Rattan: An Extensible and Scalable Modular Internet Path Emulator

Yixin Shen*
Tsinghua University
Beijing, China
syx@ieee.org

Minhu Wang*
Tsinghua University
Beijing, China
minhuw@acm.org

Bo Wang†
Tsinghua University
Beijing, China
wangbo2019@tsinghua.edu.cn

Haixuan Tong
Tsinghua University
Beijing, China
thx24@mails.tsinghua.edu.cn

Chenghao Li
Tsinghua University
Beijing, China
ch-li21@mails.tsinghua.edu.cn

Yutong Xie
Tsinghua University
Beijing, China
xieyt26@mails.tsinghua.edu.cn

Yixuan Gao
Tsinghua University
Beijing, China
gyx@ieee.org

Mingwei Xu†
Tsinghua University
Beijing, China
xumw@tsinghua.edu.cn

Jianping Wu
Tsinghua University
Beijing, China
wjp@tsinghua.edu.cn

Abstract

Modern transport and application innovations require evaluation over fast, heterogeneous, and dynamic Internet paths. Existing emulators do not simultaneously provide the expressiveness to model complex paths, the efficiency for high-throughput and concurrent experiments, and the scalability for large evaluation campaigns. We present Rattan, a modular Internet path emulator that separates path expression from execution. Users compose cells, each implementing one path effect, into channel graphs. A graph-aware runtime executes these graphs using metadata-oriented packet I/O and graph-aware scheduling, while a manager and orchestrator provision isolated channels across machines. Our evaluation demonstrates significant throughput improvement, efficient execution of complex path compositions, and dense concurrent emulation with stable results. A multipath case study shows that Rattan can reveal changes in performance rankings missed by other emulators. Deployment at a large CDN provider further demonstrates sustained pre-deployment evaluation and production extensibility. After integration into its workflow, the average congestion-control rollout cycle decreased from 24 weeks to 18 days.

CCS Concepts: • **Networks** → **Network performance evaluation**.

Keywords: Network emulation, Internet paths, modularity, reproducibility, transport evaluation

*Equal contributions
†Corresponding author



This work is licensed under a Creative Commons Attribution 4.0 International License.

SIGOPS ATC 2026, Hong Kong, China
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2818-1/2026/11
https://doi.org/10.1145/3828523.3857129

ACM Reference Format:
Yixin Shen, Minhu Wang, Bo Wang, Haixuan Tong, Chenghao Li, Yutong Xie, Yixuan Gao, Mingwei Xu, and Jianping Wu. 2026. Rattan: An Extensible and Scalable Modular Internet Path Emulator. In *The 2026 ACM SIGOPS Annual Technical Conference (SIGOPS ATC 2026), November 15–18, 2026, Hong Kong, China.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3828523.3857129

1 Introduction

Large-scale online services increasingly rely on transport optimizations across the stack to deliver high-quality service over diverse Internet paths, including new transport protocols [34], multipath scheduling [21, 22, 38, 63], congestion control [1, 8, 16, 18, 49, 60], and adaptive bitrate selection [40, 51, 53]. A change that improves one class of paths, however, can regress another, and at large scale even rare degradations affect many users and translate into significant revenue loss [25]. Online A/B testing validates changes on real traffic, but safety confines it to a small fraction of users, so its results are slow to reach significance, inherently biased, and hard to reproduce for diagnosis [50]. Offline emulation is therefore the necessary complement: it exercises real, unmodified implementations under controlled and reproducible path conditions before optimizations are deployed in production or used to support research claims.

Serving this role is becoming progressively harder, because the Internet paths (§2.1) an emulator must reproduce are becoming faster [43], more heterogeneous [37, 57, 62, 65] and more dynamic, with properties drifting over days to weeks [59, 61]. A realistic path is thus no longer a single bandwidth-delay-loss pipe but a layered combination of effects, from asymmetric links and shared bottlenecks to traffic policing and multipath behavior. For example, an emulator may need to reproduce a time-varying mobile path with traffic policing, or a multipath connection over 5G and home broadband in which the two subflows may share a bottleneck. The population of such paths therefore forms a high-dimensional and continually changing space.

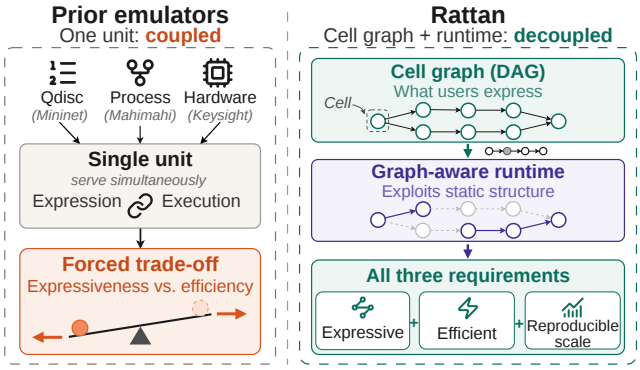


Figure 1. An illustration of the core idea of RATTAN.

This evolution places two demands on emulation (§2.2): *fidelity* to real paths from an end-to-end perspective, and *comprehensive coverage* of the path space with *reproducible and stable* results across runs. The first ensures that offline conclusions remain predictive of real-world outcomes, while the latter enables broad, comparable, and reliable evaluation for deployment decisions. For example, a large-scale video service evaluating a multipath scheduler for users combining 5G and home-broadband access must faithfully capture interactions between the two paths, efficiently sweep real-world diverse network and workload configurations, and reproduce experiments consistently enough to distinguish small performance differences.

At the emulator-design level, these two demands translate into three requirements (§2.3). Fidelity requires **expressiveness and extensibility** of emulators to represent the diverse conditions needed for faithful end-to-end modeling and accommodate new effects as paths and models evolve. Comprehensive coverage requires **execution efficiency**, to explore a broad path space by both sustaining high-throughput paths and running many experiments concurrently under a fixed resource budget. Reproducibility further requires **reproducible scale**, so that large campaigns can spread across many machines while results remain stable across runs.

