NestedNet: A Container-based Prototyping Tool for Hierarchical Software Defined Networks

Xuzhi Zhang, Narendra Prabhu and Russell Tessier
Department of Electrical and Computer Engineering, University of Massachusetts, Amherst, MA 01003

***Abstract*—Emulators for software-defined networks (SDNs) are important prototyping tools in validating network hardware performance under a broad range of topologies and parameters. Modern SDNs typically contain hierarchical collections of network nodes, each with interconnected compute devices. These devices often have widely varying compute environments making accurate emulation using discrete processes in a virtual machine (VM) difficult. In this paper, we describe NestedNet, a new container-based prototyping environment for hierarchical SDN systems. Each network node is represented as a Docker container. The node internals inside the container are implemented as nested Docker containers interconnected via an Open vSwitch. Unlike previous emulators, the execution of each heterogeneous component in a network node can be accurately performed using native code within the target execution environment. To demonstrate the flexibility of our rapid prototyping system, we emulate a mobile ad hoc network (MANET) topology of twelve interconnected nodes of five components each and evaluate its performance using throughput and latency metrics. Emulated throughput values of up to 32 Gbps per link are achieved.**

I. Introduction

As the spectrum of distributed applications and network communication protocols that rely on software defined networking (SDN) has expanded, the need for low-cost, accurate rapid prototyping environments has grown. These networks typically contain complex compute nodes that can often be characterized as SDN subnetworks with separate control and data planes. This hierarchical network model allows for design flexibility and more straightforward system management. However, increased design complexity makes it difficult to accurately verify system and network performance without expensive hardware prototyping. Two approaches to assist in verification are software-only emulation and emulation supported with some testbed hardware components. Large scale testbeds, such as GENI [1], provide an effective SDN prototyping infrastructure for some node topologies but may not be appropriate for mobile-area networks (MANETs). For distributed applications, topology changes due to mobility pose further challenges.

A significant concern with network emulators is emulation fidelity. Parallel execution environments, protocols, and interfaces must be accurately modeled in the prototyping environment to allow for faithful latency, throughput, and packet congestion analysis. In a hierarchical network, interconnected systems (nodes), such as communicating drones or vehicles, may use one set of SDN protocols, while intra-node components (e.g. inside a drone) may have different requirements. Intra-node components may vary widely in terms of compute power, network interfaces, and SDN controls. The situation is further complicated by multiple operating systems, file systems, and libraries required by each node component. The accurate emulation of these components requires compute environment isolation.

978-1-7281-8466-1/20/$31.00

In this paper, we present NestedNet, a new network prototyping environment based on nested Docker [2] containers. Each network node is represented as a Docker container. Inside each node container is a network of nested Docker containers that represent node components. Docker provides process, network, system and environment isolation using namespaces [3] and cgroups [4]. A container has its own file system, networking interfaces, and a separate process tree. Docker components can be interconnected using an SDN-compatible Open vSwitch (OVS) [5]. By using containers, NestedNet allows for the dynamic creation and assembly of networking components and nodes and for the replacement of nodes with physical hardware. Our system is customized to allow for Virtual Ethernet (Veth) [6] tunnels between network namespaces both within and between Docker-based nodes.

To demonstrate the abilities of NestedNet, we show the real-time emulation of twelve mobile nodes implemented as a MANET. We examine the throughput and latency of channels for both intra-node and inter-node communication and Docker container startup time. A visibility-based inter-node link update mechanism is supported to allow for dynamically-changing interconnect. The framework is shown to be similar in terms of memory size and compute resources versus a competing prototyping environment, Containernet [7], that does not support nesting. Our prototyping system has run successfully in PC and server-based Linux environments.

The remainder of the paper is organized as follows: Section II describes prototyping background in this area. Section III describes the design and implementation of NestedNet using a nested container approach. Section IV describes our experimental approach for NestedNet evaluation and Section V presents experimental results. Section VI concludes the paper and offers directions for future work.

II. Related Work

A. SDN Rapid Prototyping Environments

Popular SDN emulators such as EMANE [8] and Mininet [9] use distributed virtual machines (VMs) or processes to represent whole networking nodes and their interactions. EMANE creates a Linux container for each network node and nodes

are interconnected at the media access control (MAC) layer. EstiNet [10] is an OpenFlow-based network emulator that uses kernel re-entry to enable the execution of unmodified application code. The performance of this approach for isolated node emulation can be limiting. CORE [11] uses FreeBSD network stack virtualization to extend physical networks for planning, testing and development.

