Nextmini: A New Research Testbed for Network Emulation and Experimentation

Xindan Zhang, Shengwen Chang, Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

***Abstract*—As large language models are trained on datacenters with tens of thousands of compute nodes and are quickly becoming parts of our daily routines, the need for a flexible, easy-to-use, and high-performance research testbed for emulating and experimenting with new network protocols has become more pressing and relevant than ever. Conventional packet-level simulators are, by their nature of discrete-event simulation, not scalable enough; yet traditional network emulation testbeds, such as Mininet, are also showing their age with respect to the flexibility of implementing new algorithms, ability to run arbitrary application workloads, scalability to a large number of nodes, as well as the freedom of expanding beyond a single cluster to span multiple geographically distributed regions.**

In this paper, we present *Nextmini*, a modern, next-generation, high-performance networking research testbed for network emulation and experimentation. Implemented in Rust, it is designed from scratch to be as flexible as possible, accommodating a wider array of resource scheduling algorithms. It supports running arbitrary workloads — such as distributed machine learning training workloads — directly on the emulated network. Its design strikes an excellent balance between flexibility and performance, supporting both performant user-space emulation for maximum flexibility, as well as much higher kernel-level performance when users need such a performance boost. It is scalable to a larger number of nodes with ease in the same cluster, and can be easily expanded to span multiple geographically distributed datacenters. We conducted an extensive array of experiments to evaluate *Nextmini*'s tent-pole features, and to compare it with Mininet. Our results show both *Nextmini*'s raw power and its abundance of flexibility.

I. Introduction

With the prevalence of large-scale application workloads, such as distributed training of large language models [1], [2], it is increasingly important to design, implement, and experiment with new network protocols and resource allocation mechanisms at scale [3], [4]. Indeed, there is a pressing need for an experimental platform that serves as a playground for evaluating new ideas in networking research over both emulated and real-world networks, often spanning multiple geographically distributed regions.

We argue that such a pressing demand for a new experimental testbed *cannot* be satisfied by conventional discrete-event network simulators over the past four decades (*e.g.*, [5], [6]). The upshot of discrete-event simulations is that they provide packet-level granularity when it comes to reproducible experiments, but they are also well known to be lacking on *scalability*: it may take hours, sometimes even days, to simulate a second of real-world traffic as the network scales up and the volume of events that need to be processed explodes. In addition, network simulators also lack *realism*: the protocols they offer to simulate may not be exact replicas of their counterparts in a modern Linux kernel.

But what about conventional network emulation testbeds, such as Mininet [7]? Designed more than a decade ago, Mininet took advantage of some of the best technologies of its time: by placing host processes in network namespaces and connecting them with virtual Ethernet (`veth`) pairs, it supports setting up virtual networks of arbitrary topologies, without resorting to heavyweight OS virtualization alternatives, such as virtual machines. With a command-line interface and a Python API supported, Mininet's foundation is the software-defined networking architecture, where an OpenFlow controller is in charge of making control-plane decisions, and Open vSwitch [8] acts as a software switch in the Linux kernel to implement these decisions in the dataplane. A SIGCOMM Test of Time Award winner, Mininet's design is simple and elegant, while offering the best possible network performance.

That said, Mininet, along with its derivatives over the past decade, may already be showing its age. Since Mininet was introduced, Docker containers became the norm and the foundation for modern cloud computing. Similar to Mininet, these containers virtualize network namespaces; but different from Mininet, they also allow arbitrary workloads to run with minimal OS overhead. Though Docker containers may not be as lightweight as virtualizing network namespaces only, they are far lighter than virtual machines. As an even more enticing characteristic, modern container orchestration technologies — such as Docker swarm [9] and Kubernetes [10] — can be used to deploy a large number of containers across an entire cluster. These technologies may substantially simplify the deployment of a container-based emulation testbed in a large cluster, as compared to custom-tailored distributed solutions based on Mininet [11], [12].

Beyond the ubiquity of Docker containers, a new technological innovation has also emerged as the *de facto* standard of high-performance networked systems, such as web services, in the *user space*: *asynchronous network programming* with *coroutines*. A coroutine is a lightweight thread of execution: they are similar to OS kernel threads, can be suspended at predefined points of execution, and resumed to a state that they left off before suspension. Yet, with coroutines, a context switch has much less overhead — equivalent to a regular function call. Contemporary programming languages, such as Go [13] and Rust, support coroutines natively. With coroutines, network events can be processed asynchronously

and concurrently by thousands of coroutines, allowing for much more performant and battle-tested user-space networking frameworks, and making high-performance user-space networking a reality.

More than a decade since Mininet was designed, with the advent of modern technologies such as Docker containers, container orchestration, and asynchronous networking with coroutines, and driven by the pressing need for running distributed application workloads, we believe that it is the right time to revisit the spectrum of design choices that should power a modern network research testbed. While we are fans of Mininet's core design principles and its performance, we advocate for a new architectural design — from the ground up — to support better flexibility of evaluating new algorithms, the ability to run arbitrary application workloads, better scalability to a larger number of nodes across multiple physical or virtual machines, as well as the freedom of expanding beyond a single datacenter to span multiple geographically distributed regions.

In this paper, we introduce *Nextmini*, a thoroughly modern networking research testbed for evaluating new network protocols and resource scheduling algorithms under real-world application workloads. *Nextmini* inherits Mininet's fundamental design philosophy: its foundation is a software-defined networking architecture, where actions taken by all dataplane nodes are governed by a dedicated controller. Yet, despite its name, *Nextmini* makes several decidedly different design choices from Mininet:

- ♢ *Nextmini* focuses on a *user-space* design, without relying on technologies within the Linux kernel, such as eBPF [14] and kernel modules. To maximize performance, *Nextmini*'s user-space design takes full advantage of asynchronous network programming, stackless coroutines, and the Rust programming language. A major advantage of such a user-space design is its *flexibility*: it becomes much more straightforward to implement and evaluate new and complex traffic engineering or resource scheduling strategies, such as packet dropping, scheduling disciplines, traffic shapers and policers, as well as multipath routing protocols.
- ♢ In situations where the packet forwarding performance is of utmost importance and a user-space design becomes a performance bottleneck, *Nextmini* also supports a *high-performance* operating mode, which we simply refer to as the *max* mode. In its max mode, *Nextmini* trades off some flexibility to support performance approaching kernel-level packet forwarding, using the `splice` system call in the Linux kernel.
- ♢ To run arbitrary application workloads without modifications — such as distributed training of machine learning (ML) models — *Nextmini* is able to establish and expose a virtual network to an application via *TUN interfaces*. Used by virtual private networks (VPNs), a TUN interface is a virtual network device that operates at the network layer, allowing applications to send and receive network traffic obliviously, without knowledge of the underlying network topology.
- ♢ To scale up to a larger number of nodes in its dataplane, *Nextmini* takes a two-pronged approach. Similar to Mininet, it supports virtualizing only the network namespaces, and connecting nodes using virtual Ethernet (`veth`) pairs. This allows for a maximum degree of scalability on a single machine. Yet, to span multiple machines in the same datacenter — as well as multiple geographically distributed datacenters around the world — *Nextmini* also supports the deployment of dataplane nodes using Docker containers and Docker swarm [9], a modern container orchestration technology.

