Scaling Network Emulation to Data Centers for Heterogeneous Containerized Network Devices

***Abstract*—In data centers, to verify the feasibility of coupling, and efficiency of network device configuration, huge effort has been put into network *simulation* and *emulation*. While emulation with VM is greatly limiting the scalability, simulation also suffers from a lack of fidelity to the devices it models. We confirm this problem by conducting test of single-machine emulation/simulation, discovering large scalability gap in between. To further identify the bottleneck of emulation scalability, we delve into two major aspects: virtual network interfaces and process isolation. We find out that the disproportional size of unnecessary NICs, redundant encapsulation / decapsulation via kernel network stack, and process switching overhead are the major causes of this problem. Observing that *containers* has gradually become the *de facto* way of distributing open source / proprietary routing software suite, we proposed NetEmos, a *single process*, *scalable* network emulation operating system with *zero modification* to containerized routing software suites. NetEmos stripes the socket functionality from kernel to user space, leading to effortless manipulation of emulation network traffic, lowering the spacial complexity from factorial to linear. Also, NetEmos combines the functionality of multiple process into a group of threads, significantly lowers the TLB flush / cache invalidation rate. Benchmark experiments show that NetEmos successfully scaled container-based network emulation, making it approaches the limit of inaccurate simulators while maintaining its fidelity.**

***Index Terms*—operating system, routing software, container, network emulation.**

I. Introduction

In today's digital era, data centers play a pivotal role in supporting the ever-increasing demands of modern applications and services. [1] Data centers rely heavily on network devices, such as routers, to efficiently manage and route network traffic. Traditionally, these routers have been proprietary hardware devices, manufactured by companies like Cisco and Huawei. However, with the rise of software-defined networking (SDN) [2], [3] and network function virtualization (NFV) [4], there is a growing need to emulate these network devices in a software environment.

Network emulators has been constantly emerging to fit into different requirements for configuration validation. Those emulators are majorly divided into 3 catagories: First is production-ready, VM based emulators: CrystalNet [5], CloudSim [6]; second is container-based, fully-isolated network emulators: [7]–[9]; there are also emulators utilizing Linux network namespaces [10]: Mininet [11], Core [12], [13]; finally Simulation-based emulators: [8], [11], [14]–[29]

Network emulators often grapple with two primary challenges that impact their effectiveness. Firstly, many emulators face issues related to the isolation of virtual machines, leading to a lack of scalability or demanding substantial computing resources. Secondly, the reliance on idealized models within these emulators tends to overlook the nuanced implementation differences found in proprietary software.

In response to these challenges, the rise of Docker containers has significantly reshaped the landscape of network emulation. Docker containers provide a lightweight and efficient alternative to traditional virtual machines, addressing the scalability concerns faced by emulators. The containerization approach ensures improved isolation between network elements while minimizing redundant kernel space usage, thus offering a more resource-efficient solution. This is exemplified in emulators like Containerlab [8], which leverages Docker containers for rapid and streamlined network test environment deployment.

Despite the advancements brought about by Docker containers in network emulation, scalability remains a persistent challenge. Our team conducted a comprehensive study to assess the limitations of container-based emulators, specifically employing Docker and custom network configurations using veth-pair (§III). The focus of our investigation was to scrutinize the upper threshold of nodes that can be effectively run on a single host—a crucial parameter for assessing scalability.

The results of our study revealed that the scalability of container-based emulators, while an improvement over traditional virtual machines, is still constrained. Our investigation into the limitations of container-based emulators pinpointed several key factors contributing to scalability challenges. As shown in Fig. 1, chief among these factors are the disproportional size of unnecessary Network Interface Cards (NICs), redundant encapsulation/decapsulation processes via the kernel network stack, and the overhead associated with process switching.

In this paper, we propose *NetEmos* (§IV), an x86-64 operating system based on the Linux Kernel dedicating to container-based network emulation. In response to the identified challenges, our solution simplifies the manipulation of emulation network traffic, introducing a more efficient paradigm that significantly reduces the spatial complexity from factorial to linear. By relocating socket functionality to user space, our solution streamlines the emulation process, enhancing overall performance and scalability.

To maximize the capacity of the system, we implement NetEmos with a single process approach (§V). This consolidation proves instrumental in significantly lowering the Translation Lookaside Buffer (TLB) flush and cache invalidation rate. The execution of our solution involves the interpretation and

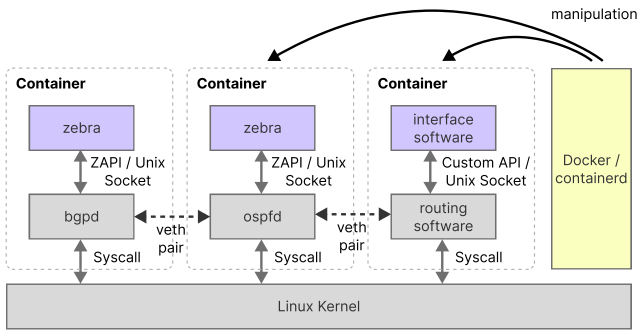


Fig. 1. Traditional containerized network emulation process

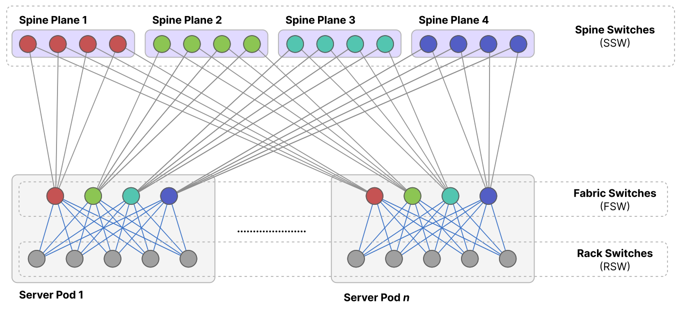


Fig. 2. Sample data center network topology configuration

loading of executables in ELF format by a singular process. This process serves as the central orchestrator, responsible for seamlessly handling the relocation of symbols related to socket system calls, such as socket, from the standard C library (libc) [30]. During the executable loading phase, our approach meticulously identifies and relocates all relevant symbols, ensuring that when these system calls are invoked within the program, our custom code executes to deliver the intended functionality.

We rigorously evaluated the effectiveness of our solution by subjecting it to comprehensive tests against simulation-based configuration validation tools, such as Batfish [31], [32]. The results demonstrated a remarkable enhancement in scalability, aligning closely with simulation-based approaches. Notably, our solution not only achieved comparable scalability but also maintained superior accuracy in the emulation of routing tables.