Mininet is a popular network prototyping tool that has been adopted for SDN emulation and development. It supports process-based virtualization to represent a broad range of network components. Each network node is assigned separate network interfaces and routing tables. Although flexible, Mininet requires file system, memory, and processor sharing for intra-node emulation which can affect node performance and accuracy, especially under high inter-node traffic rates. Mininet also cannot isolate namespaces within nodes. This issue limits the representation of hierarchical SDNs in Mininet.

B. Container-based Network Emulators

Containers, such as Docker, have grown significantly in popularity over the past few years. These virtualization environments are considerably more resource efficient than their VM counterparts. Containers support low-overhead virtualization for a native operating system (OS), including Linux, reduced startup time, low overhead, and isolated namespaces, libraries, and operating parameters (maximum CPU and memory usage). To address these issues, our emulation system advances recent container-based emulators, including several based on Docker.

To et al. [12] created Dockemu to emulate general-purpose wired and wireless networks, although support for intra-node isolation was not provided. Containernet [13] allows for individual nodes to be represented in Docker containers. Intra-node computation is represented as a series of processes. The entire emulator can also be hosted in a Docker container to support portability. Containernet 2.0 [7] extends this environment to include service function chains. CrystalNet [14] is targeted at cloud networks. It implements each network device as a VM or container and provides a network management environment. CrystalNet incorporates a nested approach to an extent by leveraging containers inside a VM. This allows for the same network management environment in each VM. However, the VMs consume significant resources. vSDNEmul [15] has many similarities to our emulator, except for support for nested containers. Each Docker container can include a host, client, or switch and inter-node connections are made via Veth tunnels. The filesystem of each container is completely independent. These previous emulators do not address the issue of nested containers for the isolation of processes in each node container.

Although previous SDN emulators have not used nested containers to represent network hierarchy, they have been used in a similar fashion for microservices. Amaral et al. [16] architected a framework for software systems based on a large number of small services. Each service is represented as a Docker container. For our system, unmodified application code can be executed within each intra-node container, allowing for realistic operating conditions and network interfaces. Each container executes a series of user-mode processes.

III. NestedNet- Design and Architecture

In this section, we present the architecture of NestedNet and, as an example, describe its use in emulating a MANET with a time-varying topology. Our modular, hierarchical approach allows for flexibility in prototyping a wide variety of network systems with nodes representing drones, satellites, or other systems. Dockerfiles are used to ease the definition and modeling of applications, binaries, and tools.

A. NestedNet Architecture

Our SDN prototyping platform can best be illustrated through an example that includes multiple nodes with interconnected components. Although we use a MANET as an example, NestedNet can be used effectively for fixed topology networks as well. In Figure 1, each mobile *node* contains five Docker containers that represent internal *components* and an Open vSwitch that is used for interconnection. The five components are encapsulated within the node which is implemented as a separate Docker container. Nodes in this context could represent an entire drone, satellite, or vehicle, while individual components represent networked subsystems, such as a control system or a physical interface, within the node.

Both nested (component-level) and node-level Docker containers have their own namespaces. Memory and CPU usage within each container can be controlled through parameterization. Each node in the example represents an interconnected system with a global network access brain (GNAB) and multiple communication interfaces (iPHYs). The *Internal Open vSwitch* acts as a virtual bridge that connects intra-node components and performs switching and routing tasks. Inter-node connectivity is provided by the *Main Open vSwitch Bridge*. This component is only used in emulation to configure inter-node connections between iPHYs. In the physical system, the iPHYs connect directly using physical media and do not require the bridge.

For this example, we assume a MANET node has time-varying visibility with other nodes. For emulation, node connectivity must be periodically re-evaluated, and interconnections between nodes must be updated. In a physical system, this process would be distributed among the nodes. For emulation, a controller is used to dynamically update the MANET topology and interconnections. A *communication assistant (CA)* is responsible for addressing time-varying changes in node visibility by configuring the addition and deletion of links between nodes. The nodes in our emulated system have limited visibility as defined by a visibility graph provided by a user. The CA is responsible for parsing the graph and distributing visibility information to the GNAB in each node. The GNABs then determine inter-node connections and inform the CA. The CA subsequently configures the *Main Open vSwitch Bridge* to make or remove inter-node connections.