With respect to *Nextmini*'s implementation, one original and unique design choice stands out and is worth highlighting. *Nextmini* uses a simple yet elegant *actor model* [15], [16], where each *actor* maintains its private states and can only affect each other indirectly through message passing. Such a design choice minimizes data locking due to concurrency, which helps avoid a wide array of potential concurrency bugs in *Nextmini*'s implementation.

Needless to say, we have devoted much of our attention to evaluating *Nextmini*'s flexibility, scalability, and above all, performance. Where applicable, we have compared the performance of *Nextmini* with Mininet, and when operating in the high-performance *max* mode, both exhibit excellent packet forwarding performance on a single physical machine, reaching 132 Gbps over two hops. Given its user-space design without the need of any kernel modules, we are pleasantly surprised by the raw performance that *Nextmini* is able to achieve. With respect to scalability, we show that in *Nextmini*'s lightweight *namespace* mode, the memory footprint is only around 4.5 MB per dataplane node, allowing us to launch up to 10,000 nodes on a single physical machine. Last but not the least, we have also conducted a wide array of experiments to showcase *Nextmini*'s capabilities, including its support for distributed ML training across multiple geographically distributed datacenters.

II. Network Emulation: State of the Union

Over the past four decades, the reproducibility of networking research is, in general, realized with any of three alternative strategies: discrete-event network simulators, network emulation environments, and real-world experimental testbeds. On one extreme of the spectrum between control and realism, discrete-event network simulation (DES) frameworks, such as ns-3 [5], offer fine-granularity control and the best reproducibility: with a fixed random seed, an entire simulation can be repeated without variations. On the opposite end of the spectrum, real-world experimental testbeds, such as PlanetLab [17], support the deployment of overlay network protocols over hundreds of servers that are geographically distributed globally. It offers the best realism, as wide-area networks provide the underlying foundation for real-world network experiments.

Striking a balance between these two extremes, it has been widely accepted that network emulation testbeds, dating back to Emulab and Netbed [18] more than two decades ago, are also indispensable for reproducible networking research.

Sacrificing some of the fine-granularity control from discrete-event simulators, a network emulation testbed is inherently designed to offer better *scalability*: as there is no need to process discrete events in their timestamp order sequentially, an emulated system makes progress in real-time regardless of scale. In addition, an emulated system subjects real-world applications, network protocols, and operating system kernels to controlled and synthetic network topologies, and as a result offers much better *realism* than discrete-event simulations.

Mininet. The epitome of such emulation environments over the past decade is Mininet [7] (officially called *Mininet-Hifi*, a high-fidelity release of the original Mininet). Dubbed *network in a laptop* [19], Mininet was initially conceived and implemented to emulate an entire virtual network in a single physical machine. To build a virtual network among OS processes, Mininet utilizes the capability of the Linux kernel to assign virtualized network namespaces to each OS process, and to connect them with virtual Ethernet (`veth`) pairs.

The architectural foundation of Mininet is *software-defined networking* (SDN) [20], where concerns in the control plane, such as routing algorithms, are separated from the data forwarding plane, where packets are processed. With SDN, it becomes feasible to add new features to the controller without modifying the packet-forwarding switches. By using the Open vSwitch [8] as a kernel module and forwarding packets entirely within the kernel, it also offers stellar performance. Completing the package with an extensible command-line interface (CLI) and a Python API, Mininet offers a simple yet highly customizable platform for creating emulated networks with the best possible performance. It has been widely used [21] and extended to multiple machines in a cluster [11] over the past decade.

Recent network emulators. In the decade after Mininet was released, several of its derivatives have been proposed, using similar container-based emulation strategies. Containerlab [22], as an excellent example, is an actively maintained, open-source network emulator that provides a custom-tailored container orchestration tool to organize multiple containers into a virtual network topology, supporting many custom network OS images such as the Free Range Routing (FRR) router [23]. Containernet [24], on the other hand, is a fork of Mininet that allows the use of Docker containers as hosts in emulated network topologies. As another recent example, Kubernetes Network Emulation [25] is a network emulator that is designed to extend basic Kubernetes [10] networking to support virtual connections between nodes in an arbitrary network topology. All of these recent network emulators are designed to use Docker containers as the foundation, and to organize them into virtual network topologies. *Nextmini* shares such a design philosophy and can be used with Docker containers; but as we shall elaborate, comes with a twist that focuses solely on *user-space* packet forwarding, eliminating the complexity that comes with custom network OS kernels.

III. Nextmini: Technological Foundation

Mininet's heritage. As fans of Mininet's simplicity and elegance, we conceive *Nextmini* to inherit much of Mininet's excellent design. To begin with, both *Nextmini* and Mininet are built on the widely recognized principles of *software-defined networking* [20], such that control-plane policies (*e.g.*, routing protocols) and dataplane packet-forwarding mechanisms (*e.g.*, packet scheduling disciplines) are cleanly separated. To maximize the flexibility of implementing a wide array of resource allocation and routing algorithms, *Nextmini* incorporates a custom-tailored controller design, without adhering to the OpenFlow standard.

As another heritage from Mininet, *Nextmini* is built upon *container-based emulation*. Just like Mininet, we support virtualizing network namespaces only and connecting nodes using `veth` pairs. By assigning different network namespaces to OS processes, our design maximizes scalability, with respect to the number of dataplane nodes to be accommodated on a single physical or virtual machine. In addition, just like more recent network emulators such as Containerlab [22], *Nextmini* comes with first-class support for Docker containers, a *de facto* standard in modern cloud computing. By providing such support from the ground up by design, *Nextmini* offers two key advantages over Mininet: ① Users can tap into a variety of modern container orchestration tools — such as Docker compose, Docker swarm [9], and Kubernetes [10] — to deploy these containers on the same machine, across multiple machines in the same cluster, or even across datacenters that span geographically distributed regions. ② Arbitrary application workloads — including distributed training of ML models — can be executed within these Docker containers without any modifications. As Docker containers incur much less runtime overhead compared to virtual machines, it may even be recommended to run these applications in a *Dockerized* environment.