This validation process reaffirms the robustness of our solution, showcasing its ability to seamlessly integrate with and surpass existing tools in the domain of configuration validation. The subsequent sections will provide detailed insights into the testing methodology, results, and implications of this comparative evaluation.

II. Background

In this section, we briefly describe the need of network emulation in data centers.

A. Data Center Networking Architecture

Initially, data centers predominantly emphasized managing north-south traffic, employing the conventional spine-leaf architectures as their foundational framework [33]. However, with the ascendancy of cloud computing as the prevailing paradigm within data center operations, the significance of east-west traffic has surged, progressively constituting over 70% of the overall data center network traffic. This evolution necessitates denser networks, accompanied by an augmented demand for Layer-3 switching capacity [33]. A sample data center topology configuration is shown in Fig. 2.

In response to this shifting landscape, the Border Gateway Protocol (BGP), traditionally utilized for routing on the Internet, has extended its domain of application to become a pivotal protocol within data centers. This adaptation is especially prominent as systems have been purposefully designed to integrate and optimize BGP functionalities for data center environments [34]. For example, DigitalOcean have implemented BGP as an integral part of their data center fabric [35]. BGP's ability to efficiently handle dynamic and scalable routing within the data center network makes it a suitable choice for managing the surge in east-west traffic.

B. Containerized Network Devices

Proprietary network devices from companies like Cisco [36] and Huawei have long been the backbone of enterprise networks. These network devices are typically built as specialized hardware devices that implement routing protocols and provide advanced features for efficient data transmission. In recent years, these vendors have started shipping their routing software in the form of Docker images. [8] Containerization enables the deployment of router software on standard server hardware, decoupling the software from the underlying hardware.

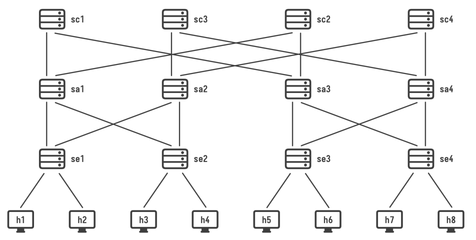
In addition to proprietary routing software, there is a significant presence of open-source routing software in the networking industry. Projects like Quagga [37], FRRouting [38], and Open vSwitch [39] provide routing and switching capabilities that can be utilized in emulated network devices. Emulating network devices with open-source routing software allows for greater customization, interoperability, and flexibility in network deployments.

III. Container Scalability Issue

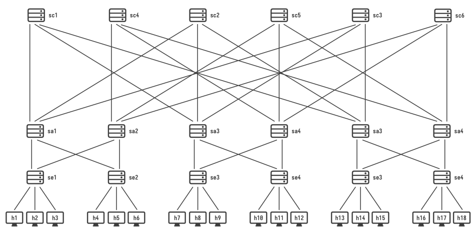
Although the evolution of containerization does provide more scalability compared to network of virtual machines, the essence of isolation brought by containers yields great overhead in the situation. In the following, we make an alternative network emulator based on Docker and use it to test the limitation of containerized network emulation[1].

On the same testbed, we tested the memory utilization and maximum number of nodes a system can emulate. We build a fat-tree topology for the representatives of emulation

[1]Source code available at https://github.com/xxx/xxx.



(a) $n = 2, m = 2$



(b) $n = 3, m = 2$

Fig. 3. Demonstration of different fat trees.

and simulation. We make a working config set for core, aggregation and edge switches that uses BGP to establish routing table. Of any scenario below occurs, we consider the emulation / simulation is a failure:

1) The system runs out of memory and is killed by the OS.
2) Any host of the system cannot ping one of all other hosts.

We run the setup, until failure, to see the maximum scalability of one solution. The results are shown in Table I.

TABLE I
SCALABILITY OF EMULATION / SIMULATION

System	Memory usage/GiB	Memory usage per node/MiB	Maximum nodes
Docker [40]	**17.0**	103.6	168
Batfish [32]	46.2	**89.6**	**528**

A. Analysis

As shown in Table I, it can be seen that when the memory usage of a single node is similar, Batfish [32] significantly increases the number of nodes that can be run on a single machine. However, Docker cannot continue to enumerate even if the memory is not full, indicating that the bottleneck of containerized emulation is in the kernel layer, acknowledging that the user space usage of docker is not high.

The subdued usage of Docker emulation, both CPU and memory, indicates that the emulation solution cannot reach its maximum scalability due to poor emulation efficiency.

1) CPU Usage: The subdued CPU usage can be attributed to frequent TLB flushes and cache invalidation, shedding light on the intricacies of resource utilization in container-based emulation.

In our emulation scenario, consider the frequent establishment and destruction of network topologies. Each change may necessitate adjustments in memory mappings, leading to TLB flushing. As a consequence, the CPU spends cycles reloading the TLB, diverting computational resources from other tasks. Also, where multiple threads or processes handle network-related tasks, such as routing updates, cache invalidation becomes significant [41]. For instance, if one thread updates routing information, other threads' caches must be invalidated to ensure they fetch the latest data. This synchronization process incurs overhead, contributing to lower overall CPU usage.

Frequent TLB flushing and cache invalidation introduce delays and extra computational steps, reducing the efficiency of CPU utilization. The CPU spends more time managing these housekeeping tasks rather than executing the core processing tasks efficiently. Consequently, the observed lower CPU usage in container-based emulation can be attributed to the overhead introduced by these operations, impacting the overall system performance.

2) Memory Utilization: In the context of single-host emulation scalability, memory utilization emerges as a critical factor. While this limitation holds true for traditional solutions like Batfish and VM, container-based emulation in our testbed faces challenges in scaling beyond 224 nodes, despite a theoretical limit of around 1000 nodes as depicted in Fig. 7b.

Due to the temporal usage of Batfish [32] and docker in Appendix B, ee infer that the primary constraint for container-based emulation lies in the redundant Layer-2 emulation facilitated by Veth pairs. These pairs, while essential for establishing network links, incur substantial CPU overhead. This observation aligns with the theoretical constraint on memory usage, suggesting that the intricacies of Layer-2 emulation, particularly in the context of Veth pairs, play a pivotal role in limiting the scalability of container-based emulation on a single host.