No existing emulator satisfies all three requirements simultaneously (§2.4). Hardware emulators [29, 32] can reach line rate on specialized ASIC platforms, but their specialization makes fine-grained time-varying conditions, new path models, and non-linear composition hard to support, while their cost and physical rigidity limit large campaigns. Software emulators [35, 42] provide greater flexibility by composing queueing disciplines or manipulating packets in nested processes, but at the cost of efficiency and stability. Moreover, they leave per-packet timing to a general-purpose scheduler, so identical settings drift between runs. These limitations share a root cause: coupling the unit of expression with the unit of execution, whether a queueing discipline, a process, or a fixed hardware block. This coupling creates a structural trade-off: making the unit more flexible tends to increase execution overhead, while specializing it for efficient execution restricts what can be expressed or extended.

We present RATTAN, a modular Internet path emulator that separates these concerns (Figure 1): it *decouples what users emulate from how the emulator executes it, while keeping the structure of the emulation explicit so that execution can exploit it.* The unit of expression is a **cell**, a programmable unit that encapsulates one path effect such as bandwidth control, delay fluctuation, or path selection; users compose cells into a **channel**, an acyclic graph describing one end-to-end path, so a new effect enters as a new cell and a new path structure as a new graph over existing cells. This fine-grained, uniform cell abstraction lets users construct complex path structures and add new effects without specifying how each cell should execute. Rather than executing cells independently, RATTAN executes the cell graph as a whole with a graph-aware runtime that exploits its explicit, static structure and the uniform interfaces between cells.

This separation also creates opportunities for efficient execution. Because common cells determine a packet's fate from metadata alone, RATTAN's virtual NIC (RVNIC) holds each payload once and circulates only descriptors through the cell graph. Because the graph is fixed for a run, the runtime can pass descriptors directly along it in user space instead of returning to a general-purpose scheduler at every hop. And as a campaign reruns the same configuration, the runtime measures per-cell cost during a run and adjusts core placement only between runs, avoiding changes that would perturb the timing of an experiment. On top of the runtime, a manager and an orchestrator provision isolated channels on one machine and dispatch campaigns across many machines, recording each channel as a reproducible artifact. Thus, the same cell graph that gives users a flexible model of the network also gives the runtime explicit structure that it can exploit for efficient and reproducible execution.

Our evaluation shows that RATTAN sustains 14 Gbps on a single emulated path, 8.7× Mininet [35] and 10× Mahimahi [42]; it holds throughput stable as composed effects deepen, where Mahimahi degrades and then fails outright; and it runs 320 concurrent channels near the configured target with low variance, where Mahimahi collapses. A multipath case study shows that channel graphs expose conclusions that linear abstractions miss: whether two subflow paths share a bottleneck can reverse the ranking of innovation variants. RATTAN has also been deployed at a large CDN provider for 21 months, executing 20,000,000 experiments. After integrating RATTAN into its evaluation workflow, the provider observed a reduction in the average congestion-control rollout cycle from 24 weeks to 18 days.

This paper makes the following contributions:

- **A cell-based abstraction for path emulation.** We introduce the cell and channel abstraction for composing built-in and user-defined path effects into end-to-end path graphs, making complex Internet path scenarios easy to express, reuse, and extend.
- **An execution stack for efficient cell-graph emula-**

tion. We design a cell-oriented execution and graph-aware scheduling stack to support high-throughput paths, complex compositions, and dense concurrent channels.
- **A management layer for reproducible scale.** We build a manager and orchestrator that provision isolated resources, reuse experiment setup, record reproducible channel artifacts, and dispatch campaigns across machines for broad, reproducible evaluation.
- **Implementation, evaluation, and deployment experience.** We implement Rattan in roughly 42,000 lines of Rust, evaluate it through microbenchmarks and a multipath case study, and 21 months of CDN deployment.

2 Background and Motivation

2.1 Internet Path Trends

Modern network infrastructure continues to advance, and the end-to-end Internet paths over which transport innovations run are evolving along three axes.

Faster. Internet access speeds have risen as access networks and last-mile deployments improve. Ookla's open dataset [43] reports that from 2019 to 2024 the global median mobile download speed grew from 16.5 Mbps to 55.9 Mbps, while the maximum grew from 804 Mbps to 4798 Mbps. Multi-gigabit access links and tens-of-gigabit aggregation are increasingly common, pushing the data rates that a path must sustain well beyond what earlier evaluation setups assumed.

More heterogeneous. Internet paths are diversifying across the whole stack. At the access layer, users reach services through 5G [4], satellite [41], fiber, and WiFi, each with its own physical characteristics and medium-access behavior. At the network layer, operators apply diverse routing and traffic-management policies [37, 57, 62]; the same nominal service may behave differently across regions or providers, as when one mobile ISP enforces an explicit rate limit that another does not [37, 64, 65]. At the transport and application layers, different rate-control algorithms and traffic patterns shape how flows interact with the network [49, 53]. A single end-to-end path therefore reflects a particular combination of choices at every layer, drawn from a large and growing set of possibilities.

More dynamic. Path properties are not fixed but drift over time, independently of one another. The prevalence of SDN makes it easy for operators to update traffic-engineering or routing policies at a faster pace and finer granularity [30], and measurements report properties shifting over days to weeks [59, 61], with each region, provider, and layer evolving on its own schedule. Meanwhile, transport mechanisms keep proliferating—new protocols [34], multipath scheduling [21, 22, 38, 63], congestion control [1, 8, 16, 18, 49, 60], application-layer adaptation [40, 51, 53]—so the set of behaviors a path may exhibit grows alongside the paths themselves.

Taken together, these trends compound: a path is no longer a simple pipe but a layered combination of fast, heterogeneous effects, and the population of such paths forms a space that is both high-dimensional and in constant motion.

2.2 Validation Demands at Scale

A hyperscale CDN or service provider must confront this high-dimensional and shifting path space when deploying any new innovation. Serving an enormous, globally distributed user base across many applications, it encounters the full diversity of network conditions in practice. A transport innovation that improves performance for one set of users can degrade it for others, and at this scale even a minor performance regression translates into substantial revenue loss [25]. The provider must therefore validate the innovation comprehensively before rolling out any change, to be confident that it is effective and free of unanticipated harm.