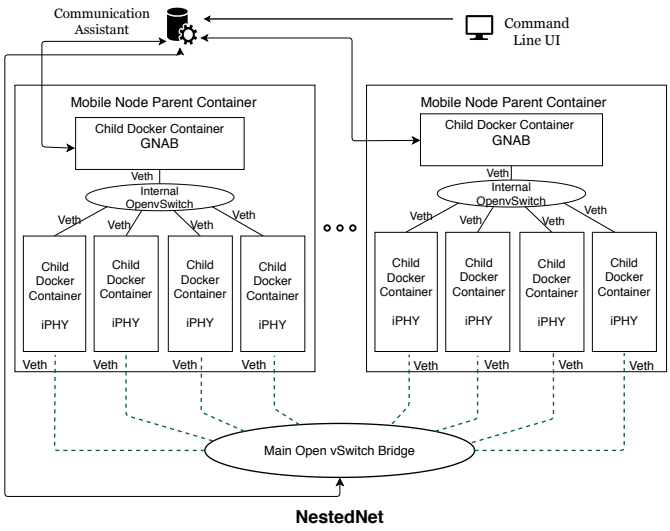

Fig. 1. An example MANET architecture implemented in NestedNet.

B. Nested Node Building Blocks

A distinctive feature of NestedNet is the use of nested containers. Since each node contains multiple components, a Docker-in-Docker (*dind*) image is used to support the execution of a Docker container inside another Docker container [17]. A *child* level of containers is created within each Docker host container (*parent*). Child containers can run application processes within the parent's resource boundaries after initialization and memory allocation. In NestedNet, each component (child) container is assigned a namespace, network stack, filesystem and process group.

To provide isolation and accessibility to each Docker container, namespaces are used. In namespace isolation, process groups are separated and cannot access resources in other groups. A network namespace provides a replica of the parent network stack but with its own routes and network devices. The use of a network namespace isolates network interface controllers and routing tables. The process identifier (PID) namespace isolates the allocation of PIDs to a specific container. Each container is instantiated with a separate process tree. The PID namespace of each child container allows each sub-component to have its own init-like process (PID 1), which controls all the processes within it. This supports container shutdown without affecting other child container operations, similar to hardware implementation.

In the mobile host, Linux-provided Veth devices are used to create intra-node connections. Veths connect namespaces in virtual hosts and switches in the emulator environment forming a virtual tunnel for packets via the network stack of the OS kernel. These tunnels are used to interconnect containers (nodes and components) to the *Internal OVS* or *Main OVS Bridge*. For intra-node communication, each end of the Veth pair is added as a port on the *Internal OVS*, and the other ends are inserted into the network namespaces in component containers. Veth links route packets between different namespaces in isolated Docker containers. Each GNAB and iPHY process is provided distinct network interfaces, each with an Internet Protocol (IP) address, a gateway, a routing table and Domain Name System (DNS) services. Processes ported from different components can have the same port or IP address and run without interference.

A control group (cgroup) [4] restricts access to resource subsystems (memory, network, disk I/O, CPU) in a Linux kernel by a collection of processes. Limits are applied to the collection of processes in the group. In our emulator, cgroups restrict access to resource subsystems such as memory, network, disk I/O and CPU, so that one container cannot exceed constraints or interfere with other containers operating in the same environment. Each child container executes binaries for a specific component. Resources are allocated by the container. There is environment separation of network interfaces, ports and libraries.

Each Docker container is built using instructions in a Dockerfile. A text-based Dockerfile contains all the commands needed to assemble an image. This approach facilitates the packaging of user-specified binaries and libraries into an image. The Dockerfile also includes file system information for the container.

IV. Experimental Setup

In this section, we describe our experimental setup, including a comparison versus another Docker-based prototyping environment.

A. Containernet

To assess the benefits of using nested containers in network prototyping, we evaluate NestedNet against Containernet, an existing network prototyping system based on Docker containers. Containernet is a fork of Mininet that uses Docker containers as nodes in emulated network topologies. Like NestedNet, it also supports OVS and Veth interfaces. Unlike NestedNet, node components are not isolated in containers. Their functionality is represented by *processes* in the node-level Docker container. As a result, Containernet cannot run binaries for multiple hardware platforms within the same container. For example, it is not possible to execute binaries that share the same IP address, MAC address, and Transmission Control Protocol (TCP) port in the node-level container. Processes running in a container share an execution environment. The execution of multiple processes in the environment that require different versions of the same library in the userspace and filesystem can cause conflicts. At the network level, each process must be configured to use separate ports. Unlike NestedNet, cgroups cannot be used across node components to manage resource usage.