In the context of our emulation system, the use of Veth pairs is integral for establishing Layer-2 network links between emulated nodes. However, this seemingly straightforward process incurs a substantial overhead due to the following reasons:

1) Redundant layer-2 emulation
2) Kernel network stack processing

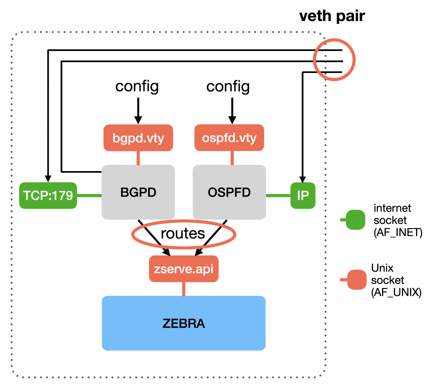
IV. SINGLE NODE MINIMIZATION

In this section, we introduce the basic idea to mitigate the problem, and the architecture of our design.

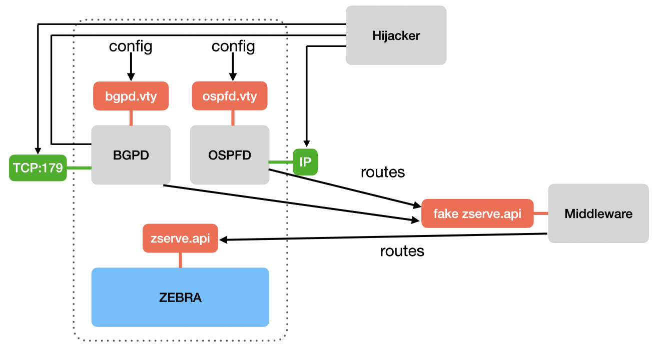
A. Basic Idea

We use FRR as an entry point of our design. As described in ContainerLab [8], the container image of proprietary routing softwares like Cisco IOS [36], Nvidia Cumulus Linux [42], and other open-source solutions like SONiC [43] use similar architecture of FRR:

- **Docker init process** serves as the initialization process responsible for generating isolated namespaces. All other processes within the system are descendants of this crucial initiation process.



(a) FRR



(b) NetEmos Node

Fig. 4. Organization of a single FRR node's daemons, and the injection places in solution. (a) FRR original organization. (b) places of interception

- **Daemons** is the core components handling routing tasks. Each daemon is assigned a specific functionality, encompassing these constitutional daemons:
  - *Routing protocol softwares*
  - *Routing / forwarding information base integration*
  - *Interfacing*
- **Unix Socket Communication**, inter-process communication (IPC) among the routing software suites primarily occurs through Unix domain sockets. This communication method bypasses the protocol stack, enhancing performance in the exchange of information between processes.

Illustrated by the Free Range Routing (FRR) [38] example in Fig. 4a, a node typically opens sockets for communication. These sockets form the backbone of both internal and external communication, influencing major scalability challenges such as Veth pair usage, redundant kernel network stacks, and cache invalidation. Moreover, the integration of Routing/Forwarding Information Base (RIB/FIB) functionality, initially requiring a dedicated process per node, can be consolidated into one process to reduce memory consumption.

To implement this idea, we introduce the NetEmos operating system, a solution that places control over all sockets in user space, diverging from the traditional kernel-side approach. This involves hijacking socket-related system calls to eliminate reliance on the kernel network stack.

B. Details of NetEmos

In our proposed NetEmos operating system, the implementation involves intercepting specific socket-related system calls to enhance efficiency and control. When a node necessitates inter-node traffic, it initiates a call to `socket(AF_INET)`. At this point, our system hijacks the call, facilitating the direct establishment of a communication channel between the involved workers. Similarly, for intra-node (inter-process) traffic, when a node invokes `socket(AF_UNIX)`, we can process it directly to circumvent context switching or pass it to the kernel if needed. Additionally, when a node seeks information about its Network Interface Controllers (NICs) by calling `socket(AF_NETLINK)`, we exploit this system call interception to create distinct virtual network devices for different nodes. This approach enables a fine-grained control over network interactions, optimizing both inter-node and intra-node communication within the emulation system.

The Linux kernel offers network namespaces as a feature, allowing the creation of distinct network environments within a single operating system. However, the limitation arises from the smallest granularity being a process, preventing the creation of separate namespaces within a process. Despite this restriction, consolidating multiple processes into one yields significant advantages, such as reducing the frequency of TLB flushing and cache invalidation, leading to increased cache hits. Our solution addresses this by combining multiple nodes into a single process.

The implementation of the single-process idea is depicted in Fig. 4a, comprising two crucial components: the **hijacker** and the **middleware**. The hijacker primarily handles the direction of inter-node traffic, while the middleware redirects RIB/FIB integration to obtain routing information. Additionally, it facilitates the standardization of RIB/FIB integrators across all nodes. This dual-component structure ensures efficient management of both inter-node communication and routing information integration within a unified process.

The hijacker plays a central role in directing inter-node traffic. Meanwhile, the middleware takes charge of redirecting the RIB/FIB integration process, ensuring that routing information is seamlessly acquired. This middleware functionality proves instrumental in streamlining the RIB/FIB integrators, creating a cohesive approach across all nodes in the system. Together, these components contribute to the successful implementation of the single-process model, optimizing the handling of inter-node communication and routing data integration. In the overarching architecture of the NetEmos, as illustrated in Fig. 5, a single-process model is realized, incorporating the Linux kernel. This configuration eliminates the need for TLB flushing within the system. The operating system, driven

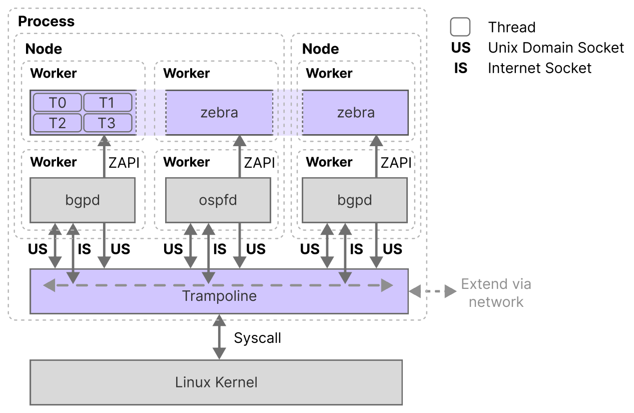


Fig. 5. The NetEmos operating architecture architecture.

by a solitary process, efficiently manages both inter-node and intra-node (inter-process) communication through memory sharing—considered the most effective means of information exchange without introducing racing conditions.

Upon loading the emulation configuration, the OS operates with only one process alongside the Linux kernel. Inter-node and intra-node communications are seamlessly facilitated through memory sharing, optimizing the efficiency of information transfer. Each node is equipped with a dedicated buffer for sending routing information, and the received data is never duplicated. This memory-sharing approach minimizes the use of extra space, resulting in a linear ($O(n)$) allocation and deallocation process. The architecture, characterized by its streamlined communication and memory utilization, exemplifies the efficiency and scalability achieved by the NetEmos.

