NeST: Network Stack Tester

Shanthanu S Rai, Narayan G, Dhanasekhar M, Leslie Monis, Mohit P. Tahiliani
Wireless Information Networking Group (WiNG), Department of Computer Science and Engineering
National Institute of Technology Karnataka, Surathkal, Mangalore, India, 575025.
(shanthanu.s.rai9, gnarayang, sekhardhana529, lesliemonis)@gmail.com, tahiliani@nitk.edu.in

ABSTRACT

Linux network namespaces are a cost-effective and scalable alternative to physical systems for the design and experimental evaluation of network protocols. These evaluations are required for a practical understanding of how various networking algorithms would perform in the real world. However, manually setting up testbeds and obtaining results in the desired format using network namespaces can be quite cumbersome and error-prone. Although writing scripts could make these tasks easier, it becomes tedious and impractical if the network under consideration is large and complex. In this paper, we propose a python based package called NeST (Network Stack Tester) to perform tests for different congestion control algorithms and queue disciplines. It uses Linux network namespaces and provides APIs to create complex emulated networks, run tests and extract the statistics using *iproute2* and *netperf* in a single python script. We validate the results obtained from NeST against those obtained from a physical testbed, and a virtual testbed setup manually by using network namespaces. The experiments with NeST are easy to reproduce because it is a wrapper around the existing tools and does not introduce new system dependencies.

CCS CONCEPTS

• **Networks** → **Network experimentation**.

KEYWORDS

Network Namespaces, Network Stack, Emulation

ACM Reference Format:
Shanthanu S Rai, Narayan G, Dhanasekhar M, Leslie Monis, Mohit P. Tahiliani. 2020. NeST: Network Stack Tester. In *Applied Networking Research Workshop (ANRW '20), July 27–30, 2020, Online (Meetecho), Spain.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3404868.3406670

ANRW '20, July 27–30, 2020, Online (Meetecho), Spain

ACM ISBN 978-1-4503-8039-3/20/07…$15.00
https://doi.org/10.1145/3404868.3406670

1 INTRODUCTION

Performing network experiments and studying network protocol behavior is a non-trivial process. The most commonly used approach to perform network experiments is to setup a physical testbed that closely represents the desired network. But it is expensive even for relatively small networks and more importantly, not scalable. Linux network namespaces are a suitable alternative to physical testbeds because we can quickly setup a lightweight emulation testbed for the experimental evaluation. Setting up a network environment in a single system using network namespaces enhances the reproducibility aspects, and minimizes the maintenance and cost overhead. Nevertheless, building a complex topology using network namespaces can be a tedious process, starting with the creation of a large number of namespaces, establishing connections, generating different traffic patterns and extracting per-node or per-flow statistics for evaluation.

Some tools provide pre-defined experiments and intuitive means to collect the statistics (e.g., Flexible Network Tester [5]), whereas others help to quickly build a virtual network topology within the system (e.g., Mininet [7]). However, integrating the benefits of these tools is not straightforward because it requires an in-depth understanding of the APIs of both the tools and additional efforts to verify that they interact correctly. Tools that collectively provide support for topology creation and conducting experiments (e.g., transperf) are limited in terms of topology customization.

In this work, we propose NeST[1], an open source Python package that simplifies the process of performing networking experiments by using Linux network namespaces. It provides easy-to-use APIs to build a virtual network topology, run experiments and collect statistics in different data/graphical formats. Multiple instances of the same network topology can co-exist, and different experiments can be run in parallel on every instance. It provides APIs to use *netperf* to generate network traffic between any pair of nodes. Lastly, NeST provides APIs to extract the statistics of required attributes from the experiment. All these features ensure that using NeST requires fewer pre-requisites (e.g., prior knowledge of network namespaces to setup virtual testbeds or experience about setting up physical testbeds) compared to Flexible Network Tester (Flent) and other tools.