Figure 2 illustrates the MANET topology from Figure 1 constructed using Containernet. The GNAB and iPHY code runs as five separate processes in a Docker container host. The processes communicate using Veth links through an *Internal Open vSwitch*. External inter-node (iPHY-to-iPHY) connections use the *Main Open vSwitch Bridge* with Veth

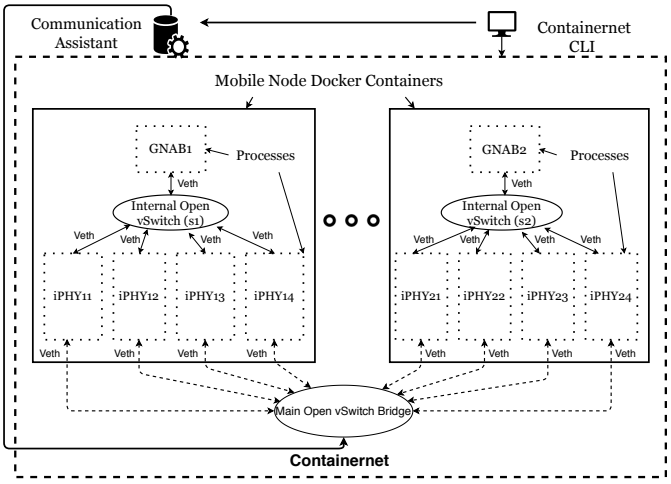

Fig. 2. Framework of MANET topology implemented with Containernet. The GNAB and iPHYs run as processes inside a parent (node) container. The GNAB and iPHY processes share interface, memory and CPU resources with their host node Docker container.

links. Internal interfaces are created with Bash scripts. The *communication assistant* and *Main Open vSwitch Bridge* operate as described in Section III-A.

B. Evaluation Overview

Our experimentation was performed with a 14-core Intel Xeon PowerEdge server (2.6 GHz, 128 GB). Both NestedNet and Containernet are launched on a VM running Ubuntu 16.04, kernel version 4.4.0. Open vSwitch version 2.11.1 and Docker daemon version 18.09.3 are used by both systems. For NestedNet, node containers and the VM host are configured with *dind*. The component containers are based on a Ubuntu 16.04 Docker image. For NestedNet, a base MANET topology is created using nested Docker containers, such that each node (parent) container contains five child containers, one GNAB and four iPHYs. $Iperf$ servers and clients are launched to evaluate link performance. For Containernet, all containers represent nodes and each container runs multiple processes ($iperf$ server/clients connected via Veth interfaces) representing GNAB and iPHYs.

C. Emulator-specific Setup

NestedNet: Each container has a set of interfaces belonging to two local networks. One is the intra-node network for iPHY-GNAB communication (192.168.0.x), the other is the network through which all iPHYs are connected via the *Main Open vSwitch Bridge* (192.168.2.x). For intra-node communication, an $iperf$ server is executed on the GNAB container allowing all iPHYs within the node to connect as $iperf$ clients. For the inter-node link test, one iPHY runs the $iperf$ server and the other executes the $iperf$ client. For latency evaluation, there is no need for a client-server relationship. Round-trip time is measured using an Internet Control Message Protocol (ICMP) echo with $ping$.

Containernet: The Containernet topology is created using an application programming interface (API) [18]. Veth interfaces in the container are not assigned to a specific namespace. An IP address within a container cannot be repeated and thus each iPHY in the network is assigned different IP addresses to enable inter-node link evaluation. A Bash script is used to create a local network using pairs of Veth links. One end of a Veth link is added as a port on the *Internal OVS* while the other has an IP address of a local LAN and remains open for process binding. For experimentation, an $iperf$ server or client can bind to an interface for intra-node and inter-node communication.

TABLE I
SYSTEM MEMORY USAGE FOR NESTEDNET AND CONTAINERNET. EACH NODE CONTAINS ONE GNAB AND FOUR IPHYS

Emulator component	NestedNet	Containernet
	Mem. usage (MB)	Mem. usage (MB)
Node container	0.6	4.0
Component container	0.5	-
Internal Docker daemon	47.0	-
Internal Open vSwitch	50.3	50.3
Main Open vSwitch Bridge	50.3	50.3
Containernet daemon	-	4.0
External Docker daemon	76.6	76.6
Single node	98.4	55.0
Four nodes	520.5	350.9

V. RESULTS

A. Initial Evaluation

In an initial experiment, NestedNet was evaluated for memory requirements. The *Docker stats* [19] and *htop* [20] tools were used to collect Docker container resource usage.