V. Implementation and Evaluation

In this section, we implemented a NetEmos prototype, and compare it to the current solution: ContainerLab [8], our container-based emulator (see Section §III) and Batfish [39].

A. Hijacking Implementation

To implement the single-process model, NetEmos employs a process to interpret and load executables in the ELF format. During the loading process, the system identifies all relocations of symbols associated with socket-related system call wrappers, such as socket, in the C library (libc). These symbols are then relocated with custom entries in the program.

Essentially, when the loaded program invokes a socket-related system call, such as `socket()`, the execution is redirected to our custom code instead of the standard libc functionality. This mechanism allows NetEmos to seamlessly intercept and handle socket-related operations, enabling precise control over the networking functionalities within the user space.

During the loading phase of executables within NetEmos, an array of functions like `socket()` and related `bind()`, `listen()`, `connect()`, `send()`, `recv()` undergo interception and substitution by the trampoline. Notably, these functions serve diverse purposes beyond mere socket operations, necessitating a meticulous record-keeping of file descriptors by the trampoline to discern calls specifically involving socket file descriptors.

One of the key challenges addressed by NetEmos during this loading phase is the potential for symbol name clashes, especially when functions share identical names across different executables. To circumvent this issue, NetEmos implements a sophisticated symbol resolution mechanism in the form of a *Directed Acyclic Graph (DAG) forest loading*. Each executable forms an independent dependency DAG, fostering a disentangled environment for symbol resolution. This approach enables trampolines and individual nodes to specify their unique preload libraries, facilitating the coexistence of nodes with similar semantics, even if sourced from different libraries, and sharing symbols bearing identical names.

The symbol resolution process is outlined in Algorithm 1. This algorithm ensures effective resolution of symbols during the loading process, allowing the trampoline to intercept and redirect function calls as needed.

Algorithm 1 Symbol Resolving

```
SYMBOL_RESOLVE($s, l$)
param $s \in \Sigma$: Symbol name
param $l \in L$: Library to search
returns $n \in \mathbb{N} \cup \{\varnothing\}$: Symbol location

BEGIN
    if $s$ not defined in library $l$ :
        return $\varnothing$
    $n \leftarrow$ resolved symbol $s$ in library $l$
    return $n$
END
```

Algorithm 2 Symbol Searching

```
SYMBOL_SEARCH($s, H, h, P$)
param $s \in \Sigma$: Symbol name
param $H \subseteq \Sigma$: Symbol set to be hijacked
param $h : H \to \mathbb{N}$: Hijacked symbol location
param $P = (l, d) \in L \times 2^L$: Library and its dependencies
returns $n \in \mathbb{N} \cup \{\varnothing\}$: Symbol location

BEGIN
    if $s \in H$ :
        return $h(s)$
    $n \leftarrow$ SYMBOL_RESOLVE($s, P.l$)
    if $n \neq \varnothing$:
        return $n$
    for $l' \in P.d$:
        $d' \leftarrow$ LIB_DEPS($l$)
        $n \leftarrow$ SYMBOL_SEARCH($s, \varnothing, \varnothing, (l', d')$)
        if $n \neq \varnothing$:
            return $n$
END
```

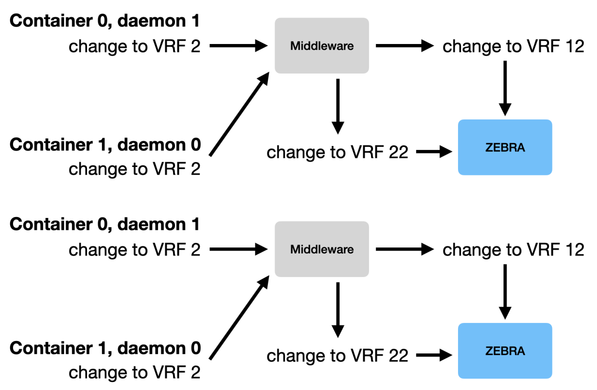


Fig. 6. VRF Translation.

where Σ is symbol set, L is library set.

To keep track of the opened sockets, and make sure that each node have a isolated set of file descriptors, NetEmos use a custom method called *File Descriptor Translation*. The file descriptors returned by irrelevant system calls, for example `open()`, is kept untranslated. On `socket()` syscall, trampoline requests a new null file descriptor from the kernel, and keeps the necessary information in the socket file descriptor table (in user space of trampoline).

To meticulously manage and isolate sets of file descriptors for each node while tracking opened sockets, NetEmos employs a proprietary approach known as *File Descriptor Translation*:

When the `socket()` syscall is invoked, the trampoline takes a strategic step by requesting a null file descriptor directly from the kernel. Subsequently, it retains essential information within the socket file descriptor table, residing in the user space of the trampoline. This meticulous process guarantees that each node possesses its dedicated and isolated set of file descriptors, preventing unintended interference or cross-contamination between nodes during the emulation process.

B. RIB/FIB Aggregator Unification

The unification of the Routing Information Base (RIB) and Forwarding Information Base (FIB) aggregator is achieved through a noteworthy feature: a single Zebra (the RIB/FIB aggregator of Zebra) has the capability to handle multiple Virtual Routing and Forwarding (VRF) instances. This architectural choice allows for the consolidation of routing information across diverse VRFs within the context of a single Zebra entity.

In the emulation system, the proper modeling of Virtual Routing and Forwarding (VRF) instances is a crucial aspect, and this necessitates the implementation of a mechanism called *VRF Translation*. Analogous to the previously discussed File Descriptor Translation (Section §V-A), VRF Translation ensures that VRF instances are appropriately modeled by the system.

As illustrated in Fig. 6, VRF Translation mechanism guarantees that the translation of VRFs within the same node is identical, facilitating coherent emulation within individual nodes. Simultaneously, it ensures that VRFs utilized by different nodes remain disjointed, preventing any unwanted overlap or interference between emulation instances.

This approach to VRF Translation contributes to the overall robustness and fidelity of the emulation system by accurately representing and isolating VRF instances as they operate across various nodes in the network emulation environment. By enabling a single Zebra instance to efficiently manage multiple VRFs, the emulation system gains a streamlined and unified approach to RIB/FIB aggregation. This not only optimizes resource utilization but also simplifies the overall management and coordination of routing information within the emulation environment. The consolidated handling of VRFs by a singular Zebra instance exemplifies a strategic design decision aimed at enhancing the efficiency and coherence of the emulation system's routing infrastructure.