Asynchronous networking with coroutines in Rust. Conventionally, the abstraction of concurrency in modern operating systems is achieved via *kernel threads*, which provides a lightweight mechanism of realizing concurrency compared to traditional OS processes. In the recent decade, however, *asynchronous* programming becomes the norm, supported by *stack-based coroutines*. With stack-based coroutines (*e.g.*, in the Go programming language), a context switch between different coroutines becomes similar to regular function calls, and incurs much less runtime overhead than kernel threads. However, stack-based coroutines may still be less efficient due to the presence of the local call stack, which can be further optimized by implementing coroutines without using stacks at all, in which case runtime overhead would "vanish entirely," as proclaimed by Weber *et al.* [26]. With such *stackless coroutines*, local data is stored as fields in an active *instance* of the coroutine, rather than in a stack frame. Suspending execution in a stackless coroutine is, therefore, mapped to an ordinary `return` statement, and a context switch becomes *precisely* as fast as a function call.

Though stackless coroutines sound appealing at the high level, it is too challenging to implement them manually. One example of these challenges is that fields and the program counter in the coroutine instance need to be captured in its own structure, rather than the call stack. In addition, a runtime *executor* needs to become an overarching driving force that

calls the next ready coroutine as soon as some progress can be made. Such an executor is akin to a thread scheduler, but with much less context-switching overhead as no per-thread stack is involved.

The good news is that stackless coroutines are officially supported by Rust since `async/await` entered Rust 1.39 in 2019 [27], when the Rust compiler natively supported capturing local fields and the program counter in a special `struct` called `Future`, and runtime executors are provided by third-party libraries. One of the most widely used runtime executors is `tokio` [28], which implements, by default, a multi-threaded executor: each CPU core corresponds to a kernel thread, and each kernel thread executes a large number of stackless coroutines, called *tasks*, concurrently. Each task requires only an allocation of 64 bytes to maintain, and tasks are managed by `tokio` in the *user space*. The combination of Rust's native support for stackless coroutines and runtime executors such as `tokio` offers drastically improved networking performance, and has quickly become the foundation of production web services.

IV. Nextmini: Design and Implementation

A. Kernel vs. User Space: The Queen's Gambit

We started with an overarching design philosophy when *Nextmini* was initially conceived: we wished to take full advantage of modern advances in concurrent programming, with the powerful combination of multi-threaded runtime executors, stackless coroutines, and *fearless concurrency* [29] offered by Rust, and to implement the dataplane *entirely in the user space*. This is a substantial deviation from Mininet, where packet forwarding is conducted in the kernel with OpenFlow switches[1], such as the Open vSwitch [8]. Switching from a (predominantly) kernel-space design to the user space will — without doubt and by a substantial margin — *degrade* packet-forwarding performance. Yet, on the flip side of the coin, implementing *Nextmini* in the user space intuitively offers the best possible flexibility of emulating user-space flows, and designing new traffic engineering and resource scheduling algorithms. When extreme performance is needed, our design trades off some flexibility so that *Nextmini*'s packet-forwarding performance can ideally reach around the same level as — or only slightly inferior to — kernel-space packet processing.

As we implement such an overarching design philosophy, we are also strong believers in the importance of making the best possible design choices in complex systems. Before even the first line of code is written, we have several design objectives in mind: ① *Simplicity*. Mininet was designed to be simple to use: a virtual network experiment can be established and run using CLI or Python scripts. With *Nextmini*'s design, we aim to be even simpler for the users: in most cases, one only needs to supply a configuration file. ② *Separation of concerns*. Though hard to define and quantify, we wish to separate the concerns shared by various components in our design as much as possible. This is inspired by the design philosophy of software-defined networking, which *Nextmini* follows as well. ③ *Flexibility* of supporting arbitrary workloads with Docker. We aim to utilize Docker containers and Docker orchestration technologies to their full potential: with Docker containers, we can run arbitrary application workloads in our emulation testbed; and with Docker orchestration, we can easily span multiple physical machines in the same cluster, or even multiple virtual machines that are geographically distributed globally.

[1]While Mininet also supports user-space OpenFlow switches, their performance is so far below expectations that they are practically unusable [30], offering around three orders of magnitude lower throughput than kernel switches.

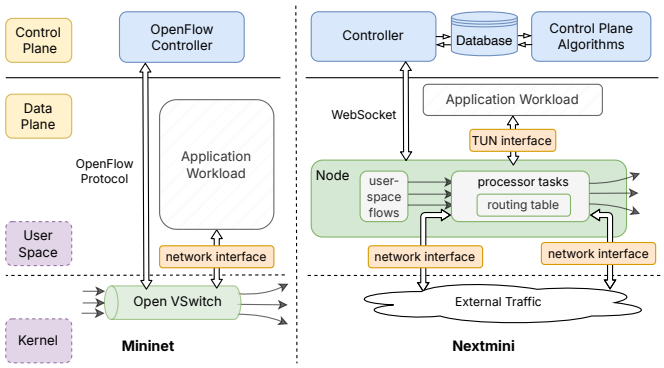


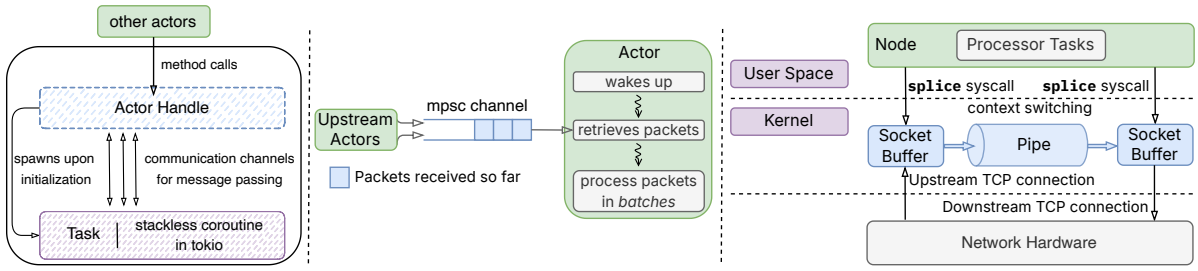
Fig. 1: Comparing Mininet with *Nextmini*: a birds-eye view.