Table I shows the system memory usage for a test run of the NestedNet emulator. A four-node system was created for this evaluation. A Docker container uses about 600 KB and an Open vSwitch instantiation consumes about 50MB. This is the minimal memory usage for Docker container execution. Each node-level Docker container is instantiated with its own Docker daemon to support component containers. The daemon consumes about 47MB of memory. Excluding the Docker daemon on the host, 99MB are needed to represent a single mobile node with containers and daemons. A total of 175MB are needed to represent a one-node emulator including the Docker software running on the host. The memory consumption for a four-node emulator is:

- Four times the memory for a single node (98.4 MB)
- Memory for the Docker daemon on host (76.6 MB)
- Memory for the main Open vSwitch bridge (50.3 MB)

The total indicates a memory requirement of 521MB for a four-node emulator. Although this memory requirement is greater that Containernet, it is still modest and allows for component-level isolation within the containers.

B. Intra-node and inter-node communication performance for a 12-node network

In series of tests, the throughput and latency of NestedNet and Containernet are compared. Intra-node and inter-node

communication is evaluated by observing intra-node iPHY-GNAB connections and inter-node iPHY-iPHY connections. Performance results are obtained for one, two and four simultaneous intra-node and inter-node network connections per node for both emulators.

A 12-node network with four iPHYs each is used for this experiment. Each intra-node link is a iPHY-GNAB link which consists of a pair of Veth interfaces, interconnected via the *Internal Open vSwitch*. One end of each interface acts as an OVS port. For NestedNet, the other end is inserted into the child container. For inter-node links, iPHYs from different nodes are connected with Veth interfaces via the *Main Open vSwitch Bridge*. Only a single iPHY-to-iPHY connection exists between any two nodes. For intra-node tests, packets are sent from iPHY to GNAB.

Bash scripts were used to run tests in both emulators. A single test entails the transfer of a continuous data stream (TCP or ICMP) between the selected nodes. A total of 100 tests of 60 seconds each were conducted to evaluate throughput and delay. For inter-node communication, one of the two iPHYs is randomly selected as an $iperf$ server to establish connectivity. Each connection sends a continuous TCP data stream of 1,500 byte packets for 60 seconds at the maximum bandwidth available. Inter-iPHY data transfer is established when the $iperf$ server and client are run. Data is sent unidirectionally from one iPHY to another in the experiments. Latency was calculated using $ping$ which echoes ICMP packets towards a given IP address for 60 seconds.

Intra-node communication: For an intra-node connection, the Bash script selects a node and one of its internal iPHYs. The $iperf$ server and client establish a link to the GNAB. For the intra-node test with two connections, two iPHYs in a node are selected. Two $iperf$ clients are connected to the GNAB $iperf$ server for simultaneous data transfer. For the four-connection tests, all four iPHYs are selected and links are created in a similar fashion.

Experimental results over 100 tests are shown in Table II. It is observed that the intra-node bandwidth of NestedNet is comparable to that of Containernet for all three sets of experiments. The latency is about 20μs higher in NestedNet due to the multiple layers of containers that requires packet processing in the component (child) container, the node (parent) container stack and the destination component container stacks. Increasing the number of network connections communicating simultaneously increases the network processing time in the container network stacks and the CPU and memory usage of the underlying virtualized OS kernel. This effect causes the per-link bandwidth to drop with an increase in the number of intra-node connections.

Inter-node communication: For single link inter-node communication, the Bash script selects a random iPHY in each of two random nodes and runs the $iperf$ test for 60 seconds. For the two node connection test, one iPHY is selected in each of four random nodes. Two iPHYs in separate nodes run the $iperf$ server while the other two run the $iperf$ client, thus creating two distinct inter-node connections. To create four inter-node connections, eight nodes are randomly chosen. Four nodes have an $iperf$ server and four have $iperf$ clients. Not more than one connection exists between two nodes. The results of the inter-node experiments are shown in Table III. Every test selects a new random set of nodes and iPHYs to allow for a distribution of samples.

TABLE II
INTRA-NODE COMMUNICATION SUMMARY FOR A 12-NODE SYSTEM

Intra-Node Metrics (iPHY-GNAB)	Containernet	NestedNet
Avg. Throughput for 1 connection	30.36 Gbps	31.76 Gbps
Avg. Latency for 1 connection	0.056 ms	0.075 ms
Avg. Throughput for 2 connections	27.46 Gbps	26.29 Gbps
Avg. Latency for 2 connections	0.053 ms	0.072 ms
Avg. Throughput for 4 connections	17.57 Gbps	19.58 Gbps
Avg. Latency for 4 connections	0.052 ms	0.072 ms