C. Hierarchy

As shown in Fig. 5, the proposed solution involves a hierarchical structure comprising five layers: **Emulation Swarm**, **Emulator Process**, **Node**, **Worker**, and **Thread**. An Emulation Swarm includes multiple hosts running the emulator process. Each emulator process can consist of multiple nodes. A node, in turn, can host multiple workers, and each worker comprises multiple threads. This layered hierarchy is designed to organize and manage the emulation environment efficiently.

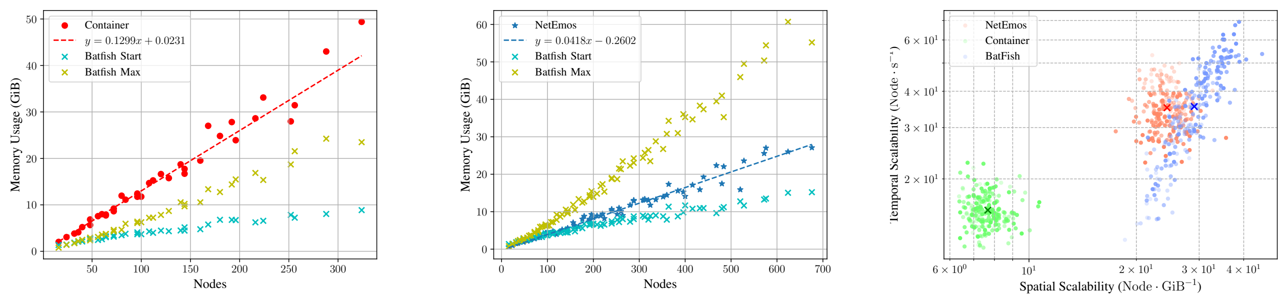
1) Emulation Swarm: In the proposed architecture, an *emulation swarm* is defined as a collection of hosts that execute emulator processes. Each host within the swarm is capable of handling local traffic. However, in cases where a host cannot manage the traffic locally, it establishes communication with other hosts in the same swarm through the network. This distributed approach allows for efficient traffic handling and resource utilization within the emulation swarm.

2) Emulator Process: In the proposed architecture, a *emulator process* serves as an emulator for multiple network nodes. The ideal scenario involves running only one emulator process on a single host to maximize the scalability of the emulation. Within the emulator process, a trampoline is responsible for loading the executables from all nodes and initiating a main thread for each of them.

It's important to note that while nodes and workers, the layers below the emulator process, are physically implemented as groups of threads, they are conceptually treated as distinct entities within the emulator process.

3) Node: In the proposed architecture, a *node* is essentially a group of workers, collectively representing a single network device. Each node within the emulation system is designed to emulate specific network functionalities. Notably, the key distinctions between nodes lie in the handling of `socket(AF_NETLINK)` related system calls and the establishment of connections.

4) Worker: In the proposed architecture, a *worker* is essentially a group of threads initiated by a single executable file. This executable file serves as the emulation core for



(a) Docker Memory Usage (b) NetEmos Memory Usage (c) Scalability

Fig. 7. Scalability comparison between Docker emulator, NetEmos and Simulator (Batfish). (a) Shows the memory usage of Docker comparing to Batfish as the node size grows. (b) NetEmos. (c) Scalability in both spatial (x axis) and temporal (y axis). The more opaque the color is, the more nodes it is emulating. '×'marks out the final score. Upper and righter is better.

specific functionalities. Within a node, multiple workers may be employed to manage distinct aspects of emulation, allowing for a modular and scalable approach to handling various network functionalities.

Theoretically, a worker can be shared among multiple nodes as long as it doesn't involve inter-node traffic, exemplified by RIB/FIB aggregator in Section §V-B.

5) Thread: In the context of the proposed emulation architecture, a *thread* represents the smallest schedulable unit in the Linux operating system. Threads within a worker are distinct units of execution, each designed to fulfill specific tasks such as communication handling or logging. Physically, these threads are tangible entities presented within the worker, and their execution is orchestrated by the Linux scheduler.

D. Evaluation

To further test our system, we build a prototype of NetEmos and use fat-tree topology to compare to containerized emulation and Batfish [32].

A **fat tree** here has two parameters: the number of pods n, and the number of leaves per pod m. By the context, the topology is irrelevant except its number of nodes and edges. The formula of the number of nodes and edges represented by the parameters are:

$$|V| = 4mn \tag{1}$$

$$|E| = mn(m + n + 1) \tag{2}$$

When $m \approx n$, $|E|$ can be approached by $|V|$ as:

$$|E| \approx mn(2\sqrt{mn} + 1) \approx \frac{1}{2}|V|^{3/2} \tag{3}$$

A proof of Equation 3 can be found in Appendix A. The experiment aimed to assess various metrics, including topology establishment time, route converging time, establishment memory usage, and routing memory usage, with the objective of deriving two comprehensive scores: spacial score and temporal score. The spacial score is formulated to gauge memory usage per node, as expressed in Equation 4, concurrently elucidating the relationship between number of nodes and memory consumption. Also according to Equation 3, it is an rough estimation of edge-memory relevance.

TABLE II
SPATIAL AND TEMPORAL SCORE OF DOCKER, NETEMOS AND BATFISH

Type	**Spatial** (Node/GiB)	**Temporal** (Node/s)
Docker	7.6669	15.6217
NetEmos	**24.3795**	**35.1218**
Batfish [32]	29.0662	35.4476

$$s = E^{-1}\left[\frac{\mathrm{d}M}{\mathrm{d}n}\right] \approx \left(\sum_{n \in N}\left(\frac{\max_n M}{2n} + \frac{\mathrm{avg}_n M}{2n}\right)\right)^{-1} \tag{4}$$

To align with the requirements of configuration validation, the temporal score assesses performance per node, specifically quantifying the time taken for the entire experiment per node. In order to capture the nuanced performance distinctions associated with processes utilizing multiple CPU cores, we introduce the concept of a **parallelogram penalty** for scenarios involving multi-core usage. The evaluation of the temporal score is delineated by the following formula:

$$t = E^{-1}\left[p_p \cdot \frac{\mathrm{d}T}{\mathrm{d}n}\right] \approx \left(p_p \cdot \sum_{n \in N}\left(\frac{t_e + t_r}{n}\right)\right)^{-1} \tag{5}$$

where p_p is parallelogram penalty, represented by total CPU usage; t_e is establishment time; t_r is route converging time.

To gain a deeper insight of the architectures and what is happening underneath, we use FRRouting [38] as the routing software in Docker emulation, NetEmos emulation, and corresponding configuration schema for Batfish emulation.

The result of the experiment is shown in Fig.7, and the final scores are listed in Table II.