[1] https://nitk-nest.github.io/

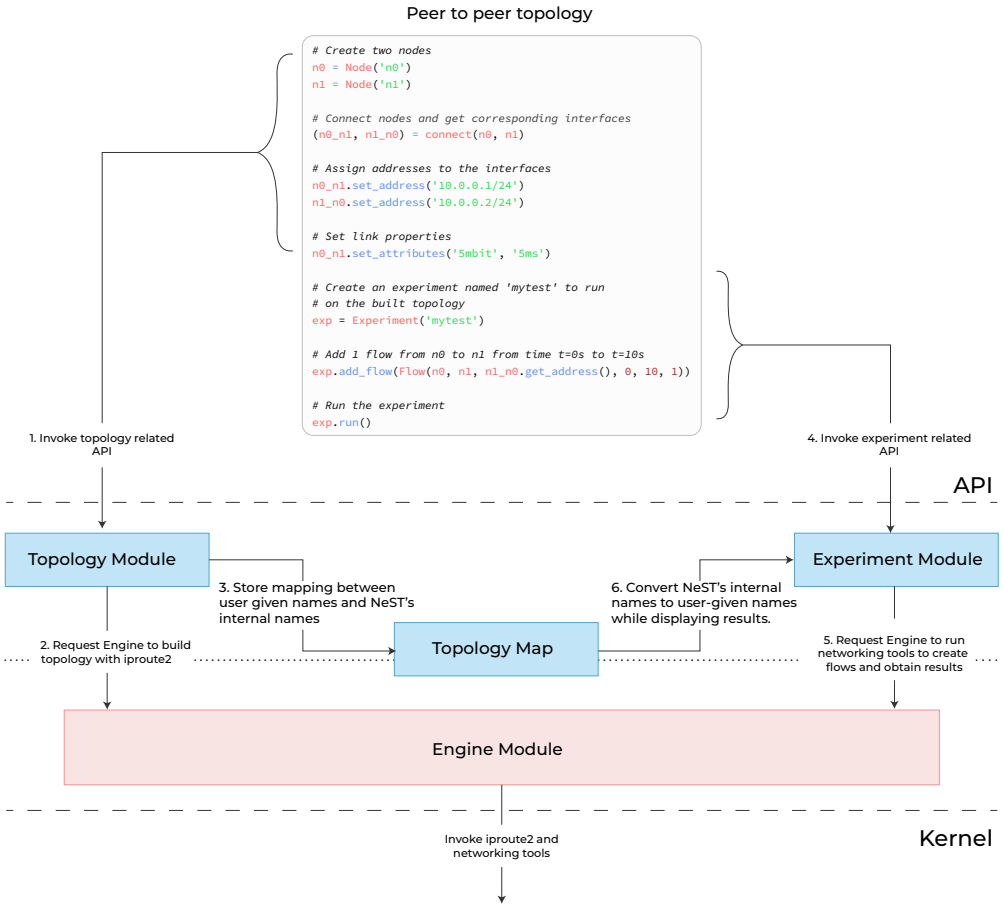


Figure 1: NeST Architecture

2 NeST

NeST uses Linux network namespaces to setup a virtual network with hosts and routers. Internally, routers are the same as hosts but with extra configurations essential for routing. Since network namespaces virtualize only the network stack, setting up a complex topology is fairly quick, and collecting network statistics is simpler because network namespaces are isolated from each other. NeST provides APIs to configure the testbed, use the required tools to generate flows and collect results. It internally assigns a unique identifier to the topology created to avoid naming collisions, and maintains a map between its internal names and the user defined names.

NeST is similar to Mininet in terms of topology creation. Mininet is more suitable for Software Defined Networking rather than as a tool for testing network protocols. It provides an abstraction to tools commonly used for testing the network configuration like *ping*, *pingall*, and *iperf* (for testing bandwidth), but unlike NeST, does not provide an abstraction for advanced traffic control options using *tc*, running *netperf* for traffic generation and *ss* for collecting socket statistics.