B. Architectural Design: Openings

Fig. 1 illustrates a birds-eye view of *Nextmini*'s architectural design, inspired by some of the same design decisions in Mininet. Applying the same philosophy of software-defined networking, *Nextmini*'s controller is designed and built as a high-performance web server, communicating with dataplane nodes using the industry-standard WebSocket protocol [31], capable of bidirectional communication. To support the maximum degree of flexibility when it comes to accommodating new data types to be transmitted between *Nextmini*'s controller and dataplane nodes, we opt to deviate from the OpenFlow protocol [32], and to allow messages of *arbitrary* data types to be efficiently packed into binary form, and then exchanged on these bidirectional connections.

The dataplane nodes are implemented in the *user space*, and typically executed within Docker containers. Each node provides a virtual TUN network interface that allows for arbitrary distributed application workloads to run without modifications, an important feature not provided by Mininet. In addition, upon receiving requests from the controller, dataplane nodes can also initiate independent *user-space TCP* flows, which can be processed and forwarded concurrently, along with application traffic from TUN network interfaces. Clients and servers of these TCP flows are launched and completed on-demand, entirely in the user space within *Nextmini*'s dataplane implementation.

Network topologies are also implemented in the user space, with persistent TCP or QUIC connections connecting dataplane nodes. This is a drastically different design choice compared to Mininet, where topologies are determined at the IP layer. *Nextmini*'s user-space topologies provide the maximum flexibility: such topologies can even span across geographically distributed datacenters, and serve as a valuable



(a) The actor model: each actor is divided into a task and a handle. **(b) Processing all packets in the channel, in batches, upon resuming execution.** **(c) Implementing Nextmini's max mode using the splice system call in Linux.**

Fig. 2: Optimizing *Nextmini*'s dataplane: (a) The actor model, where actors do not share states and interact by passing messages only. (b) To improve performance, packets are processed in batches. (c) Using the `splice` system call to "connect" two TCP connections, drastically improving packet-forwarding performance by an order of magnitude.

experimental testbed for real-world networking protocols at the application layer, beyond conventional network emulation.

C. Control Plane: Exchanges

A Rust-powered asynchronous web server. A single centralized controller serves as the "brain" and a focal point in *Nextmini*'s design. It is far more capable and flexible than a conventional OpenFlow controller in software-defined networking: sophisticated traffic engineering and network resource optimization algorithms can be implemented in the control plane, based on *real-time* performance monitoring from all dataplane nodes. Such flexibility is made feasible by *exchanging* control messages with dataplane nodes in two-way communication channels, implemented as persistent WebSocket connections. To maximize the performance of establishing a large number of such connections and of sustaining substantial aggregate throughput between the control and data plane, we choose to implement the controller entirely in asynchronous Rust with the `tokio` runtime.

PostgreSQL database and real-time triggers. Following the norm of designing modern web servers for production, the controller stores all its states — including configurations, routing policies, topologies, link rates, as well as performance metrics reported by the dataplane — in a PostgreSQL database, a state-of-the-art database engine.

A core responsibility of the controller is to analyze variations in performance metrics that dataplane nodes report in real-time, and make potential updates to the routing, traffic shaping, and scheduling policies in response to these variations. In *Nextmini*, control-plane decisions are programmatically made at runtime by running simple Python scripts, making it straightforward for users to implement new algorithms. These Python-scripted control-plane algorithms depend on real-time subscriptions to insertions and updates in the PostgreSQL database, called *triggers*, which are special kinds of stored procedures that automatically fire when certain events, such as `INSERT` and `UPDATE`, occur on a database table. As an example, whenever a measured throughput value is received by the controller, a corresponding database table will be updated, and control-plane algorithms will be notified in real-time to make their resource scheduling or routing decisions.

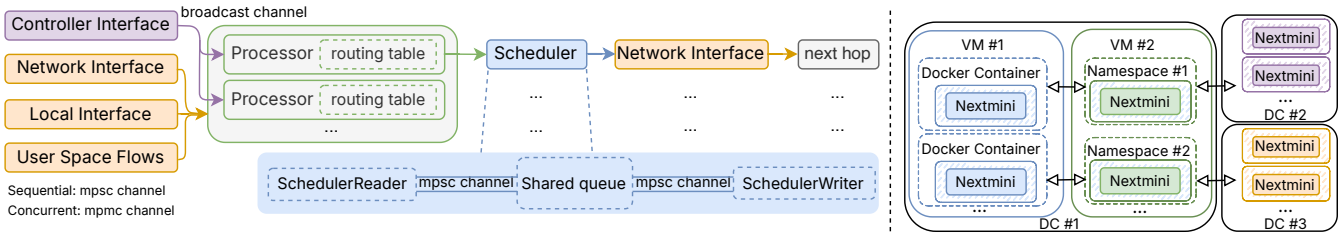
Since the database is *shared* between the Rust-powered controller and control-plane algorithms, the controller can also respond, in real-time, to triggers whenever the database is updated. This becomes useful when a control-plane algorithm makes a decision, such as an update to the routing policy. The algorithm simply needs to update the database, which triggers the controller to promptly send control messages to relevant nodes in the dataplane, installing or modifying pertinent routes. The logical assumption that all updates to the database will be implemented in real-time reflects a *separation of concerns*: the algorithm only needs to interact with the database as an intermediary, and is not tightly coupled with the controller itself.

D. Data Plane: Fork

Actor model. As we alluded, one unique design choice stands out and is worth highlighting: *Nextmini* uses the *actor model* [15], [16] throughout its dataplane design. As a conventional mechanism of communicating between threads, sharing memory is widely known to be questionable. Synchronization and locking mechanisms, such as semaphores and mutual exclusion locks, must be used to prevent data races and deadlocks when multiple threads write to a shared variable. As challenges of ensuring *thread safety* made concurrent software development more complex and error-prone, message passing has again been promoted with the *actor* model, where threads avoid sharing any states and interact by passing messages only using channels. As the `tokio` runtime in Rust supports high-performance broadcast and multi-producer single-consumer (MPSC) channels between tasks (*a.k.a.* stackless coroutines), it is very well suited for the actor model [33], where each actor is implemented as one or several tasks and spawned by its *handle* upon its initialization. As Fig. 2(a) illustrates, actors interact with each other by calling methods in their respective handles, implemented by sending messages to each other via MPSC channels.

Design: a first cut. Using the actor model, the starting point in each dataplane node revolves around the *processor* actor, which is responsible for forwarding inbound packets to next-hop destinations using a routing table. As illustrated in Fig. 3(a), upstream and downstream actors interact with the processor by passing messages on MPSC channels.