As shown in Fig. 7, efficient routing information exchange in Batfish relies on a routing calculation schema that prioritizes time efficiency at the cost of increased space consumption, as indicated by both Fig. 7a and Fig. 7b. In routing scenarios, the memory usage experiences a substantial surge, reaching four times the idle state, exemplified in these figures.

In Fig. 7a, containerized emulation bases on Docker exhibit significant spacial overhead attributable to redundant processes, TLB switches and Veth pairs. Notably, the memory usage of VMs surpasses that of Batfish during active routing.

Thanks to the adoption of single-process emulation, NetEmos brings about a notable enhancement in scalability.

In Fig. 7b, NetEmos performs exceptionally, with the memory usage's growth rate similar to Batfish's, and far less maximum usage.

In comparison to traditional simulation solutions like Batfish, NetEmos offers greater fidelity and a more general solution. Regardless the fidelity improvement, NetEmos shows similar scalability just like simulators.

VI. Related Works

Some work has been done to minimize VM-based emulation to achieve greater scalability.

VM Based Emulators: A VM-based network emulator involves the use of virtual machines (VMs) to replicate and simulate network environments. GNS3 [13] continues to be a popular network emulation tool, delivering new versions regularly. Recognized for emulating networks of commercial and open-source router. CrystalNet [5] is a VM-based, production-ready emulator. It uses BGP safe boundary to determine the emulation scale, and utilizes VXLAN as the tool to scale greatly. Colosseum [24]provides open-source wireless software for wireless network emulation. Based on standard PC hardware and radios.

Container-based Emulators: A container-based network emulator utilizes containerization technology, such as Docker, to emulate network environments. Containers are lightweight, standalone, and portable units that encapsulate software and its dependencies. Containerlab [8] is a dynamic, open-source network emulator that swiftly constructs network test environments using a DevOps-style workflow. Mininet is a widely used network emulator for research and education in Software Defined Networking [11]. It remains actively maintained, with minor development observed on its GitHub repo. Tinet [29] is a container-based network emulator supporting a simple YAML config file for virtual network construction.

Simulators: A network simulator, irrespective of the underlying technology (VM or container), is a tool that models the behavior of a network by simulating the interactions and communication between network entities. Unlike emulators that replicate real-world components, simulators may use mathematical models and algorithms to predict how a network will behave under specific conditions. EVE-NG [44] Community Edition receives ongoing updates, offering a network emulator supporting virtualized commercial router images and open-source routers. Although focusing on the commercial EVE-NG product. Kathara is the next evolution in network emulation by the original developers of Netkit [45]. Supporting Docker and Kubernetes, Kathara facilitates network emulation scenarios across various operating systems and environments. CrowNet [26] is an open-source simulation environment modeling pedestrians using wireless communication. CupCarbon [27] simulates wireless networks in cities and integrates data from OpenStreetMap. MimicNet [9] is a network simulator using machine learning to estimate large data center network performance, was released in July 2019.

VII. Conclusion

In our exploration of network emulation challenges within data centers, we compared the limitations of virtual machine (VM) emulation and simulation methods. Our investigation uncovered significant scalability constraints in VM-based emulation and fidelity issues in simulation approaches. To identify the bottlenecks affecting emulation scalability, we delved into the aspects of virtual network interfaces and process isolation.

Our findings underscored issues such as oversized unnecessary NICs, redundant encapsulation/decapsulation through the kernel network stack, and process switching overhead, all of which contribute significantly to the scalability problem in emulation. Recognizing the widespread adoption of containers for distributing routing software suites, we introduced NetEmos. NetEmos is presented as a scalable network emulation operating system designed for seamless integration with containerized routing software suites, requiring zero modifications.

We achieve scalability in NetEmos by transitioning socket functionality from the kernel to user space, streamlining emulation network traffic manipulation, and reducing spatial complexity from factorial to linear. Additionally, our system consolidates the functionality of multiple processes into a group of threads, leading to a substantial decrease in TLB flush and cache invalidation rates. Benchmark experiments demonstrate NetEmos' successful scalability in container-based network emulation, bringing it close to the limits of less accurate simulators while maintaining high fidelity.

Appendix

A. Proof of Equation 3

The following a proof to formal representation of Equation 3: $\frac{1}{4}|V|^{3/2}$ is a second-order approachment of $|E|$ at input space $n = m$, or

$$\lim_{n\to\infty}\left(\left.\frac{|E|}{|V|^{3/2}/4}\right|_{m=n}\right) = 1 \tag{6}$$

$$\forall d \neq 0, \lim_{n\to\infty}\left(\left.\frac{\nabla^\top |E| d}{\nabla^\top \left(|V|^{3/2}/4\right) d}\right|_{m=n}\right) = 1 \tag{7}$$

First is the proof of Equation 6. As replacing $|V|, |E|$ with parameters m, n, the proof is obvious:

$$\begin{aligned}
&\lim_{n\to\infty}\left(\left.\frac{|E|}{|V|^{3/2}/4}\right|_{m=n}\right)\\
&=\lim_{n\to\infty}\left.\frac{mn(m+n+1)}{(4mn)^{3/2}/4}\right|_{m=n}\\
&=\lim_{n\to\infty}\frac{n^2(2n+1)}{8n^3/4}\\
&=\lim_{n\to\infty}\frac{2n^3+n^2}{2n^3}\\
&=1
\end{aligned}$$

The gradient of $|E|$ respect to (m,n) is:

$$\begin{aligned}
&\nabla_{n,m}|E|\\
&=\left[\frac{\partial}{\partial m}mn(m+n+1),\frac{\partial}{\partial n}mn(m+n+1)\right]^\top\\
&=\left[2mn+n^2+n,2mn+m^2+m\right]^\top
\end{aligned}$$

The gradient of $\frac{1}{4}|V|^{3/2}$ respect to (m,n) is:

$$\begin{aligned}
&\frac{1}{4}\nabla_{n,m}|V|^{3/2}\\
&=\frac{1}{4}\left[\frac{\partial}{\partial m}(4mn)^{3/2},\frac{\partial}{\partial n}(4mn)^{3/2}\right]^\top\\
&=3\left[m^{1/2}n^{3/2},m^{3/2}n^{1/2}\right]^\top
\end{aligned}$$

When $m=n$, Equation 7 can be transformed to:

$$\begin{aligned}
&\lim_{n\to\infty}\left(\left.\frac{\nabla^\top|E|d}{\nabla^\top\left(|V|^{3/2}/4\right)d}\right|_{m=n}\right)\\
&=\lim_{n\to\infty}\left(\frac{(3n^2+n)d_m+(3n^2+n)d_n}{3n^2d_m+3n^2d_n}\right)\\
&=\lim_{n\to\infty}\left(\frac{3n^2(d_m+d_n)+n(d_m+d_n)}{3n^2(d_m+d_n)}\right)\\
&=\lim_{n\to\infty}\frac{3n^2+n}{3n^2} \qquad \text{if } d_m+d_n\neq 0\\
&=1
\end{aligned}$$

If $d_m+d_n=0$, there is $\nabla^\top|E|d=\nabla^\top\left(|V|^{3/2}/4\right)d=0$, the approachment still holds. Q.E.D.