2.1 Architecture

Figure 1 shows the architecture of NeST. It is a minimal wrapper around *iproute2* and networking tools. Internally, it is a collection of modules (*Engine*, *Topology* and *Experiment*), where each module provides a specific service. This enforces a modular design, which makes it relatively easy to make changes in the existing code, implement new features and add other traffic generators, such as *iperf* and *httperf*.

Engine module runs *iproute2* and *netperf* commands. It provides a set of low level APIs for other modules. Hence, *Engine* is the closest module to the kernel. *Topology* module creates the topology by using virtual nodes and interfaces. Virtual nodes are setup using network namespaces and virtual interfaces are setup by using tools in *iproute2*. These features are provided to the user as an object-oriented API.

NeST internally auto-generates unique identifiers for the topology created by the user. This makes sure that no two namespaces have the same name, and also allows multiple network topologies with entities having the same user given names to co-exist in the system. The mapping between user

given names and NeST's internal names are maintained in a *Topology Map*, which is a JSON-like data structure.

Experiment module provides APIs to generate traffic and extract the statistics from nodes and interfaces. While presenting results to the user, *Topology Map* is used to convert NeST's internal names to user given names. *Experiment* is a diverse module because it: (i) handles invoking networking tools at different times, (ii) parses output of various tools and compiles them, and (iii) dumps the results as a timestamped JSON file and plots the requested performance parameters.

2.2 API demonstration

Figure 1 shows the code snippet for a short lived flow in a simple peer to peer topology. It demonstrates the APIs provided by NeST. First, two nodes are created, namely *n0* and *n1*. This internally creates two network namespaces with unique names auto-generated by NeST. The mapping between user-given names and NeST names is maintained in *Topology Map*. Subsequently, the nodes are connected by using the *connect* API. This API creates a pair of virtual Ethernet (*veth*) devices and assigns them to *n0* and *n1*. Next, IP addresses are assigned to these interfaces. Afterwards, the *set_attributes* API is used to set the properties of the link from *n0* and *n1*. Note, the link attributes are being assigned to the interface at *n0*. The peer to peer topology is now setup.

Next, an experiment called 'mytest' is setup and a flow is added in it. The *Flow* API takes in arguments for *source node*, *destination node*, *destination address* to uniquely identify itself. Additionally, the *start time*, *stop time* and *number of flows* are specified as arguments. Note, just a single flow is added in this case. But NeST allows to add multiple flows between any two pairs of nodes. Finally, *exp.run()* is invoked to run the experiment. After the experiment is complete, statistics such as throughput, *cwnd*, RTT and others are provided to the user as timestamped JSON dumps and plots.

2.3 Scope and Limitations

Scope. Currently, NeST supports the performance tests for different congestion control algorithms and queue disciplines (*qdiscs*). At the endpoints, it allows the user to configure TCP congestion control algorithm (e.g., Reno, BBR), initial congestion window (*initcwnd*) and other TCP parameters that can be found in `/proc/sys/net/ipv4/`. Besides, NeST allows the user to set link parameters such as bandwidth and delay, and configure the *qdisc* at the intermediate nodes.

NeST internally uses *netperf* to generate network traffic, and parses the statistics reported by it to plot goodput. It uses *ss* to obtain socket statistics like the congestion window (*cwnd*), Round Trip Time (RTT), Slow Start Threshold (*ssthresh*), per-flow throughput, *pacing rate* and *delivery rate* of TCP flows. NeST uses *tc* to configure the bandwidth and delay of the links, configure queuing disciplines, buffer sizes and filters on router interfaces, and extract *qdisc* specific statistics like queue length and queue delay.

Due to the modular architecture, it is relatively easy to integrate additional tools with NeST. This can be accomplished by adding functions to run the low level, tool specific commands in the *Engine* module, the required high level APIs to use the tool in the *Topology* and *Experiment* modules.

Limitations. NeST does not use hardware NICs, so the results that are obtained from NeST do not show the effects of hardware level optimizations that are done in modern NICs. These optimizations include offloading certain tasks to the NIC that are otherwise done at the software level. The lack of hardware queues also prevents NeST from showing the effects of Byte Queue Limit (BQL), a mechanism to limit the size (in bytes) of the hardware transmission queues in a NIC.

