g. VM image, configuration files). Optionally, a topology can be granted privileges such as access to the host Docker or LibVirt sockets. Once a session begins, NEAT instantiates the topologies in isolated container environments and provides the appropriate access to host resources. With the topologies instantiated, NEAT exposes endpoints to execute arbitrary commands on its nodes. If topologies need more advanced configuration, such as configuration that responds to dynamic content in the topology, NEAT provides hook points for scripts to be executed.

A NEAT test block is the smallest test unit. For example, a ping between 2 nodes is a testing block, whereas, testing multiple packet types via hping could be expressed in a set of blocks. NEAT provides a block catalog of fundamental network testing tools (i.e. ping, iperf) allowing non-expert users to quickly produce tests suites. Each block contains *mutables*, test block input configuration parameters (e.g. testing end-points), and *expressions*. Expressions use a Go expression library [17] and allow users to evaluate test outcomes, using mathematical and logical operators on the results of a testing block and its mutables. The boolean return of an expression dictates the pass/fail state of a block in the session report. Expressions reflect the outcome of a test, and additional logging information can be collected to troubleshoot failures. Finally, a special *script* block can be used to define custom testing mechanisms. A script block allows a user to define custom scripts, execute tests, process results, and evaluate test outcomes. The blocks execute their commands via the RESTful API made available by the emulation layer extensions. Testing blocks in NEAT do however encounter limits. Emulation is not a panacea and certain performance limits appear as network topologies grow in size. As a result, NEAT focuses on blocks that validate non-performance related network properties, like connectivity and correctness.

The configuration system provided by NEAT also provides the means to facilitate some NetDevOps methodologies. As the tests are distinct from the topologies, the 2 components can be sourced separately. Although a VNF developer may be able to produce a simple test suite that ensures that their code integrates as expected into a simple generic network, they could source topologies from the destined network operator to ensure operational correctness in the network environment, prior to roll-out.

IV. Evaluation

In this section, we evaluate the performance and the flexibility of our NEAT architecture. For our experiments, we use a dual-socket DELL server (2xIntel E5-2697, 32G RAM, Ubuntu 20.04), equipped with several virtualization platforms (Xen 4.11.4 Hypervisor, Docker v20.10.7, KVM v1.4.2, and libvirt) and for our KVM-based NEAT tests, we use the ClickOS [18] VM images. Our experiments measure the performance trade-offs when using the different technologies to emulate network nodes in a test and the speed-up gains when parallelizing test execution.

NEAT Scalability Evaluation: To evaluate the scalability of our platform, we run a series of experiments to evaluate the time required to perform a basic connectivity test on a topology of growing size and when using different emulation host types. Our topology consists of a varying number (2, 4, 8, 16) of connected pairs of a "network host", emulated using a network namespace, and a "forwarding appliance". All node pairs are interconnected via a secondary interface on the forwarding appliance in a star topology using a central Open VSwitch instance, operating as a learning switch. We assign a static IP on a /24 subnet to each node connected, and we configure every forwarding appliance to route traffic between any pair of network hosts. Our experiments use five forwarding appliance configurations: a network namespace running a click instance (Mininet), a click docker container (Docker), an

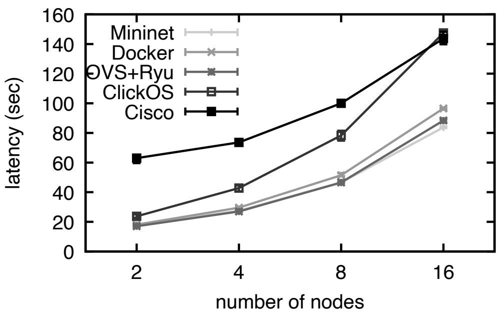

Fig. 2: Median execution time to create a star topology and run a connectivity test.

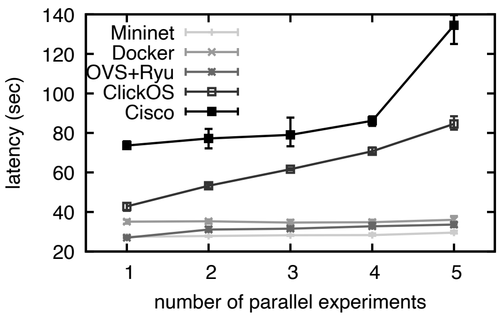

Fig. 3: Median total execution time for parallel NEAT experiments.

Open VSwitch switch controlled by a Ryu router application (OVS+Ryu), a ClickOS unikernel (VM) and an emulated Cisco 7200 router instance running on Dynamips (Cisco). Our connectivity test runs a ping command in every instance and we measure the total time to complete all ping tests.