TABLE III
INTER-NODE COMMUNICATION SUMMARY FOR A 12-NODE SYSTEM

Inter-Node Metrics (iPHY-iPHY)	Containernet	NestedNet
Avg. Throughput for 1 connection	26.92 Gbps	32.05 Gbps
Avg. Latency for 1 connection	0.082 ms	0.080 ms
Avg. Throughput for 2 connections	24.11 Gbps	25.42 Gbps
Avg. Latency for 2 connections	0.075 ms	0.074 ms
Avg. Throughput for 4 connections	14.19 Gbps	15.74 Gbps
Avg. Latency for 4 connections	0.074 ms	0.073 ms

In Table III, the inter-node throughput of NestedNet is slightly better than that of Containernet for all three experiments. NestedNet has a higher throughput for a single link, as each process is encapsulated by a container and thus can share the parent network stack evenly, as compared to Containernet. This effect can be explained as follows: The VMs of both emulators are allotted four CPU cores, where each core represents 100% CPU. The *Docker stats* command indicates the usage of each container with respect to total available CPU i.e. 400%. It was observed that while running the $iperf$ server and client in Containernet, the node with the client used 120% of 400% CPU, while the server used only 80%. Meanwhile, in NestedNet, the client used 118%, while the server used 104%. Resource allocation unevenness in Containernet caused the server to react more slowly, requiring TCP re-transmission and delayed acknowledgement packets. NestedNet provides better resource sharing and more consistent data transmission.

The latency for NestedNet is comparable for all sets of experiments. For inter-node communication, the ICMP packets in Containernet must cross different container network stacks via the *Main Open vSwitch* and thus similar latency is seen. Similar to intra-node connections, increasing the number of connections simultaneously in the network worsens the network processing and CPU and memory usage leading to packets being delayed and backlogged.

C. Stress test for intra-node processes

In this experiment, the benefits of isolating node components in containers is explored via an intensive network test. A network intensive background process is executed in a node to consume resources and affect other iPHY and GNAB pro-

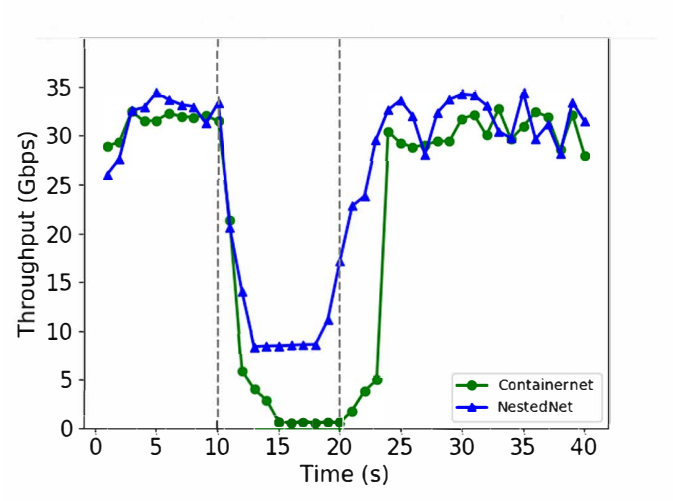

Fig. 3. Effects of stress test on the baseline process throughput. The background process executes from 10s to 20s. A large throughput drop is observed for the baseline process in Containernet (98%) versus NestedNet (72%).

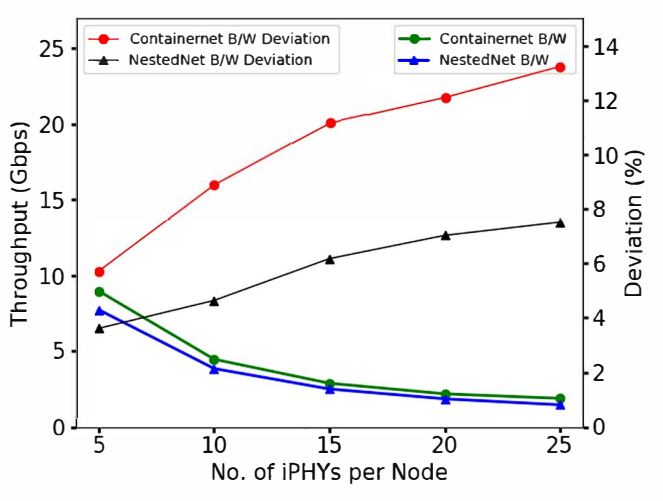

Fig. 4. Intra-node throughput effect of scaling intra-node components in a two-node network. The average GNAB-iPHY throughput per node and corresponding percentage standard deviation from average with increasing numbers of iPHYs per node are shown.

cesses. The background process bombards the network stack and increases system load by occupying the network stack and increasing the CPU and memory usage of the container. An $iperf$ client that generates 120 TCP concurrent threads and a $loopback$ interface is used to implement the background process. For testing, a standard intra-node iPHY-GNAB link is established as a baseline. An $iperf$ client sends 1,500 bytes of TCP data to an $iperf$ server using a single thread to form the baseline process. Linux processes are scheduled to run using prioritized round robin scheduling. In the case of TCP-based $iperf$ processes, slight delays at the data receive/send queues cause the scheduler to temporarily preempt a process. For instance, if the time slice for a process receiving data ends before the receiving buffer's lock is released, the lock will remain until the next time slice is allocated to the process. The time when the process resumes execution is directly dependent on system load. The background process increases the system load by introducing multiple parallel TCP streams.