Since NeST only uses the network stack provided by the Linux kernel, it is currently not possible to emulate a test environment that contains different implementations of network stacks such as those in BSD, Windows or stacks that run atop userspace packet processing libraries such as Netmap [9] and Data Plane Development Kit (DPDK) [1]

NeST has limited support for debugging the test environment, which includes checking link status using *ping*. Interfaces to external tools such as *tcpdump* and *traceroute* [6] shall be added in a future release.

3 VALIDATION

We perform two experiments using Flent to validate the correctness of the results obtained from NeST. The first experiment is performed on a simple topology configured in a physical testbed and the second, on a complex topology which is configured manually by using network namespaces. For NeST, experiments are run on network namespaces only.

3.1 Experiment 1

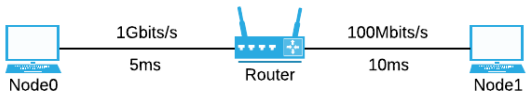
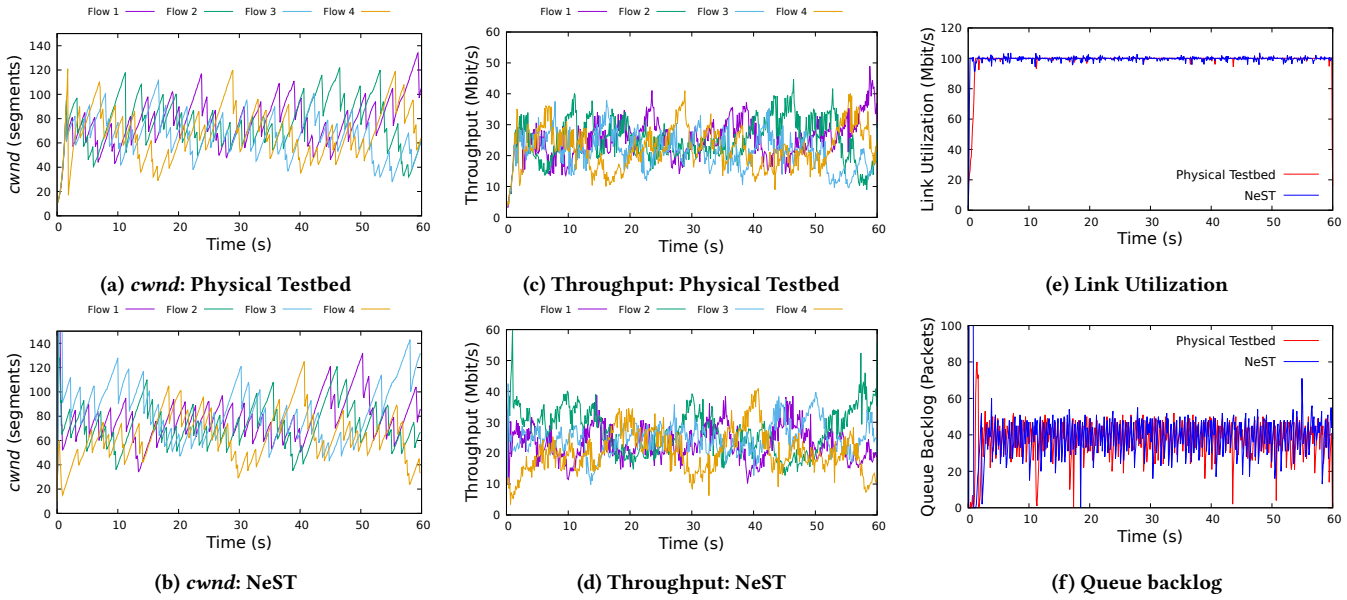