Online A/B testing is the gold standard for such validation because it runs real implementations on real paths, but its reliance on real traffic is also its limitation. To stay safe, only a small fraction of users can be exposed to a new change, so results are inevitably biased and slow to reach statistical significance [50]. It also reports only aggregate, statistical outcomes: it reveals that performance changed but says little about *where* or *why*, and it is difficult to reproduce a specific case for diagnosis when a problem does surface.

These limits motivate complementary offline experiments, where environments are controlled and repeatable: experiments can be parallelized to traverse many conditions quickly, cover the space far more comprehensively than live traffic allows, and run with full observability so that pathologies can be reproduced and diagnosed. To be useful for a deployment decision, however, an offline experiment must remain faithful to what the same change would do in the real world; otherwise its conclusions would lose credibility. Offline experiments mainly take two forms. Simulation rebuilds protocols and applications on top of a discrete-event model to gain speed and parallelism, but faithfully reimplementing modern stacks, including congestion control, queueing interactions, Linux-based transports with pacing, application feedback, and so on, is hard, and keeping the reimplementation synchronized with fast-evolving production code is harder still. Emulation instead runs the real implementations over emulated network conditions, naturally inheriting much of the fidelity of real-world testing while retaining the benefits of offline experimentation. **For a hyperscale provider evaluating transport or application innovations, emulation is therefore the necessary complement to A/B testing**.

To iterate on and evaluate an innovation across this high-dimensional, continually evolving path space, emulation must meet two demands. First, the emulated network conditions must maintain high *fidelity* to the real world from an end-to-end perspective; otherwise, evaluations of end-to-end transport or application innovations would diverge from their real-world outcomes. Second, emulation must execute a *comprehensive range* of scenarios to cover the conditions that arise in practice, with *reproducible and stable* results across runs for comparison and to support deployment decisions.

Category (Rep. Work)	Expressiveness	Efficiency	Reprod. scale	Modeling unit	Composition	Exec. unit	Topology
Kernel-based (Mininet [35])	medium	medium	low	qdisc	kernel-fixed	kernel	linear
User-space (Mahimahi [42])	medium	low	low	shell/process	process nesting	process	linear
Hardware (Keysight [32])	low	high	low	impairment block	hardware-fixed	ASIC/FPGA	point-to-point
Rattan	**high**	**high**	**high**	**cell**	**graph**	**cell+runtime**	**any DAG**

Table 1. Categories of existing emulators with representative works, evaluated against the three requirements (§2.3) and characterized by their underlying design choices.

2.3 Requirements on the Emulator

The two validation demands translate into three requirements on the emulator.

The demand for fidelity requires expressiveness and extensibility. Fidelity asks more of what the emulator can express. The network conditions of an emulated path are determined by the network emulation model, and the pursuit of fidelity has continually pushed it forward. Early models [48] described a path with fixed analytic parameters—a constant bandwidth, delay, and loss; later ones replayed measured link traces to reproduce the time-varying behavior of a real link [42]; more recent studies infer richer, data-driven settings based on their more complex network models, approximating what a real path would do [5, 9, 10, 59]. The evolution of network emulation models captures more of a real Internet path, and asks more of the emulator beneath.

A network emulation model, after all, is only as useful as the emulator beneath it: a condition the emulator has no capacity to produce cannot be emulated, however faithfully the model describes it. The emulator's capabilities are thus the ceiling on fidelity, and modern network models want the ceiling to be raised along two axes. First, they demand **expressiveness**: the emulator must be able to articulate the conditions arising in the modern Internet, from high-throughput access links to complex, non-linear path structures, and must *realize* those conditions in real time, sustaining execution at the line rate of the path it represents. Second, as these models evolve with the path space they track, they demand **extensibility**: the emulator must readily admit new effects and host new models without redesign.

The demand for coverage and reproducibility requires execution efficiency and Reproducible scale: Coverage and reproducibility concerns how the emulator executes. To traverse a combinatorial and shifting space affordably, the emulator must execute *efficiently*: under a fixed resource budget, it must run experiments at high density so that broad coverage remains practical. And for the conclusions to be trustworthy rather than merely rapid, execution must be *reproducible at scale*: results must remain stable between runs, and stay so as campaigns are spread across many machines, so that outcomes are comparable and can support a deployment decision.

Together, consolidating the capabilities that the two demands require of the emulator yields the following three requirements:

- **Expressiveness and extensibility**: it must be able to represent the full range of conditions that faithful models describe, and to easily accommodate new ones as paths and models evolve.
- **Execution efficiency**: it must be performant enough to sustain a single high-throughput path, and concurrently run many experiments at high density under a fixed resource budget.
- **Reproducible scale**: it must scale readily across many machines for large-scale parallel execution, and keep its results reproducible regardless of that scale.

2.4 Limitations of Existing Emulators

Existing emulators are based on hardware or software, but neither satisfies all three requirements simultaneously, and each trades one set of requirements for another (Table 1).

Hardware emulators reach line rate but are rigid to extend and difficult to scale. Hardware emulators such as Keysight [32] and HoloWAN [29] emulate network conditions on specialized platforms with ASICs, P4, or FPGAs [19, 52], and thereby attain excellent throughput. Their specialization, however, limits flexibility (i.e., expressiveness and extensibility): they do not support fine-grained, flexible, time-varying network conditions, and new path models and composition logic are hard to implement on fixed hardware. Moreover, their cost and physical rigidity make them ill-suited to large campaigns that demand many concurrent scenarios and repeated runs, leaving scalability out of reach.