Two possible upstream actors may feed packets into the processor: ① the *local interface* reader, which receives packets from the application via the local TUN interface; and ② the



(a) Designing Nextmini: a first-cut starting point. **(b) Nextmini with container orchestration.**

Fig. 3: Designing *Nextmini*. (a) A first-cut starting point that revolves around *processor* tasks. (b) With container orchestration technologies, *Nextmini*'s dataplane nodes can easily be deployed across virtual machines or even datacenters.

network interface, which establishes and maintains TCP or QUIC connections with other dataplane nodes, as requested by the controller according to the network topology. Packets received from upstream nodes via these connections will be sent to the processor.

Correspondingly, two potential downstream actors may handle packets from the processor. ① the *local interface* writer, which sends packets destined to the local host to the application, again via the local TUN interface; and ② the *scheduler*, which queues outbound packets and schedules them according to a scheduling discipline (with *First-Come-First-Served* and *Weighted Round Robin* supported by *Nextmini*). The scheduler also incorporates packet dropping mechanisms (*Tail Drop* and *Random Early Detection*), traffic shapers (*Token Bucket*), and rate limiters. Ultimately, it sends outbound packets to the *network interface* actor, which is in charge of sending them out on the persistent TCP or QUIC connection it manages.

As multiple flows are routed through the processor, it becomes crucial to spawn multiple independent tasks, each maintaining its own local copy of the routing table and processing its own share of flows. There are two modes of mapping inbound flows to processor tasks: *concurrent* or *sequential*. With concurrent mapping, each packet in an inbound flow can be processed by any processor, which can be implemented efficiently by a multi-producer multi-consumer (MPMC) channel between the upstream actors and processor tasks. With sequential mapping, in contrast, each flow is mapped to only one processor, using a consistent hash function (such as the jump hash algorithm [34]).

As we observe in Fig. 4, under *sequential* mapping, controller-assigned lossless flows achieve substantially higher aggregate throughput than `smoltcp` flows. Interestingly, adding more processor tasks does not necessarily increase throughput, and throughput even degrades as the number of flows increases, suggesting that the performance bottleneck may not be related to packet processing capacity. We shall elaborate on this further in this paper when we seek to maximize performance.

The *controller interface* actor is responsible for interacting with the controller via a persistent WebSocket connection, and upon receiving updated routing policies from the controller, it broadcasts these updates to all processor tasks via a broadcast channel. On the reverse path, the network interface sends its live performance measurements to the *controller reporter*, which computes the metrics necessary to be sent to the

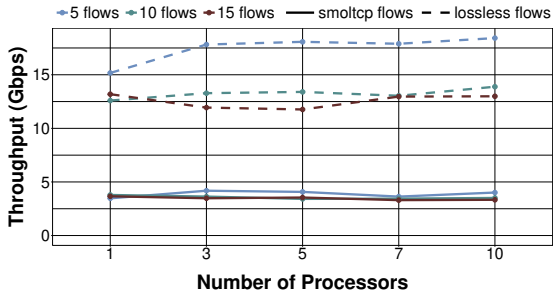


Fig. 4: The total aggregate throughput across all flows as the number of processors scales up for controller-assigned `smoltcp` and lossless synthetic flows.

controller.

Routing tables in the processor are designed to allow the maximum flexibility in making routing decisions. Each flow, defined by its flow identifier — a 4-tuple involving the source and destination addresses and port numbers — can be forwarded on its own route (which remains fixed once selected). Different flows with the same source and destination addresses can be forwarded on different routes, allowing multi-path routing. To accommodate such levels of flexibility, two hashmap lookups are required: the first maps a flow identifier to its route, and the second maps the route to the next-hop destination.

"Doubled pawns". Beyond application traffic via the TUN interface, users often require *synthetic* flows to be generated, much as in discrete-event network simulators. Such synthetic flows can, of course, be generated by benchmark applications such as `iperf`, but it is much simpler — and more scalable — to declare them in the controller configuration and have *Nextmini* generate them automatically. To this end, *Nextmini* supports two types of synthetic flows, our "doubled pawns": ① `smoltcp` flows, which run a full user-space TCP/IP stack; and ② lossless flows, which use a lightweight session protocol atop the existing TCP/QUIC overlay.

To shoehorn `smoltcp` into *Nextmini*'s dataplane is a non-trivial exercise, since it is not designed to operate behind a virtual network acting as a *de facto* "proxy" between its clients and servers. Once integrated, `smoltcp` flows enjoy full TCP semantics — connection handshakes, congestion control, and retransmissions.

As a sweetener to this recipe, lossless flows take a decidedly leaner approach. Since the persistent TCP/QUIC overlay links already provide reliability, ordering, and congestion control,

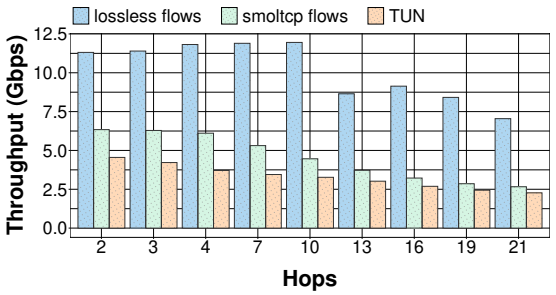


Fig. 5: User-space flows using lossless and `smoltcp` vs. the TUN interface: achievable throughput.

Nextmini introduces no additional TCP stack; instead, lossless flows use a minimal sliding-window session protocol: the sender streams fixed-size chunks and the receiver emits cumulative acknowledgements to ensure deterministic transfer completion. Losslessness follows from propagating backpressure through the user-space pipeline: when a downstream hop slows, upstream senders naturally throttle via transport-level flow control, avoiding packet drops in the dataplane while still allowing *Nextmini* to enforce per-flow shaping. As Fig. 5 illustrates, lossless flows reach nearly 12 Gbps for paths up to 10 hops on an Intel i7-13700K desktop (`Intel`), roughly twice the throughput of `smoltcp` flows. `smoltcp` flows themselves outperform the TUN interface for shorter paths, and are no worse for longer paths.

Arbitrary application workloads with Docker. A highlight in *Nextmini*'s design is its flexibility: due to its user-space design, it natively supports execution within Docker containers, and exposes a TUN interface for distributed applications to run without changes. As we show in Fig. 3(b), with a single `docker compose` command, multiple dataplane nodes can be started in seconds within the same virtual or physical machine, without the need for Mininet's proprietary CLI. With modern container orchestration technologies such as *Docker swarm* or *Kubernetes*, it is also straightforward to scale to multiple VMs in the same datacenter, or across multiple datacenters that are globally distributed. Regardless of where a dataplane node is situated, they connect with each other in a virtual topology via persistent TCP or QUIC connections. In this sense, *Nextmini*'s design punches above its weight: it is a network emulation testbed, but can serve as a real-world network testbed as well, running applications in a geographically distributed fashion.