Figure 2: Simple topology for NeST validation

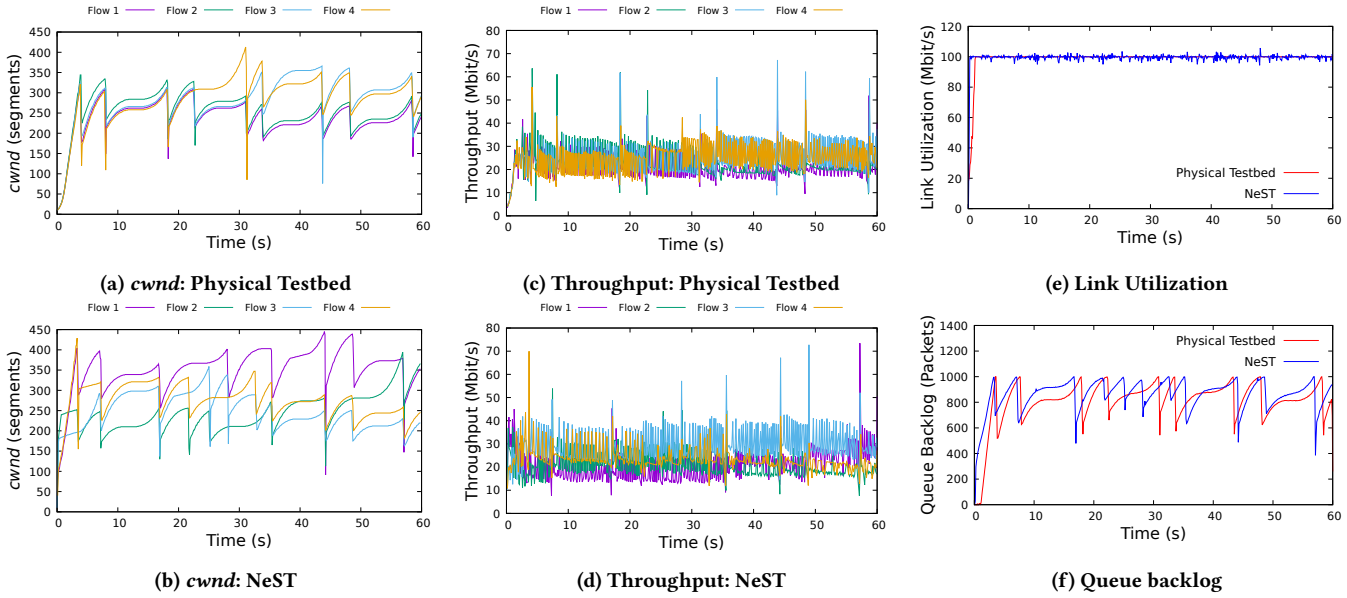
The topology shown in Fig. 2 is setup in a physical testbed, and in NeST. Flent's *tcp_4up* test (4 TCP upload flows from Node0 to Node1) is used to run the experiment on the physical testbed, and four flows with similar characteristics are setup in NeST. CUBIC TCP [4], the default congestion control algorithm in Linux, is used for the experiments. The router uses Controlled Delay (CoDel) [8] *qdisc* with a *target queue delay* of 5ms, *interval* of 100ms and *queue limit* of 1000 packets. *netem* and *htb* are used for latency and bandwidth management. The experiment is repeated by using



(a) *cwnd*: Physical Testbed (c) Throughput: Physical Testbed (e) Link Utilization

(b) *cwnd*: NeST (d) Throughput: NeST (f) Queue backlog

Figure 3: Experiment 1 – Results with CoDel *qdisc*



(a) *cwnd*: Physical Testbed (c) Throughput: Physical Testbed (e) Link Utilization