Figure 2 depicts the min, median, and max (errorbars are not easily distinguishable due to low variance) execution time to setup the star topology and run a connectivity test across five experimental runs. From the results, we highlight that lightweight virtualization technologies like Mininet and containers achieve very good scalability, requiring approximately 90 seconds to run a 16-node topology. Hypervisor-based technologies exhibit more diverse execution profiles. For small topologies, performance is comparable to light-weight virtualization, however the boot overheads increase significantly the total execution time for larger topologies. In contrast, DynaMIPS topologies with a few nodes exhibit comparably higher execution times due to the impact of software emulation on boot-time. Nonetheless, these boot-up latencies are masked by other topology setup latencies, as more DynaMIPS nodes are added in the topology. For example, a 16-node topology exhibits the same execution time as a VM-based topology of equal node count. It is worth highlighting that DynaMIPS utilizes a core at 50% during idle. Finally, the NEAT platform is lightweight and has negligible impact on execution times (<2 sec).

To evaluate the ability of NEAT to parallelize test execution, we revisit the previous experimental setup and change the number of parallel tests running each time. Specifically, we use a star topology with four nodes and run a varying number of topologies (from one to five), in parallel. The selection of the topology size and the number of parallel topologies ensures that our platform has sufficient CPU resources to cope with the size of the experiment, without any node experiencing resource starvation. Figure 3 reports the min, median, and max execution time across five experimental runs. Lightweight virtualization again demonstrates the best scalability, with the execution time remaining unaffected by the parallel execution of experiments. Due to the impact of full host virtualization however, the ClickOS and Cisco topologies experience a slight increase in execution times. This can be attributed to the increased resource requirements to boot hosts, which slow down the topology creation. Nonetheless, execution time is increased by only 50%-60% when running five parallel testbeds. Although the DynaMIPS/Cisco host type exhibits the worst scalability properties, it allows users to precisely emulate the network behavior and users can control the trade-offs between test fidelity and scalability by adopting appropriately their topology files.

V. Conclusion

A major challenge to adopt network softwarization in production is the lack of mechanisms to holistically test and validate network configuration. This becomes increasingly important as legacy devices will need to co-exist and interoperate with new networking technologies in the near future, during the transition of production systems to new operational models. We introduce NEAT, an automated network testing architecture with a network emulation platform designed to capture the diversity of modern infrastructures. NEAT's Mininet-based emulation layer can emulate a wide range of network device types, including VMs and unikernels, Linux namespaces, and OvS bridges, effectively capturing the heterogeneity of network infrastructures. Adding support for virtualization in Mininet negatively affects its scalability capabilities. Nonetheless, NEAT allows network managers to control the trade-off in the realism and the execution time of a test by using VMs for network elements that require increased emulation precision. In parallel, strong isolation between test runs allows NEAT to parallelize and achieve significant test execution speed-ups.

As for future work, we aim to improve the scalability of the emulation layer in order to support larger, more complex heterogeneous topologies whilst keeping test execution times within time limits reasonable for the CI/CD context. Also, NEAT's goals for a codeless test definition are not fully realized, as topology files still depend on a Python Mininet file. A future update to NEAT could contain a YAML layer that enables Mininet topologies definition in a user friendly way.

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

[2] S. Prabhu, K. Y. Chou, A. Kheradmand, B. Godfrey, and M. Caesar, "Plankton: Scalable network configuration verification through model checking," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 953–967.

[3] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky, "VeriCon: Towards verifying controller programs in software-defined networks," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 282–293.

[4] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 699–718.

[5] M. Peuster and H. Karl, "Profile your chains, not functions: Automated network service profiling in devops environments," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2017, pp. 1–6.

[6] A. Nuriddinov, W. Tavernier, D. Colle, M. Pickavet, M. Peustery, and S. Schneidery, "Reproducible functional tests for multi-scale network services," in *2019 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov 2019, pp. 1–6.

[7] ETSI, "Testing and test control notation version 3 (TTCN-3)," http://www.ttcn-3.org/.

[8] International Telecommunication Union, "SG11: Signalling requirements, protocols, test specifications and combating counterfeit products," https://www.itu.int/en/ITU-T/studygroups/2017-2020/11/Pages/default.aspx.

[9] IEEE, "Future networks tested working group," https://futurenetworks.ieee.org/testbeds.

[10] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: Association for Computing Machinery, 2010.

[11] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "MaxiNet: Distributed emulation of software-defined networks," in *2014 IFIP Networking Conference*, 2014, pp. 1–9.

[12] R. R. Fontes, S. Afzal, S. H. B. Brito, M. A. S. Santos, and C. E. Rothenberg, "Mininet-WiFi: Emulating software-defined wireless networks," in *Proceedings of the 2015 11th International Conference on Network and Service Management (CNSM)*, ser. CNSM '15. USA: IEEE Computer Society, 2015, p. 384–389.

[13] 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 *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, 2018, pp. 1–9.

[14] R. Emiliano and M. Antunes, "Automatic network configuration in virtualized environment using GNS3," in *2015 10th International Conference on Computer Science Education (ICCSE)*, 2015, pp. 25–30.

[15] M. Peuster, J. Kampmeyer, and H. Karl, "Containernet 2.0: A rapid prototyping platform for hybrid service function chains," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, June 2018, pp. 335–337.

[16] "Dynamips (cisco router emulator)," https://github.com/GNS3/dynamips, 2021.

[17] antonmedv, "expr," https://github.com/antonmedv/expr, 2021.

[18] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the art of network function virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 459–473.