E. Performance and Scalability: End Game

While our first-cut design emphasized flexibility, the Rust-powered `tokio` asynchronous runtime, used routinely in production web services, has already offered solid user-space *performance*. In addition, Docker containers are also widely known to be lightweight, offering an excellent memory footprint and *scalability*. But our objectives are more ambitious: we wish to offer *extreme* levels of scalability and performance, ideally approaching kernel-space solutions in Mininet.

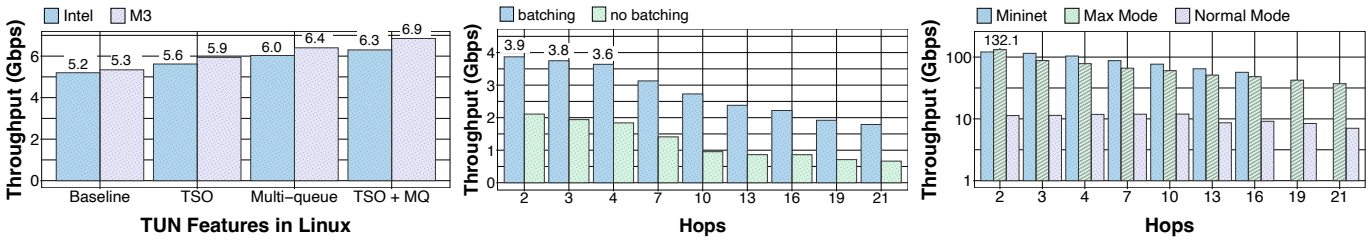
Maximizing performance in the user space. To maximize the performance of *Nextmini*'s dataplane, we painstakingly revisit every design choice and implementation detail — from thread-safe MPMC queues to production-quality QUIC frameworks — leaving no stone unturned. Two noteworthy examples stand out as highlights.

First, to maximize the throughput of application traffic via the TUN interface, we take full advantage of (a) multi-queue support since Linux kernel 3.8 [35], which uses multiple file descriptors to parallelize reading and writing when multiple flows co-exist; and (b) TCP Segmentation Offload (TSO), which improves throughput by offloading the segmentation of large TCP packets to the network interface card, reducing CPU overhead and allowing more efficient data transmission [36]. By supporting both multi-queue and TSO, experiments using Docker containers in the `Intel` desktop and a MacBook Pro M3 Pro notebook (`M3`) show that throughput improves from 5.3 to 6.9 Gbps, corresponding to an improvement of up to $1.3\times$, as illustrated in Fig. 6(a).

Second, to maximize the overall packet-forwarding performance, we also fine-tuned how actors process inbound packets throughout the dataplane design. Rather than processing packets one by one in each actor task, every time a task resumes execution, we make non-blocking calls — `try_recv()` — to process all outstanding packets queued in an inbound channel in a *batch*, as illustrated in Fig. 2(b). When applied to all actor tasks and coupled with vectored I/O support in `tokio` when sending to network connections, we show our experimental results on our `Intel` desktop in Fig. 6(b): *batching* improves throughput by up to 186%, and reaching a throughput of 3.9 Gbps for a single flow on the Intel desktop.

Bypassing the user space with `splice` system calls. As we target extreme levels of packet-forwarding performance, we refactored the dataplane to support two different operating modes: *normal* and *max*. In normal mode, packets are processed and forwarded entirely in the user space to maximize the flexibility of implementing new algorithms, with routing policies governed by the controller, and with a repertoire of scheduling, packet dropping, and traffic shaping disciplines. In contrast, in the *max* mode that seeks to maximize performance, packets are directly forwarded to the next hop by *"connecting"* two TCP connections in the Linux kernel, using the `splice` syscall. As we illustrate in Fig. 2(c), the `splice` syscall is designed to move data between two file descriptors without copying between kernel and user address spaces. By setting up a pipe within the kernel space, two `splice` syscalls can be made to connect two TCP connections via such a pipe, eliminating context switching all together during packet forwarding.

But how does such a TCP splicing strategy perform? Cloudflare studied the performance of using `splice` syscalls, and made the surprising observation that TCP splicing *outperformed* the use of eBPF and `SOCKMAP` [37], known for kernel-level performance. Inspired, we conducted our own experiments and our results are shown in Fig. 6(c). It is evident that *extreme* levels of performance can be achieved using TCP splicing: on our consumer-grade `M3` notebook, 132.1 Gbps can be achieved with two hops only, which even slightly outperformed Mininet with kernel packet switches. With more hops on the path, TCP splicing offers slightly lower throughput than Mininet, with the exception of cases



(a) TUN interface: maximizing performance. (b) Processing packets in batches. (c) Bypassing the user space using `splice`.

Fig. 6: Towards maximizing *Nextmini*'s performance: several noteworthy highlights.

where more than 16 hops are involved — the default Mininet switch supports a maximum of 16 hops only. As we expected, operating in the *max* mode with TCP splicing offers performance improvements by an order of magnitude, but trades off some flexibility: as two TCP connections must be "connected" within the kernel, our repertoire of scheduling, packet dropping, and traffic shaping disciplines is no longer applicable. To mitigate the negative impact of such a tradeoff, our design allows normal-mode and max-mode flows to coexist in the dataplane.

Supporting the *max* mode in *Nextmini* brought us an intriguing inspiration: can we support real-world external TCP traffic, in addition to application workloads and synthetic user-space flows? As an encore to our end game on performance, we turned the *Nextmini* dataplane into a `SOCKS5` proxy server: any real-world distributed application — such as a web client and server — can send its traffic through *Nextmini*'s dataplane, by obliviously treating it as any other `SOCKS5` proxy. When coupled with the *max* mode, we are no longer bound by the achievable throughput of a TUN interface, and can achieve well north of 100 Gbps in throughput. In fact, this is how we conducted our benchmarking experiments in Fig. 6(c).

Maximizing scalability on a single physical machine. Mininet was originally conceived to work as a "network in a laptop" [19], which offered outstanding convenience and usability. It also achieves excellent scalability by virtualizing network namespaces only, rather than virtualizing all resources, including CPU cores (`cgroups`) and storage volumes (`chroot`), as in the case of using Docker containers. The use of container orchestration tools, such as the `docker compose` command, is convenient on a single machine, but not necessarily scalable with respect to the memory footprint it takes to run all the containers concurrently. To inherit Mininet's excellent design on single-machine scalability, we have implemented the support for virtualizing network namespaces only in *Nextmini* as an alternative to its users. Similar to Mininet, in such a *namespace* mode — shown in Fig. 3(b) — *Nextmini* creates isolated network namespaces for each dataplane node (running in its own process), as well as a network bridge and virtual Ethernet (`veth`) pairs to enable communication between these namespaces.