A better unbiased approachment exists that works for any n:

$$|E|=e'_v(|V|)=\frac{1}{4}|V|^{3/2}+\frac{1}{4}|V| \tag{8}$$

It satisfies second-order unbiased approachment:

$$|E|\big|_{m=n}=\left.\left(\frac{1}{4}|V|^{3/2}+\frac{1}{4}|V|\right)\right|_{m=n}$$

$$\nabla|E|\big|_{m=n}=\nabla\left.\left(\frac{1}{4}|V|^{3/2}+\frac{1}{4}|V|\right)\right|_{m=n}$$

however, Equation 8 has no simple inverse (the inverse is Equation 9), making it not as practical as Equation 3.

$$e'^{\,-1}_v(x)=\frac{1}{3}\left(C-\frac{24x-1}{C}+1\right) \tag{9}$$

$$C=\sqrt[3]{216x^2+24\sqrt{3}(27x^4-x^3)^{1/2}-36x+1}$$

B. Temporal Usage of Batfish and Docker

As shown in Fig. 8, efficient routing information exchange in Batfish relies on a routing calculation schema that prioritizes time efficiency at the cost of increased space consumption, as indicated by both Fig. 8a and Fig. 8b. In routing scenarios, the memory usage experiences a substantial surge, reaching four times the idle state, exemplified in these figures.

References

[1] X. Zhang, P. Yu, P. Wu, X. Li, and S. Huang, “Multiple dedicated end-to-service path protection (mdespp) scheme in optical distributed datacenter networks,” in *2017 16th International Conference on Optical Communications and Networks (ICOCN)*, 2017, pp. 1–3.

[2] Y. Jia, N. Hua, Y. Li, and X. Zheng, “Applying multi-controller collaboration in fine-grained all-optical intra-datacenter networks,” *Journal of Optical Communications and Networking*, vol. 10, no. 7, pp. 37–48, 2018.

[3] D. Comer, A. Rastegarnia, and W. J. Reed, “Dclab: A reconfigurable sdn testbed for datacenter networks,” *IEEE Networking Letters*, vol. 2, no. 3, pp. 145–148, 2020.

[4] M. M. Erbati and G. Schiele, “Application- and reliability-aware service function chaining to support low-latency applications in an nfv-enabled network,” in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 120–123.

[5] 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 *SOSP ’17 Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, October 2017, pp. 599–613. [Online]. Available: https://www.microsoft.com/en-us/research/publication/crystalnet-faithfully-emulating-large-production-networks/

[6] J. O. Williams, D. A. Kapoor, V. C. Vikram, and L. T. Jackson, “Cloudsim,” http://www.cloudbus.org/cloudsim/, 2022, release 6 in August 2022.

[7] I. W. Claire, N. K. Aarohi, and K. Adebayo, “cnet,” https://www.csse.uwa.edu.au/cnet/index.php, 2022, version 3.5.3, released in April 2022.

[8] V. Choudhary, “Containerlab,” https://containerlab.dev/, 2023, version 0.36.1, released in January 2023.

[9] L. T. Alexander, N. A. Khanna, and S.-H. Kang, “Mimicnet,” https://github.com/eniac/MimicNet, 2019, last commit in July 2022, research project.

[10] Y. Zeng, M. Chao, and R. Stoleru, “Emuedge: A hybrid emulator for reproducible and realistic edge computing experiments,” in *2019 IEEE International Conference on Fog Computing (ICFC)*, 2019, pp. 153–164.

[11] P. L. Antoine, M. O. Thompson, K. Minho, J.-H. Kim, C. Ling, W. S. Noah, and C. Xin, “Mininet,” http://mininet.org/, 2021, version 2.3.0, last released two years ago.

[12] A. Sharma, L. Thomas, R. M. Aishwarya, D. K. Anjali, M. O. Thompson, and R. S. Verma, “Core: Common open research emulator,” http://coreemu.github.io/core/, 2022, version 9.0.1, released in November 2022.

[13] L. Ka, P. R. Maxime, J. Thomas, Y. Mei, N. Kwok, and J. W. Obi, “Gns3,” https://gns3.com/, 2023, version 2.2.37, released in January 2023.

[14] T. Ka, W. Smith, R. A. Malhotra, V. T. Sofia, C. Okafor, O. Anderson, O. D. Anderson, and Y. Ka, “Cloonix,” http://clownix.net/, 2023, version 28, released in January 2023.

[15] V. Torres and P. Lambert, “Linux network test stack (lnts),” http://lnst-project.org/, 2019, version 15.1, released in August 2019.

[16] N. Kwok, K. M. Adebayo, and J. Taylor, “Nemu: Network emulator for mobile universes,” https://gitlab.com/v-a/nemu, 2023, version 0.8.0, released in January 2023.

[17] C. Ho and H. Lefevre, “Netlab,” https://github.com/ipspace/netlab, 2023, version 1.5.0, released in February 2023.

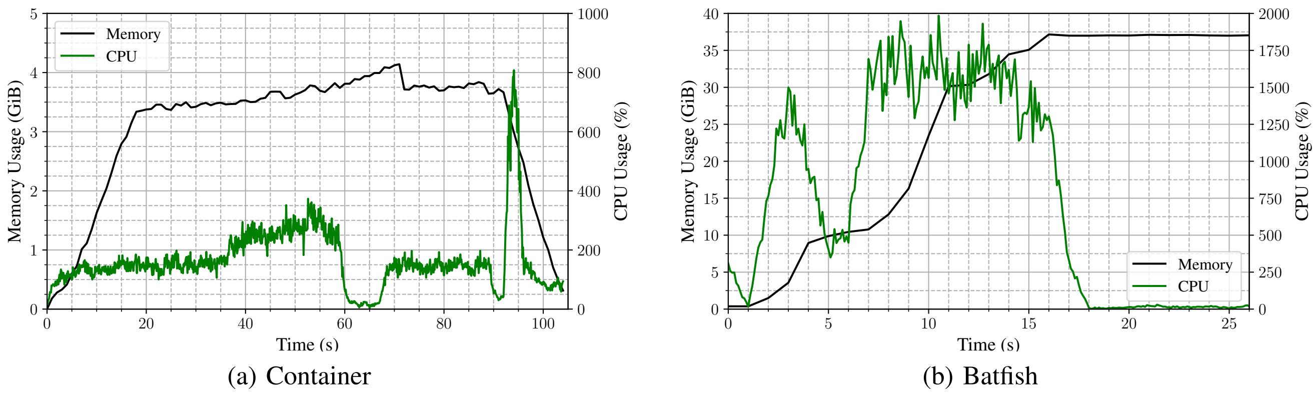


Fig. 8. Usage (cpu, memory) over time. (a) shows container-based emulation using topology $\mathtt{fattree}(6, 6)$. (b) shows Batfish simulation using topology $\mathtt{fattree}(10, 10)$.