(b) *cwnd*: NeST (d) Throughput: NeST (f) Queue backlog

Figure 4: Experiment 1 – Results with FIFO *qdisc*

FIFO *qdisc* at the router. Figures 3 and 4 show that the results obtained from physical testbed and NeST correlate well. The *cwnd* and throughput plots in Figures 3(a-d) and 4(a-d), respectively, validate the functionality of NeST in terms of endpoint behavior, and the link utilization and queue backlog plots in Figures 3(e-f) and 4(e-f), respectively, validate the functionality of NesT in terms of the network performance. CoDel *qdisc* keeps the backlog under control (Fig. 3f) while keeping the bottleneck bandwidth fully occupied (Fig. 3e), whereas FIFO *qdisc* has a high backlog (Note, the range of Y-axis is different for Fig. 3f and Fig. 4f) due to passive queue management. When CoDel is used, the throughput obtained for every flow is fair because CoDel has a better queue control, and avoids bulk packet losses due to congestion. FIFO, on the other hand, has poor queue control and leads to bulk packet losses when congestion occurs. This results in unfair allocation of the bottleneck bandwidth among four flows. As expected, the bottleneck link is fully utilized with FIFO *qdisc*.

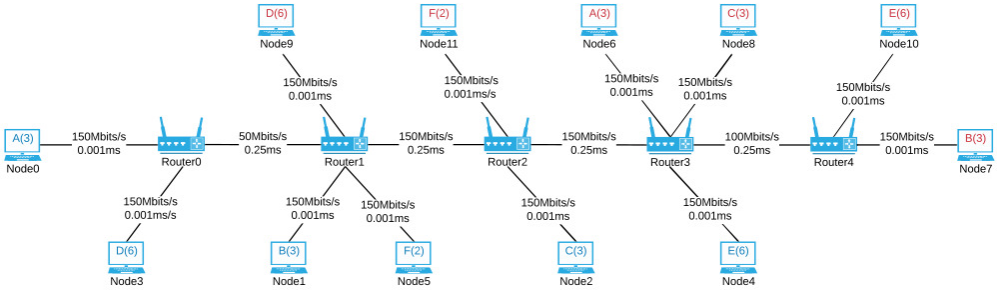
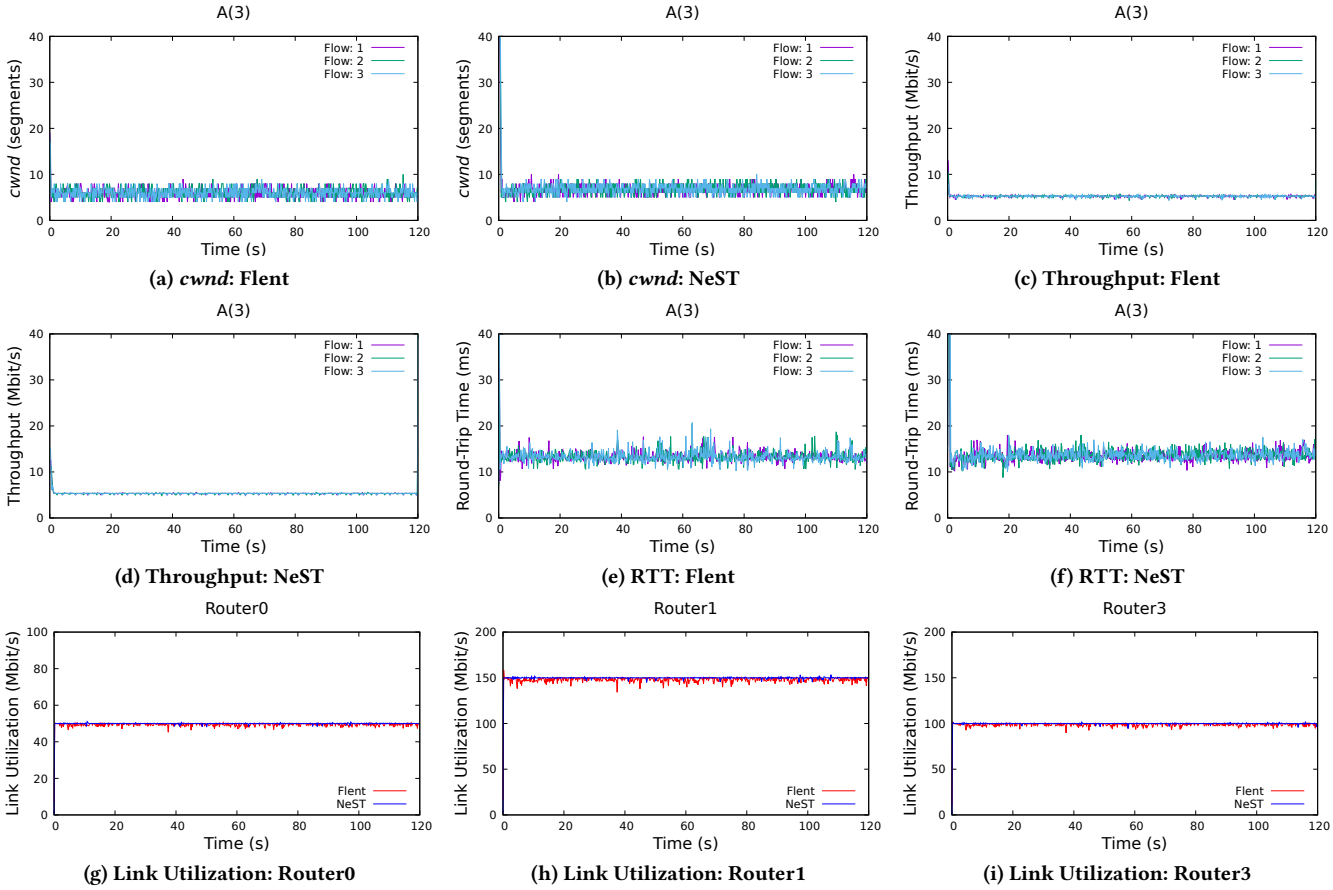