To evaluate the scalability of operating in such a *namespace* mode, we conducted a series of experiments on our `Intel` desktop with both Mininet and *Nextmini*, measuring achievable throughput between a `curl` web client and a Python-powered web server, while varying the number of hops on the path in between. From the perspective of throughput, as

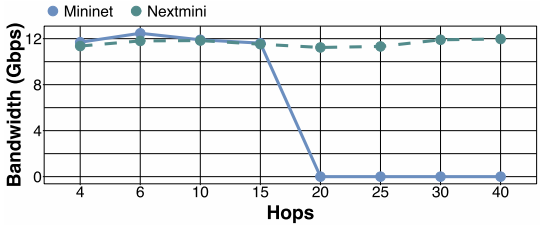


Fig. 7: Achieved throughput as measured by the `curl` client.

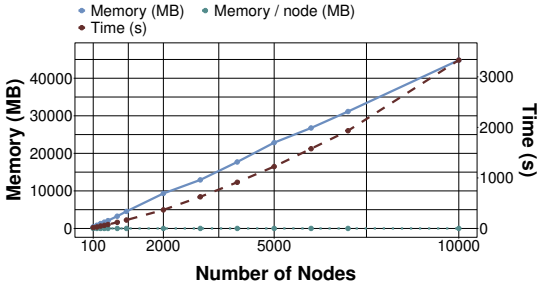


Fig. 8: Total memory footprint, per-node memory, and startup time as we scale up the number of dataplane nodes in *Nextmini*'s *namespace* mode.

we show in Fig. 7, both Mininet and *Nextmini* achieved very similar throughput of around 12 Gbps, and no performance degradation occurred as we scale up the number of hops on the path. However, much to *Nextmini*'s advantage, Mininet's default kernel-space packet switch can only support up to 16 hops, whereas *Nextmini* can scale all the way up to 40 hops without performance degradation.

From the perspective of the memory footprint and initialization time, we show in Fig. 8 that, much to our surprise, both scale predictably as the number of dataplane nodes increases in *Nextmini*'s *namespace* mode. These results are obtained on a dual-socket server with two AMD EPYC 9554 processors (64 cores / 128 threads each). Total memory scales linearly, while the per-node footprint remains nearly constant between 4.2 and 4.7 MB. As a result, a single machine can accommodate up to 10,000 namespace-based nodes with less than 45 GB of memory. Startup time also grows roughly linearly, taking 166 s for 1,000 nodes and 3,332 s for 10,000 nodes (about 0.33 s per node), offering a practical single-command way to construct very large topologies on one machine.

V. Deploying Nextmini in Practice

Thus far, we have evaluated several design highlights in *Nextmini*, predominantly in the same physical machine. In

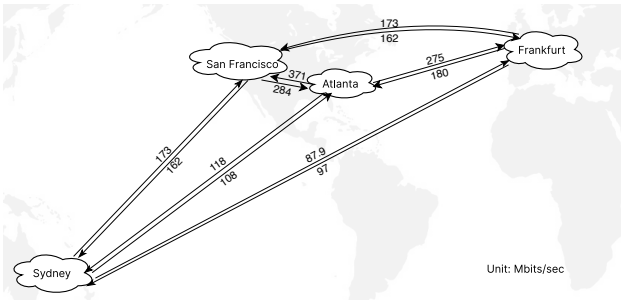


Fig. 9: Deploying *Nextmini* across geographically distributed datacenters.

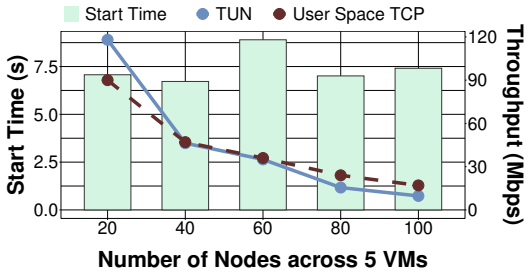


Fig. 10: Deploying *Nextmini* using Docker swarm across 5 VMs in the same cluster.

practice, however, thanks to its use of Docker containers and container orchestration such as Docker swarm, *Nextmini* can be deployed, without much fanfare, across multiple physical or virtual machines in the same datacenter, or even across multiple geographically distributed datacenters. Fig. 9, for example, shows a live deployment of *Nextmini* using Docker swarm across four datacenters around the world, as well as the achievable throughput between these datacenters, as measured live.

Deploying *Nextmini* across VMs in the same datacenter. The most straightforward way of deploying *Nextmini* across multiple VMs in a datacenter is to use Docker swarm, a part of the Docker engine. A swarm cluster can easily be constructed by assigning one VM as the swarm manager, and additional VMs as worker nodes. An immediate benefit of initializing a swarm cluster is that it supports multi-host networking, which implies that an *overlay* network is constructed across all containers in the swarm, and the swarm manager automatically assigns private IP addresses to containers on the network. This eliminates the need for *Nextmini* to manually configure these addresses for a large number of nodes.

Fig. 10 shows our measurement results of running a series of experiments with an increasing number of dataplane nodes — from 20 up to 100 — across a cluster of 5 VMs, each with 16 CPU cores, in a datacenter located in Victoria, BC. Quite surprisingly, it takes less than 10 seconds to start all these nodes using Docker swarm. As expected, the throughput achieved in a simple chain topology involving all the nodes degrades as the number of nodes in the topology increases. Similar throughput can be achieved whether traffic comes from the TUN interface or synthetic user-space TCP flows, showing that the inter-VM bandwidth has been the performance bottleneck.

Deploying *Nextmini* across geographically distributed datacenters. Thanks to *Nextmini*'s user-space design and the

Fig. 11: Running a simple distributed machine learning workload in *Nextmini*, using a Docker swarm across 4 datacenters.

power of Docker swarm for container orchestration, it is straightforward to deploy *Nextmini* across multiple datacenters that are globally distributed, as we have shown in Fig. 9, with an overlay topology consisting of persistent TCP or QUIC connections. As each dataplane node exposes a TUN interface that supports arbitrary application workloads, distributed applications can run without any changes using the virtual network that *Nextmini* exposes, as if it is just a private network within the same cluster.