Performance degradation of the baseline process due to the background process is determined as $D = \frac{T1-T2}{T1} \times 100$, where $T1$ and $T2$ are the throughputs of the baseline process before and after introduction of the background process, respectively. For Containernet, the background process executes in the same container as the baseline process. For NestedNet, the processes are isolated in separate containers. Containernet and NestedNet experiments were conducted for 40 seconds. The background process was started at the 10 second point and ended at the 20 second point, as shown in Figure 3.

The figure shows a significant throughput drop of 30Gbps to 500Mbps for Containernet (D = 98%). The baseline process executes in the same environment (parent container) as the background process. They share the same namespace, interfaces and network stack. These effects may cause packet processing delays due to higher system load exerted by the background process. Concurrent bombardment of packets by the multi-threaded process further complicates timely packet processing of the baseline process by the network stack of the parent container. In Figure 3, a throughput drop from 30Gbps to 8Gbps is observed in NestedNet, i.e. D = 72%. The processes execute in child containers with dedicated network stacks, interfaces and namespaces. The containers enable fairer CPU and memory sharing among the processes, so the baseline process can send and receive packets in a timely manner. The 72% drop is attributed to the load on the underlying virtualized kernel OS of the parent container and the host.

D. Scalability evaluation of intra-node components

In this experiment, the effects of using more than five nested containers per node on intra-node throughput are considered. The MANET topology is regenerated for each experiment with an increasing number of iPHYs per node, ranging from five to twenty-five. The bandwidth on iPHY to GNAB links is considered. For a given number of iPHYs per node, a total of 100 tests with TCP streams running for 90 seconds were conducted. Each test entails $iperf$ clients (iPHYs) sending constant TCP data of 1,500 bytes to the GNAB $iperf$ server. The throughput was averaged over all iPHYs per node. The standard *deviation* of the throughput was obtained for the intra-node links. The effect of scaling on the GNAB-iPHY link throughput for a two-node system is shown in Figure 4. The green and blue plots indicate the average link intra-net throughput per node for Containernet and NestedNet. The red and black plots show the increase in standard deviation of throughput per node as a percentage of the average. Each point is averaged over 100 trials.

In Figure 4, it is observed that increasing the number of iPHYs significantly drops the throughput per intra-node link. The iPHYs and GNAB processes execute in the parent

TABLE IV
CONTAINER STARTUP TIME (SECONDS)

Nodes	iPHYs	Containernet	NestedNet
12	4	28s	98s
2	25	9s	93s

container resulting in network processing delays and affecting the throughput. Our measurements indicate that average throughput per link can deviate from 6% to 13% from the average for Containernet. This deviation can be attributed to the processes competing for the CPU and network resources. Each client (process) may receive intermittent access to CPU resources, resulting in a delay in transmission/re-transmission. Moreover, the GNAB server may encounter delays in sending acknowledgments due to the lack of CPU time consumed by the clients. This issue may lead to TCP re-transmission, duplicate packets and packet loss.

NestedNet shows a similar drop in intra-node throughput. Increasing TCP application usage increases overall system load, stressing the underlying kernel of the parent and the host. The link throughput deviation is less in NestedNet, varying from 4% for five iPHYs per node to 7% for 25 iPHYs per node. The division of CPU shares among iPHY containers during execution mitigates process allocation limitations. Hence, each iPHY $iperf$ process is more likely to transmit and receive packets in a timely manner and receive associated acknowledgements.

E. Emulator Startup

For NestedNet, emulation starts when the CA sends the visibility graph to the GNABs. The CA then configures the *Main Open vSwitch Bridge* to create direct connections between iPHYs according to the link requests received from the GNABs. Thus, initialization time is defined as the time taken for the creation of all the containers and Open vSwitches followed by the initial link creation. The emulator initialization time relative to node count was measured.