Figure 5: Complex topology for NeST validation



(a) *cwnd*: Flent **(b) *cwnd*: NeST** **(c) Throughput: Flent**

(d) Throughput: NeST **(e) RTT: Flent** **(f) RTT: NeST**

(g) Link Utilization: Router0 **(h) Link Utilization: Router1** **(i) Link Utilization: Router3**

Figure 6: Experiment 2 – Results with FQ-CoDel *qdisc*

3.2 Experiment 2

The complex topology [3] in Fig. 5 is manually setup by using network namespaces and by using NeST. It comprises of 12 nodes and 5 routers, with all links connecting routers being the bottleneck links. Every sender uses CUBIC TCP, and the routers use Flow Queue CoDel (FQ-CoDel), the default *qdisc* in Linux. CoDel's configuration in FQ-CoDel is same as described in Experiment 1. We run three flows from *Node0* to *Node6*, *Node1* to *Node7*, *Node2* to *Node8*, six flows from *Node3* to *Node9*, *Node4* to *Node10* and two flows from *Node5* to *Node11* for 120 seconds. In Fig. 5, A(3) on *Node0* and *Node6* indicates that 3 TCP flows are run from *Node0* to *Node6*.

Figure 6 shows the results for this experiment. Due to limited space, we present the *cwnd*, throughput and RTT only for flows *A*, because they traverse via multiple congested routers. We present the link utilization for Router 0, 1 and 3

to cover links of different bandwidth. Other results of this experiment are available on the link given in the footnote.

These results validate the functionality of NeST in a complex network setup with FQ-CoDel *qdisc*. This experiment was run on a system having Arch Linux with kernel v5.6, Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz, 4 cores (8 threads) and 8 GB of RAM. The RAM utilization for 120 seconds of the experiment with NeST was 3.12%, hence it does not have high system requirements. NeST only logs all the commands that are used to build the topology and does not interfere with the experiments. During the experiment, the intermediate stats collected from network interfaces are stored in RAM, and these stats are written onto a file only after the entire experiment has completed. Hence, disk I/O is not a limiting factor while running experiments with NeST.

The source code and the steps to reproduce both experiments described in this section are openly available[2]. NeST requires *iproute2* and Python 3 packages to be pre-installed.