Software emulators improve expressiveness and extensibility, but at the cost of efficiency and reproducibility. To attain the flexibility that hardware lacks, a second category emulates network conditions in software, atop a general-purpose operating system and its network stack. Compared with fixed hardware, software offers more general interfaces for flexibly configuring path properties and more readily admits new effects. Kernel-based emulators, such as Mininet [35] and DummyNet [47], compose queueing disciplines to assemble a wide range of network paths with flexible topologies. Yet every packet must traverse the in-kernel network stack and its disciplines multiple times, and this per-packet overhead bounds both the throughput a single path can sustain and the number of paths a machine can host simultaneously. Their predefined queueing disciplines also limit the path effects they can express and do not natively support replaying the fine-grained, time-varying conditions of real-world networks. To overcome this expressive limitation, user-space emulators like Mahimahi [42]

manipulate packets freely and can therefore replay traces that reproduce fine-grained, time-varying conditions. They compose effects, however, by nesting one process within another, which hands each packet across process boundaries; the resulting context switches and copies cap throughput and concurrency more severely still. This nesting is also structurally linear, and thus cannot represent complex non-linear topologies such as multipath. Both, finally, leave the timing of every effect to the general-purpose scheduler. Under CPU contention and scheduling jitter, the same configuration executes with subtly different timing on each run, and these per-packet discrepancies accumulate into results that drift between runs—an instability that worsens with scale, which we quantify in §5.1. Software emulators thus gain expressiveness but forfeit efficiency and reproducible scale.

Root cause: the coupling between expression and execution. Existing designs use one abstraction unit, whether a kernel queueing discipline, an operating-system process, or a fixed hardware block, to determine both what can be *expressed* and how that expression is *executed*. As a result, changing the unit to gain modeling flexibility also changes its execution cost, while specializing it for efficient execution restricts what can be expressed or extended. This creates a structural trade-off between expressiveness and execution efficiency in these designs. The trade-off is therefore not inherent to emulation itself; it follows from coupling the two concerns in the same abstraction unit.

2.5 Basic Idea

Rattan breaks this coupling by separating the *unit of expression* from the *unit of execution*, while keeping the structure of the emulation explicit so that the runtime can exploit it. The unit of expression is a *cell*, a fine-grained programmable unit that encapsulates one path effect. Users compose cells into a *channel*, an explicit acyclic graph describing an end-to-end path. A new effect can therefore be introduced as a new cell, while more complex path structures are constructed by composing cells rather than changing the execution engine.

A natural concern is that this expressive power comes at the cost of performance. A path that a coarser tool runs as a single process becomes, in Rattan, a graph of many small cells, and a naive implementation—each cell as an isolated actor with its own execution context—would make every packet cross too many boundaries to sustain the multi-gigabit rates and many concurrent paths that modern evaluation demands.

Rattan avoids this cost because a cell is an *expression unit, not an independent execution unit*. The runtime executes the channel graph as a whole. Moreover, this runtime need not behave like a general-purpose packet processor: it serves emulation specifically, where the path is defined and controlled by the system, fixed before a run, and often replayed across an evaluation campaign. It therefore need not incur several obligations of a general-purpose system, such as repeatedly moving packet payloads between processing units, scheduling work without prior knowledge of the processing structure, operating over paths it does not control, or providing general-purpose fairness among independent workloads.

These properties give the runtime several opportunities to exploit the structure exposed by the cell graph. First, because the path is defined in advance, the cell graph is explicit and static for the duration of a run; with the whole path laid out before it rather than discovered one cell at a time, the runtime can plan how packets move and where work is placed far more efficiently than any cell could decide on its own. Second, since most cells decide a packet's fate from metadata alone, the runtime can apply effects to the packet's metadata, with only descriptors passed between cells, and defer the actual forwarding until the packet leaves the emulator. Third, with the whole cell graph in view and no fairness requirement, it can impose a deterministic order on how cells run and how packets pass between them, so that the same configuration reproduces the same execution from one run to the next.

Fine granularity therefore does not merely provide flexibility; it also exposes structure that the runtime can exploit for efficiency and reproducibility. The same fine-grained structure that lets cells express what coarser units cannot is what lets the runtime execute them efficiently and stably. Consequently, expressiveness, efficiency, and reproducible scale arise from the same choice rather than trading against each other.

3 Rattan Design

To turn the opportunity that decoupling expression from execution opens into a working emulator, Rattan is organized as a stack that follows the principle directly: users express a path as a **channel**, a directed acyclic graph of **cells**; a customized **runtime** executes that graph; and a **manager** and **orchestrator** provision and dispatch channels on one machine and across many. We first describe how cells and channels deliver expressiveness (§3.1), then how Rattan reclaims efficiency in packet movement and scheduling (§3.2), and finally how the same channel abstraction makes large, reproducible campaigns inexpensive to run (§3.3).

3.1 Cells and Channels: Composing Path Effects

A cell encapsulates one emulation function. Built-in cells cover common effects such as bandwidth limiting, propagation delay, random loss, queueing and AQM, trace replay, traffic policing, and path selection. Users can also implement new cells when an experiment requires a path model that is not already provided, such as a wireless-specific delay model or a proprietary traffic-management policy.

Each cell exposes four interfaces. `enqueue` receives packets from upstream cells. `dequeue` releases packets downstream according to the cell's emulation logic. `control` updates cell parameters at runtime, such as bandwidth, delay, loss rate, or trace position, without changing the channel topology. `telemetry` exposes state, packet traces, and logs