The start-up time for NestedNet is significantly higher (Table IV) than Containernet. A 12-node environment with five sub-components (four iPHYs and one GNAB) needs over a minute and a half to build. The primary overhead is related to the child container image loading and creation. The time taken for emulator startup is roughly linear in the number of nodes. This trend is a result of Docker and Open vSwitch instantiation and link setup that is proportional to the number of direct links that must be established.

VI. CONCLUSION

Our new prototyping environment provides the first emulation environment that includes nested containers. This feature allows for a more accurate representation of subsystems in complex network nodes. Both node components and nodes themselves are represented as containers. Open vSwitch is integrated into the system to allow for intra- and inter-node connectivity. We have successfully tested this new system using an emulated MANET with 12 nodes. Our system provides better throughput (32 Gbps) and more stable results compared to a competing Docker-based emulation system. The use of nested containers has a CPU, memory usage and start-up time overhead, but the nested-containers approach is a suitable model for network system hierarchies. Experimentation with node counts of up to 120 is underway. The emulator could also be used to evaluate networks-on-chip [21].

REFERENCES

[1] *Geni: An Open Infrastructure*, Geni, Jun. 2020. [Online]. Available: "http://www.geni.net"

[2] *Getting Started with Docker*, Docker, Inc., Jun. 2020. [Online]. Available: "http://www.docker.com"

[3] E. W. Biederman, "Multiple instances of the global Linux namespaces," in *Proceedings of the Linux Symposium*, 2006, pp. 101–112.

[4] F. Checconi, T. Cucinotta, D. Faggioli, and G. Lipari, "Hierarchical multiprocessor CPU reservations for the Linux kernel," in *Proceedings of the International Workshop on Operating Systems Platforms for Embedded Real-time Applications*, 2009, pp. 15–22.

[5] B. Pfaff, J. Pettit, K. Amidon, M. Casado, T. Koponen, and S. Shenker, "Extending networking into the virtualization layer." in *Hotnets*, 2009.

[6] M. Kerrisk, "Linux Programmer's Manual - Virtual Ethernet Device (veth)," Feb. 2018. [Online]. Available: "http://man7.org/linux/man-pages/man4/veth.4.html"

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

[8] S. Galgano, "EMANE distributed wireless network emulation framework," Feb. 2020. [Online]. Available: "https://github.com/adjacentlink/emane"

[9] R. R. Fontes, S. Afzal, S. H. Brito, M. A. Santos, and C. E. Rothenberg, "Mininet-WiFi: Emulating software-defined wireless networks," in *IEEE International Conference on Network and Service Management (CNSM)*, 2015, pp. 384–389.

[10] S.-Y. Wang, C.-L. Chou, and C.-M. Yang, "EstiNet OpenFlow network simulator and emulator," *IEEE Communications Magazine*, vol. 51, no. 9, pp. 110–117, 2013.

[11] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "CORE: A real-time network emulator," in *IEEE Military Communications Conference*, 2008, pp. 1–7.

[12] M. A. To, M. Cano, and P. Biba, "DOCKEMU–A network emulation tool," in *IEEE International Conference on Advanced Information Networking and Applications Workshops*, 2015, pp. 593–598.

[13] M. Peuster, H. Karl, and S. van Rossem, "MeDICINE: Rapid prototyping of production-ready network services in multi-PoP environments," in *IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2016, pp. 148–153.

[14] H. H. Liu *et al.*, "CrystalNet: Faithfully emulating large production networks," in *ACM Symposium on Operating Systems Principles*, 2017, pp. 599–613.

[15] F. Farias, A. de O. Junior, L. B. da Costa, B. A. Pinheiro, and A. J. G. Abelem, "vSDNEmul: A software-defined network emulator based on container virtualization," *International Journal of Simulation Systems, Science & Technology*, vol. 20, no. 4, 2019. [Online]. Available: https://arxiv.org/abs/1908.10980

[16] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *IEEE International Symposium on Network Computing and Applications*, 2015, pp. 27–34.

[17] J. Petazzoni, "Docker in Docker," Feb. 2018. [Online]. Available: "https://github.com/jpetazzo/dind"

[18] M. Peuster, "Containernet API," 2020. [Online]. Available: "https://github.com/containernet/containernet"

[19] *Display a live stream of container resource usage statistics*, Docker, Inc., Jun. 2020. [Online]. Available: https://docs.docker.com/engine/reference/commandline/stats/

[20] H. Muhammad, "htop - an interactive process viewer for Unix," https://hisham.hm/htop/, 2020, last accessed 7 June 2020.

[21] A. Laffely *et al.*, "Adaptive systems on a chip (aSoC) for low-power signal processing," in *Asilomar Conference on Signals, Systems, and Computers*, 2001.