4 RELATED WORK

Flent [5] provides pre-defined tests, and simplifies the collection and analysis of data from the experiments. It can be used with physical systems and network namespaces. It does not provide tools to build a topology, hence requires a pre-setup physical or virtual network topology. This is sufficient for small scale testbeds or even medium sized testbeds with a relatively simple configuration, but has limited scalability for experiments requiring large testbeds. NeST allows for emulating large scale testbeds using network namespaces.

Netesto [2] is similar to Flent but has fewer number of tests and provides less data on *qdisc* statistics. TEACUP [10] on the other hand concentrates on testbed configuration and data collections, but requires a physical topology setup beforehand. Transperf[3] is developed by Google specifically for testing transport protocol performance. It expects a pre-setup topology with physical systems (multi-server mode) but handles creation of network namespaces when run in single server mode. However, it offers very little flexibility in terms of the testbed configuration.

Mininet [7] provides an API and a command line interface to create and manage a network of virtual hosts, switches, controllers and links. But Mininet provides limited options with respect to *qdisc* configuration on hosts. Running experimental tests using Mininet will either need additional configuration to use a testing tool or the user may have to run complex commands (to setup flows, collect stats, etc.) in the Mininet script which could become tedious.

Initially, we attempted to build a wrapper for Flent and incorporate topology setup using network namespaces. But Flent is designed to run from a single node or controller in a network, and uses SSH to control and get statistics from remote machines. Using SSH to communicate between namespaces is an overhead. Although it can be achieved with minor tweaks, testing many-to-many flows and collecting statistics is a difficult process with Flent. Moreover, Flent comes with a set of pre-configured tests, most of which can be given test parameters to vary the number of flows, target hosts, etc, but running any custom designed test requires adding a new test configuration. Mininet, on the other hand, provides nice APIs for setting up topology using network namespaces, but lacks support for advanced *tc*, which is very important for our use cases. Since NeST uses components of *iproute2* for collecting statistics and advanced *tc*, it's more efficient to set up network namespaces using *iproute2* than using Mininet.

5 CONCLUSIONS AND FUTURE WORK

In this paper, we proposed NeST, a python package to simplify the process of performing experimental evaluations of congestion control algorithms and queue disciplines, and ensure reproducibility. NeST allows users to easily setup and configure scalable testbeds, and run tests in a single python script. We plan to enhance the capabilities of NeST in future by providing additional intuitive APIs to make the process of integrating other tools simpler, and adding features like automatic routing, support for experiments other than TCP and *qdiscs*, support for a hybrid (namespace-physical system) configuration, a GUI extension for interactive plotting and an extensive debugging support via tools like *tcpdump*.

[2]https://gitlab.com/nitk-nest/nest-anrw20

[3]https://github.com/google/transperf/

REFERENCES

[1] 2014. *Data Plane Development Kit.* https://www.dpdk.org/

[2] Lawrence Brakmo. 2017. Netesto, a Network Testing Toolkit. NetDev.

[3] Luis Guijarro, José Vidal, and J. Martinez. 1999. Guaranteeing Fairness and Protection in ABR by means of Fair Queueing. 22 – 31.

[4] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS operating systems review* 42, 5 (2008), 64–74.

[5] Toke Høiland-Jørgensen. 2015. Flent: The FLExible Network Tester. In *The 11th Swedish National Computer Networking Workshop (SNCNW)*. 120–125.

[6] V Jacobson. 1989. Traceroute ftp://ftp. ee. lbl. gov/traceroute. tar. gz.

[7] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-Fefined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. 1–6.

[8] K Nichols, V Jacobson, A McGregor, and J Iyengar. 2018. RFC 8289: Controlled Delay Active Queue Management.

[9] Luigi Rizzo. 2012. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (USENIX ATC'12)*. USENIX Association, USA, 9.

[10] Sebastian Zander and Grenville Armitage. 2015. *TEACUP v1.0 – A System for Automated TCP Testbed Experiments.* Technical Report 150529A. Melbourne, Australia.