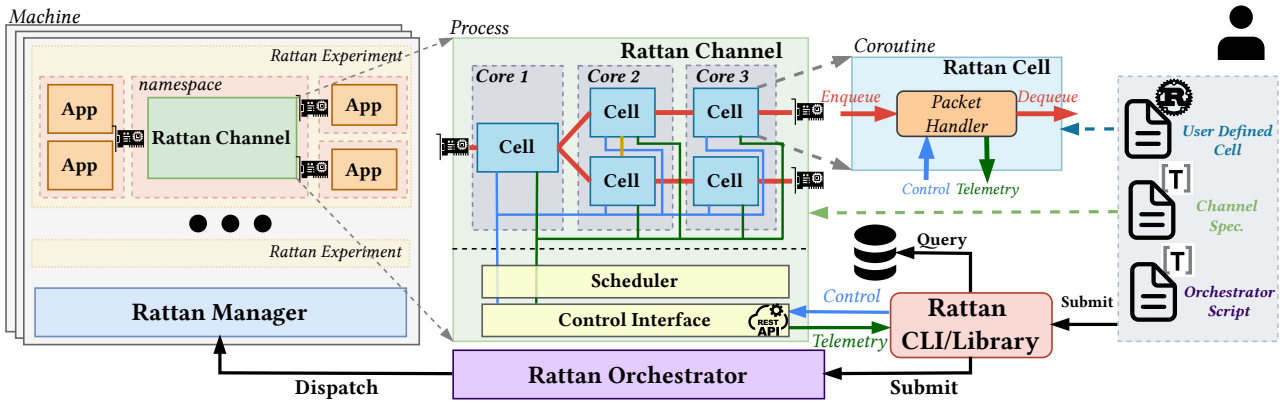


Figure 2. Overview of RATTAN.

```rust
struct DelayCell { internal_queue: VecDeque<P>, timer: Timer, config: DelayCellConfig, stats: DelayCellStats }
impl RattanCell for DelayCell {
    fn enqueue(&self, mut packet: P) { self.internal_queue.push_back(packet) }
    async fn dequeue(&self) -> Option<P> {
        let mut packet = self.internal_queue.pop_front()?;
        packet.set_timestamp(packet.get_timestamp() + self.delay); // advance logical time
        self.timer.sleep_until(packet.get_timestamp()).await;      // release at that instant
        self.stats.update(&packet);
        Some(packet)
    }
    async fn control(&self, config: Self::Config) { self.config.update(config).await; }
    async fn telemetry(&self) -> DelayCellStats { self.stats.collect().await }
}
```

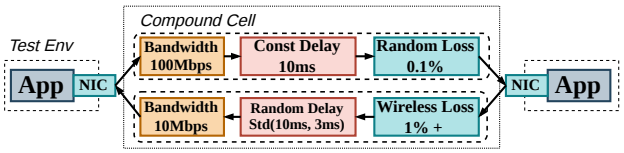
Figure 3. A RATTAN cell that adds a configurable propagation delay to passing packets.

to users and controllers. These interfaces divide the cell into two planes: a data plane that carries packets between cells, and a control plane that handles configuration and measurement messages. Because the two planes are separate, the data plane can be optimized on its own to carry the information a cell needs for its decisions, independently of how control messages are delivered (§3.2). Built-in and user-defined cells use the same interfaces, so new effects compose and schedule in the same way as existing ones. Figure 3 illustrates these interfaces with a simplified delay cell.

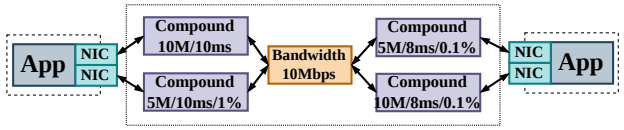
A cell's timing is carried by the packets themselves. Each packet holds a logical timestamp that marks when it next becomes eligible for downstream processing, and a cell advances this timestamp by the delay or service time it imposes. A delay cell advances the timestamp by its configured propagation delay; a bandwidth cell advances it by the transmission time of the packet at the configured rate. The cell then releases the packet at the logical instant the timestamp now designates, and the downstream cell inherits that timestamp as its own starting point. As each effect is accumulated into the timestamp rather than re-derived from the wall clock at each hop, the end-to-end timing of a chain of cells is the composition of the per-cell effects, not an accumulation of per-hop scheduling error. The timestamp defines the target release time; if runtime overload causes actual release times to drift from these targets, RATTAN detects the drift and warns users that the emulation may be inaccurate. This makes it safe to build a complex path from many small cells.

Beyond this timing contract, RATTAN places no restriction on a cell's internal model except that it not block its execution thread. A cell can implement a simple queue, replay a measured trace, call into a learned model, or communicate with an external service, as long as it follows this non-blocking rule, so more specialized cells can model WiFi-layer retransmissions, dynamic capacity allocation, or LEO satellite link dynamics. The same uniformity lets RATTAN build its own machinery from cells: NIC cells connect a channel to virtual or physical interfaces, router cells direct each packet to one of several downstream branches and merge several upstream branches into one, and logging cells record packet streams for later inspection. These four stable public interfaces define the contract for implementing every effect, built-in or user-defined, so developers need only understand this contract and the modeled effect, without knowing or modifying the internals of the runtime, RVNIC, manager, or orchestrator.

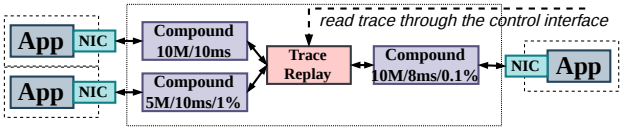
Cells become an emulated path when they are composed into a **channel**, whose data plane is a directed acyclic graph (DAG) of cells with edges representing packet flow. A simple path is a chain of bandwidth, delay, and loss cells. Richer scenarios are expressed by constructing the appropriate graph before a run rather than by adding an ad-hoc mechanism: asymmetric uplink and downlink paths, shared bottlenecks between otherwise separate paths, mixtures of analytic link models and trace replay, and multipath or path-selection



(a) An emulated path with asymmetric up and down links.



(b) An emulated multipath condition with a shared bottleneck.



(c) An emulated path combining link emulation and trace replay.

Figure 4. Examples of RATTAN channels demonstrating how cells combine to emulate advanced network conditions. The shared bottleneck in (b) is a single bandwidth cell with two incoming edges, not a special multipath mechanism.

behavior. Figure 4 shows representative channel graphs, all built from the same cell abstraction; time-varying conditions are handled separately through `control` messages that update parameters inside the existing graph. For example, path switching can be represented by a router cell selecting among predeclared branches, while time-varying bandwidth, delay, and loss are represented by changes within individual cells. These changes affect packet behavior without adding or removing cells or edges.

From the application's perspective, a channel appears as a network path between virtual or physical NICs. Unmodified client and server applications run in their own network namespaces and communicate through these NICs without depending on RATTAN internals, which also permits hybrid experiments in which one side of a channel is an emulated path and the other forwards traffic to a physical NIC. Channels can be built programmatically through the RATTAN library or declaratively from a configuration file that lists cells and links.