To show evidence that this is the case, we borrowed a simple Python example on training deep learning models with distributed data parallelism from the PyTorch tutorial, without any modifications to the script. We then executed the script directly within the Docker containers where *Nextmini*'s dataplane nodes are running, and the script ran without a hitch. In Fig. 11, we show the amount of time needed to run one iteration of the training workload, across 5 different convolutional neural network models. The per-iteration training time is inevitably longer due to the limited amount of bandwidth available between datacenters. As an example with the LeNet-5 model, it took 0.08 seconds on average to train one iteration if all four nodes are co-located on the same physical machine (no GPUs were used). In stark contrast, it took 2.24 seconds on average to run the same training workload across our four geographically distributed datacenters, a 28× decrease in performance due to the lack of inter-datacenter bandwidth.

VI. Concluding Remarks

One year in the making of *Nextmini*, it has evolved into a thoroughly modern network emulation testbed that is custom-tailored for the *user space*, fueled by Rust's state-of-the-art support for stackless coroutines, multi-threaded runtime executors, and asynchronous programming with the actor model. Its core design — going for user-space flexibility first and then strong performance using the `splice` system call in Linux — worked out surprisingly well, a pleasant surprise of "having your cake and eating it too." While delivering stellar performance, excellent single-machine scalability, and flexibility due to its user-space design, *Nextmini*'s architecture based on the actor model is also conceptually simple: with only around 9000 lines for its dataplane, 3200 lines for its controller, and a few dozens of megabytes for its memory footprint. We hope that it will become a strong candidate in the community as an emulation and experimentation testbed for new networking research — much needed in the era of artificial intelligence.

References

[1] x.ai, "Announcing Grok," https://x.ai/, Nov. 2023.

[2] J. Pezzone, "Zuckerberg and Meta set to purchase 350,000 Nvidia H100 GPUs by the end of 2024," https://www.techspot.com/news/101585-zuckerberg-meta-set-purchase-350000-nvidia-h100-gpus.html, Jan. 2024.

[3] W. Won, T. Heo, S. Rashidi, S. Sridharan, S. Srinivasan, and T. Krishna, "ASTRA-sim2.0: Modeling Hierarchical Networks and Disaggregated Systems for Large-model Training at Scale," in *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2023, pp. 283–294.

[4] A. Shah, V. Chidambaram, M. Cowan, S. Maleki, M. Musuvathi, T. Mytkowicz, J. Nelson, O. Saarikivi, and R. Singh, "TACCL: Guiding Collective Algorithm Synthesis using Communication Sketches," in *Proc. 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX, 2023, pp. 593–612.

[5] G. F. Riley and T. R. Henderson, "The ns-3 Network Simulator," in *Modeling and Tools for Network Simulation*. Springer, 2010, pp. 15–34.

[6] A. Varga, "A Practical Introduction to the OMNeT++ Simulation Framework," in *Recent Advances in Network Simulation: The OMNeT++ Environment and its Ecosystem*. Springer International Publishing, 2019, pp. 3–51.

[7] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible Network Experiments using Container-based Emulation," in *Proc. 8th International Conference on Emerging Networking Experiments and Technologies (CoNeXT)*. ACM, 2012, pp. 253–264.

[8] "Open vSwitch: An Open Virtual Switch," http://openvswitch.org/, Jul. 2025.

[9] "Swarm mode | Docker Docs," https://docs.docker.com/engine/swarm/, Jul. 2025.

[10] "Kubernetes: Production-Grade Container Orchestration," https://kubernetes.io/, Jul. 2025.

[11] B. Lantz and B. O'Connor, "A Mininet-based Virtual Testbed for Distributed SDN Development," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 365–366, Aug. 2015.

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

[13] "The Go Programming Language," https://go.dev/, Jan. 2024.

[14] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast Packet Processing with eBPF and XDP: Concepts, Code, Challenges, and Applications," *ACM Comput. Surv.*, vol. 53, no. 1, pp. 1–36, 2020.

[15] Wikipedia, "Actor Model," https://en.wikipedia.org/wiki/Actor_model, Jan. 2022.

[16] C. Hewitt, P. Bishop, and R. Steiger, "A Universal Modular ACTOR Formalism for Artificial Intelligence," in *Proc. the 3rd International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann Publishers Inc., 1973, p. 235–245.

[17] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: An Overlay Testbed for Broad-Coverage Services," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 3, p. 3–12, Jul. 2003.

[18] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," in *Proc. 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Dec. 2002, pp. 255–270.

[19] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proc. 9th ACM Workshop on Hot Topics in Networks (HotNets)*, Oct. 2010.

[20] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.

[21] L. Yan and N. McKeown, "Learning Networking by Reproducing Research Results," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 2, p. 19–26, May 2017.

[22] "Containerlab," https://containerlab.dev/, Jul. 2025.

[23] "FRRouting Project," https://frrouting.org/, Jul. 2025.

[24] "Containernet," https://containernet.github.io/, Jul. 2025.

[25] "Kubernetes Network Emulation," https://github.com/openconfig/kne/, Jul. 2025.

[26] D. Weber and J. Fischer, "Process-Based Simulation with Stackless Coroutines," in *Proc. 12th System Analysis and Modelling Conference*. ACM, 2020, p. 84–93.

[27] N. Matsakis, "Async-await on Stable Rust! ," https://blog.rust-lang.org/2019/11/07/Async-await-stable.html, Nov. 2019.

[28] Tokio, "Tokio: Build Reliable Network Applications without Compromising Speed." https://tokio.rs/, Jan. 2024.

[29] S. Klabnik, C. Nichols, C. Krycho, and Rust Community, "Fearless Concurrency," https://doc.rust-lang.org/book/ch16-00-concurrency.html, Jul. 2025.

[30] B. Lantz, "Bad TCP SYN packets generated on veth interfaces in Ubuntu 16.04," https://github.com/mininet/mininet/issues/653, Aug. 2016.

[31] I. Fette and A. Melnikov, "The WebSocket Protocol," Tech. Rep., 2011.

[32] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[33] A. Ryhl, "Actors with Tokio," https://ryhl.io/blog/actors-with-tokio/, Feb. 2021.

[34] J. Lamping and E. Veach, "A Fast, Minimal Memory, Consistent Hash Algorithm," 2014. [Online]. Available: http://arxiv.org/abs/1406.2294

[35] M. Krasnyansky, M. Yevmenkin, and F. Thiel, "Universal TUN/TAP device driver," https://docs.kernel.org/networking/tuntap.html, Jul. 2025.

[36] "Segmentation Offloads," https://docs.kernel.org/networking/segmentation-offloads.html, Jul. 2025.

[37] M. Majkowski, "SOCKMAP - TCP splicing of the future," https://blog.cloudflare.com/sockmap-tcp-splicing-of-the-future/, Feb. 2019.