[18] R. Malhotra, E. S. Sophie, W. Yee, and H. Smith, "ns-3," https://www.nsnam.org/, 2022, version 3.37, released in November 2022.

[19] J.-H. Kim, P. Lambert, and Y. Ka, "Omnet++," https://omnetpp.org/, 2022, version 6.0.1, released in September 2022.

[20] Y. Hui, A. S. Ravi, and L. Il-Sung, "Openconfig-kne," https://github.com/openconfig/kne, 2022, version 0.1.7, released in December 2022.

[21] S. Taylor, A. Davis, T. Ka, P. A. Lambert, N. C. Wai, M. O. Thompson, A. S. Ravi, and W. Yee, "Shadow," https://shadow.github.io/, 2023, version 2.4.0, released in January 2023.

[22] A. Nkosi, "Virtual networks over linux (vnx)," http://web.dit.upm.es/vnxwiki/index.php/Main_Page, 2023, latest version released on Sep 14th, 2020.

[23] A. P. Raj, Y. Soo, H. Wagner, and W. Mei, "vrnetlab," https://github.com/vrnetlab/vrnetlab, 2021, last commit in December 2021.

[24] J. Obi, A. P. Sharma, H. Smith, O. A. Daniel, and C. B. Marie, "Colosseum," https://www.northeastern.edu/colosseum/, 2022, srsRAN 22.10 released in November 2022.

[25] J.-Y. Choi and Y. Mei, "Cooja," https://docs.contiki-ng.org/en/develop/doc/tutorials/Running-Contiki-NG-in-Cooja.html, 2023, no official release yet.

[26] S.-H. Kang, A. Davis, Y. Mei, N. W. Chun, E. Johnson, and V. Choudhary, "Crownet," https://github.com/roVer-HM/crownet, 2022, version 0.9.0, released in May 2022.

[27] P. Lambert, "Cupcarbon," http://cupcarbon.com/, 2019, no official release, recent commit refers to Version 5.2.

[28] K. C. Ng, J. Obi, S. T. Ethan, J. W. Obi, and O. D. Anderson, "Meshtasticator," https://github.com/GUVWAF/Meshtasticator, 2023, no tagged release, recent pull requests merged in February 2023.

[29] V. S. Torres, E. J. Grace, Y. Mei, K. Minho, and H. Lefevre, "Tinet," https://github.com/tinynetwork/tinet, 2020, version 0.0.2, released in July 2020, development ongoing.

[30] H. Li, Y. Dang, G. Sun, G. Liu, D. Shan, and P. Zhang, "LemonNFV: Consolidating heterogeneous network functions at line speed," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1451–1468. [Online]. Available: https://www.usenix.org/conference/nsdi23/presentation/li-hao

[31] M. Brown, A. Fogel, D. Halperin, V. Heorhiadi, R. Mahajan, and T. Millstein, "Lessons from the evolution of the batfish configuration analysis tool," in *Proceedings of the ACM SIGCOMM 2023 Conference*, ser. ACM SIGCOMM '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 122–135. [Online]. Available: https://doi.org/10.1145/3603269.3604866

[32] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, "A general approach to network configuration analysis," in *USENIX - Advanced Computing Systems Association*. USENIX - Advanced Computing Systems Association, April 2015. [Online]. Available: https://www.microsoft.com/en-us/research/publication/a-general-approach-to-network-configuration-analysis/

[33] "When you should consider leaf-spine network architecture." [Online]. Available: https://searchnetworking.techtarget.com/tip/When-you-should-consider-leaf-spine-network-architecture

[34] A. Abhashkumar, K. Subramanian, A. Andreyev, H. Kim, N. K. Salem, J. Yang, P. Lapukhov, A. Akella, and H. Zeng, "Running BGP in data centers at scale," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 65–81. [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/abhashkumar

[35] "Scaling Droplet Public Networking — digitalocean.com," https://www.digitalocean.com/blog/scaling-droplet-public-networking.

[36] "Cisco internetworking operating systems (ios)." [Online]. Available: https://www.cisco.com/c/en/us/support/docs/ios-nx-os-software/ios-software-releases-110/13327-ios-early.html?dtid=osscdc000283

[37] A. R. Singh and O. Anderson, "Quagga 1.2.4 Released," https://lists.quagga.net/pipermail/quagga-dev/2018-February/033333.html, 2018, version 1.2.4, released in February 2018, final release.

[38] R. V. Surya, N. K. Chun, J. Obi, and L. Hyun, "FRRouting — frrouting.org," https://frrouting.org/, 2023, version 9.1, released in December 2023, development ongoing.

[39] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vSwitch," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, May 2015, pp. 117–130. [Online]. Available: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff

[40] D. Merkel *et al.*, "Docker: lightweight linux containers for consistent development and deployment," *Linux j*, vol. 239, no. 2, p. 2, 2014.

[41] K. Fawaz and H. Artail, "Dcim: Distributed cache invalidation method for maintaining cache consistency in wireless mobile networks," *IEEE Transactions on Mobile Computing*, vol. 12, no. 4, pp. 680–693, 2013.

[42] "NVIDIA Cumulus Linux Architecture — nvidia.com," https://www.nvidia.com/en-us/networking/ethernet-switching/cumulus-linux/.

[43] "Sonic Foundation — sonicfoundation.dev," https://sonicfoundation.dev/.

[44] L. Yee, R. Verma, J. Ji-Ae, J. Thomas, and Y. Soo, "Eve-ng community edition," https://www.eve-ng.net/index.php/community/, 2022, version 5.0.1-13, released in August 2022.

[45] D. Kapoor, Y. Mei, and A. M. Singh, "Kathara," https://www.kathara.org/, 2023, version 3.5.5, released in January 2023.