The channel graph is the central difference from prior emulator abstractions. NetEm-style queueing disciplines and Mahimahi-style process nesting can stack effects along one path, but they do not directly represent a point where two independent paths converge and contend for one queue. In RATTAN, that shared bottleneck is simply a bandwidth cell with two incoming edges: two upstream sub-paths, each built from ordinary cells, feed one internal queue where the flows contend (Figure 4b). Expressiveness in RATTAN thus has two complementary sources: a new path effect is introduced as a new cell, and a new path structure as a new graph over existing cells. Both are realized through the same interfaces, extending the emulator by composition rather than by special-casing its execution engine.

3.2 The Cost of Modularity, and How We Pay It

Composing a path from many small cells is what gives RATTAN its expressiveness, but a naive implementation would pay for it in performance: every packet now crosses many cell boundaries, and the runtime must decide when to run each cell. At the throughputs and channel densities modern evaluation demands, these costs would dominate. RATTAN drives both costs down by exploiting three properties a general-purpose packet processor cannot assume: most emulation effects require no payload bytes, the cell graph is fixed for the duration of a run, and a campaign reruns the same configuration repeatedly.

Packet movement: RVNIC. The first cost is data movement. A general forwarding fast path must preserve every packet's payload, so handing a packet from one cell to the next risks copying its bytes at each hop. Common path-emulation effects do not need those bytes: common bandwidth, delay, loss, and queueing cells decide when to release a packet, whether to drop it, and how far to advance its timestamp using only the packet's length, selected headers, and timestamp. RATTAN exploits this by separating a packet's *descriptor* from its payload. When a packet arrives, the payload is held once, and what flows along the edges of the cell graph is a small descriptor carrying the fields needed by ordinary cells plus a token that identifies the held payload. Cells enqueue, delay, and dequeue descriptors; dropping a packet is pushing its token to a drop list. If a specialized cell needs payload data, it uses the token to fetch only the byte range it inspects, instead of forcing every upstream and downstream cell to move the full packet. The payload is reattached only when the packet leaves the channel. A path of N cells therefore moves N small descriptors and copies payload bytes only on demand, making the common per-hop cost independent of packet size. This is the concrete payoff of the control–data separation in the cell interface (§3.1): because packet handoff is a dedicated path, it can be specialized for descriptor movement while still allowing payload access when a cell explicitly requires it.

RATTAN realizes this split in its own RATTAN virtual NIC (RVNIC). RVNIC adopts the memory and ring model of AF_XDP [54]: user space and the kernel share a pre-registered packet buffer pool and communicate through lock-free producer/consumer rings, so packets cross the kernel boundary without per-packet system calls or full-packet copies. Unlike AF_XDP, which attaches to an existing interface, RVNIC is implemented as a dedicated kernel NIC driver, which lets it expose exactly the rings an emulator needs: alongside the usual receive, transmit, fill, and completion rings, it adds a drop ring, so a cell that discards a packet returns the buffer directly without copying its payload into the cell graph. The kernel retains each packet and hands user space only its descriptor and token; when a cell explicitly reads payload, RVNIC copies only the requested bytes. The descriptor-first data path therefore extends from the wire through the cell

graph, with payload movement limited to the cells that actually need it.

Cell scheduling. The second cost is scheduling. Running many fine-grained cells is affordable only because a cell is a coroutine rather than a thread or a process: thousands of coroutines can be multiplexed onto a few cores, a switch between them costs a function call rather than a kernel context switch, and a cell waiting to release a packet suspends without occupying a thread. This is already far cheaper than running each path effect as a separate process, as a shell-based emulator must, or as a queueing discipline confined to the kernel. RATTAN builds on a coroutine runtime for this reason, and then specializes it to reduce the overhead that a general-purpose runtime retains.

This residual overhead comes from generality: a general runtime assumes nothing about how its tasks relate, so after a cell finishes a packet, control returns to a scheduler that polls its ready set to find the next task to run, and a packet crossing N cells incurs up to N such handoffs. RATTAN avoids them because it knows what a general runtime does not: the cell graph is fixed for the run, so the cell that runs after a given cell releases a packet is known in advance. Its runtime therefore follows the graph directly. When a cell releases a packet, the runtime passes the descriptor to the downstream cell and continues executing along the graph in user space, reducing the per-cell handoffs to a single traversal; it returns to the general scheduler, through a single `epoll` wait, only when a channel has no runnable work and must wait for the next packet or external timer. Within a channel, cells are ordered by the logical timestamps of the packets they hold, so the runtime always advances the packet that is due next.

Spinning along the graph in user space maximizes the throughput of one channel but occupies cores, which is the wrong trade-off for the hundreds of low-rate channels a broad evaluation also needs. RATTAN therefore lets a channel either hold a dedicated core and spin, for high-throughput paths, or share a core with many others and yield through `epoll` whenever it is idle, for low-rate paths that spend most of their time waiting. The harder question is which cells and channels to place on which cores. The runtime cannot read per-cell CPU cost off the configuration: a trace-replay or learned-model cell can cost far more than a constant-rate link of the same nominal bandwidth, so a placement computed from configured rates alone risks overloading some cores while others idle. Reacting to the imbalance during a run is worse: detecting an overloaded core and migrating a cell takes time, and during that time the runtime has already added scheduling delay to the packet timings the experiment is meant to measure, while the migration itself perturbs them further. Online re-balancing thus distorts the very measurement it is meant to improve.

RATTAN resolves this with a property no general scheduler can assume: an emulation campaign often runs the same channel configuration repeatedly, so a good placement is worth computing once and reusing. A cell's computational cost is largely determined by its processing logic and, for trace-driven cells, the replayed trace, so measurements from one run provide a useful basis for placement in subsequent runs of the same configuration. During a run, the runtime only records the realized peak CPU cost of each cell and core but never migrates, so the run itself is never perturbed. The same measurements double as a fidelity check: the runtime tracks each cell's scheduling delay, the gap between when it becomes runnable and when it actually runs, and flags any run in which this delay exceeds a threshold or packet timestamps drift beyond the configured tolerance. Such runs are discarded rather than reported as valid results. Between runs, if any core was overloaded, RATTAN uses these observed peak costs as graph-partitioning weights and assigns the newly partitioned subgraphs to CPU cores. The manager records this placement keyed to the exact channel configuration (§3.3). Subsequent runs of the same configuration start from the recorded placement; if the workload CPU cost changes enough to overload a core again, the failed run is discarded and RATTAN computes a new placement for the next run. This converts an online scheduling problem, which a general runtime would have to solve reactively and at the cost of fidelity, into an offline one, giving RATTAN both the stability of a fixed placement and the accuracy of one derived from measured cost.

3.3 Managing and Scaling Channels

Because a channel is already an isolated, self-contained graph, scaling emulation from one machine to many is a matter of provisioning and dispatching channels. Two components handle it: a *manager* that provisions channels on a single machine, and an *orchestrator* that dispatches campaigns of channels across many.

Provisioning and isolating channels: the manager. The manager turns a channel configuration into a running channel on one machine. It first creates the network environment the channel needs: network namespaces, virtual interfaces, and the routing and firewall state that lets unmodified client and server applications send traffic through the emulated path. It then constructs the cells, starts the channel runtime, and exposes each cell's `control` and `telemetry` interfaces as REST endpoints, through which users and higher-level controllers reconfigure cells and read their state while the channel runs. Because successive experiments often share much of this environment, the manager reuses provisioned resources when isolation allows, so channel setup is paid for once rather than per run.

Provisioning makes each channel a reproducible artifact. The manager records the channel graph, cell parameters, workload commands, random seeds, core placement, namespace and NIC-queue allocation, runtime logs, and validity flags. On a rerun, it reconstructs the same local environment and synchronizes time-dependent cells before traffic begins.

This makes the emulated *path* reproducible across repeated runs; it does not make the system under test deterministic, so a congestion-control algorithm may still react differently to identical path conditions. What RATTAN guarantees is that such differences come from the system under evaluation rather than uncontrolled variation in the emulator.

Dispatching campaigns: the orchestrator. A broad evaluation is a sweep of many channels over path settings, workloads, optimization configurations, and random seeds, often run across a cluster. The orchestrator separates the intent of such a campaign from the logistics of running it. Users submit tasks that record the experiment properties needed to reproduce a run, such as the channel description, workload, required worker capabilities, and query labels. Workers register with the orchestrator and advertise their capabilities, and the orchestrator stores submitted tasks persistently, places ready tasks onto matching workers, and lets workers poll for work rather than maintaining long-lived execution sessions. For a RATTAN task, the selected worker invokes its local manager to provision and run the channel, then uploads results and logs before committing the task's final state. Heartbeats let the orchestrator detect failed workers and return unfinished tasks to the ready pool. Because task specifications, worker identity, state transitions, and artifacts are stored as structured records, a campaign can be monitored while running, queried after completion, and rerun from the same experiment description.

4 Implementation

RATTAN is implemented in roughly 42,000 lines of Rust and 1,500 lines of C (the RVNIC kernel driver), and supports Linux kernels from 5.4 to 6.17. RATTAN is open source[1], with all configurations needed to reproduce the experiments in this paper available[2]. Users interact with it through two entry points: a campaign interface that submits evaluation campaigns to a worker cluster through the orchestrator, and a single-binary emulator for one-off local experiments and debugging. The local emulator defaults to fully isolated namespaces for the server, client, and emulator, and also provides a Mahimahi-style shell mode [42] that eases the migration of existing Mahimahi experiment scripts. The rest of this section describes several notable implementation details.

Cell control and observability. Control and observability are exposed through two mechanisms: an Axum REST server realizing the `control` and `telemetry` interfaces, and an optional per-packet log for post-run analysis. The REST server can reconfigure any cell at runtime, supporting interactive debugging where reconfiguration can be triggered externally upon reaching a specific condition (e.g., slow-start exit). The optional per-packet log records the action applied (e.g., forward, delay, drop) and the cell's state (e.g., queue length), with compact events (e.g., 32 bytes) keeping overhead low.

[1]Open sourced at: https://github.com/stack-rs/rattan

[2]Artifact: https://doi.org/10.5281/zenodo.23267891

Realizing RVNIC across the kernel boundary. RVNIC consists of two components: a Linux kernel module that registers one virtual NIC per opened `/dev/rvnic`, and a Rust user-space library `librvnic` that wraps this device interface for cells. The shared buffer pool and five rings introduced in §3.2 are `mmap`-allocated in user space and pinned by the kernel on registration, while the full `sk_buff` stays kernel-side in a token-indexed hash table and is freed only when its token returns through the drop or TX ring. To amortize wakeup cost, RVNIC batches RX notifications and wakes user space once every 32 packets or 64 μs.

Custom async runtime. RATTAN's custom async runtime passes each packet descriptor following the cell graph in user space until blocked on a cell's timer, without returning to a scheduler. It then waits for either the next timer to fire or RVNIC to receive a packet, using a hierarchical timing wheel augmented by a min-heap of upcoming timers to read the next deadline in constant time. The runtime yields through a single `epoll_wait` when the next deadline falls beyond a configurable threshold, or otherwise spins until the deadline arrives. Raising the threshold favors spinning, sacrificing channel density for reproducibility.

Manager and orchestrator. The manager is a per-host daemon that provisions namespaces, RVNIC pairs, and per-channel `nftables` chains. It reuses shared environments across runs and records each run under a content-addressed directory. The orchestrator dispatches tasks to capability-matched workers via a lease-renewed state machine, reschedules runs flagged invalid by the runtime's drift check, and triggers repartitioning of cells across cores based on the measured costs for subsequent runs of that configuration.

5 Evaluation

We organize our evaluation around three questions:

- **Q1 Performance:** Does RATTAN sustain high forwarding throughput, handle complex path compositions efficiently, and support dense concurrent emulation? (§5.1)
- **Q2 Expressiveness:** Can RATTAN express realistic path scenarios and reveal conclusions that simpler emulator abstractions may miss? (§5.2)
- **Q3 Practicality:** Is RATTAN practical for real deployment and large-scale evaluation workflows? (§5.3)

5.1 Microbenchmarks

We run all microbenchmark experiments on one machine with an Intel Core i7-10700F CPU, 64 GiB of memory, and Debian 13 with Linux kernel 6.12.88.

Single-channel forwarding throughput. We first measure the forwarding throughput that each emulator can sustain on a simple bandwidth-delay path. We compare RATTAN with Mahimahi [42] and Mininet [35] by configuring an equivalent bidirectional path in each system: a RATTAN channel with bandwidth and delay cells, a Mahimahi shell stack using `mm-link` and `mm-delay`, and a Mininet topology using Linux HTB and NetEm. Each direction is configured with the same

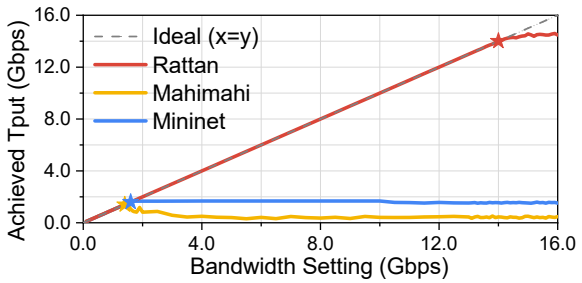


Figure 5. Single-channel forwarding throughput on a 50 ms RTT path with one TCP CUBIC flow (shaded bands show the standard deviation across trials). The highest reliably achievable throughput is marked as the stars on the figure.

25 ms delay, and we sweep the bottleneck bandwidth from 0.1 Gbps to 16 Gbps in 0.1 Gbps increments. RATTAN and Mahimahi use infinite queues, while Mininet uses a Linux qdisc queue limit of 10^7 packets, large enough to approximate an infinite queue. Each emulator is pinned to six CPU cores. For each bandwidth setting, we run five 10-second `iperf3` trials with a single long-lived TCP CUBIC [26] flow, discard the slow-start period as warm-up, and report the mean and standard deviation. The sender and receiver can exceed the tested rates without emulation, so throughput is bounded by the emulator rather than the workload endpoints.

Figure 5 reports the achieved throughput. RATTAN tracks the configured bandwidth through 14.0 Gbps and then plateaus around 14.5 Gbps. In contrast, Mininet follows the target only up to about 1.6 Gbps and then saturates around 1.6 Gbps, while Mahimahi reaches 1.4 Gbps at low settings but drops below 1 Gbps once the configured bandwidth reaches 1.7 Gbps. At its peak, RATTAN achieves 14.0 Gbps reliably, which is 8.7× Mininet's 1.6 Gbps and 10× Mahimahi's 1.4 Gbps. These results show that RATTAN sustains multi-gigabit forwarding throughput for a single emulated path, while the baselines are limited by kernel processing overheads or process-based packet handoff cost.

Path complexity. We next measure how each emulator scales as an emulated path includes more composed effects. We increase path complexity by repeatedly appending a loss, bandwidth, and delay sequence, configured with 0% loss, 1 Gbps bandwidth, and 1 ms delay. Thus, a path with complexity N contains N such sequences, placing more emulator components on the path while keeping the target throughput fixed. Each emulator is pinned to four CPU cores. The workload and measurement procedure are the same as in the forwarding-throughput experiment.

Figure 6 shows that RATTAN remains stable as the number of composed effects grows. RATTAN sustains the 1 Gbps target through complexity 22 and then degrades, still achieving 0.970 Gbps at the deepest chain we test. Mahimahi degrades much faster: it starts at 0.74 Gbps with complexity 2 and drops to 0.20 Gbps by complexity 15. Beyond complexity 15, Mahimahi fails to complete the experiment. Mininet follows the target at low complexity, but its performance begins to drop earlier than RATTAN's as the qdisc chain grows, and remains lower than RATTAN's once the complexity exceeds 8. These results show that RATTAN keeps the cost of modular path composition controlled, while process nesting in Mahimahi and long qdisc chains in Mininet become bottlenecks as path complexity grows.

Density and reproducibility. We next measure how many emulated paths each system can run concurrently under a fixed hardware budget, and whether the resulting paths remain reproducible under load. Each instance is an independent bidirectional bandwidth-delay path configured with 16 Mbps bandwidth and 40 ms RTT, carrying one `iperf3` TCP CUBIC flow. We pin each emulator to eight CPU cores. We vary the number of concurrent instances from 8 to 512. For each concurrency level, we run that many emulator instances in parallel and repeat the experiment 512 times, reporting the mean achieved throughput and standard deviation across trials. The mean captures how much concurrent load the emulator can sustain, while the standard deviation captures reproducibility: an emulator that faithfully provides the configured path should keep achieved throughput close to the target with low variance across repeated trials. We evaluate both static-bandwidth paths and trace-driven paths whose mean bandwidth is also normalized to 16 Mbps. Mininet is omitted from the trace-driven experiment because it does not support the trace-replay model.

As Figure 7 shows, RATTAN sustains dense concurrent emulation with low variance up to 320 instances. In the static-bandwidth experiment, RATTAN stays close to the 16 Mbps target from 8 to 320 concurrent instances, achieving 15.98 Mbps with a standard deviation of 0.017 Mbps at 320. Mahimahi collapses at the same load, dropping to 5.15 Mbps with a standard deviation of 7.43 Mbps, while Mininet remains near the target in this static setting. At 512 instances, RATTAN also becomes resource-limited, but still delivers 14.00 Mbps on average, 3.2 × Mahimahi's 4.37 Mbps. The trace-driven experiment shows the same trend without Mininet. These results show that RATTAN supports substantially denser concurrent emulation than Mahimahi while keeping repeated trials close to the configured path behavior until the machine becomes oversubscribed.

5.2 Benchmark

We use RATTAN to evaluate a multipath transport benchmark that exercises path scheduling, congestion control, and their interactions with realistic path structures. Each run transfers a 50 MiB file over a single MPTCP [22] connection with two subflows, one assigned to each emulated path, using a custom MPTCP application on Linux MPTCP v0.96 kernel. All experiments run on a machine with an Intel Xeon E5-2690 v2 CPU, 128 GiB of memory, and Ubuntu 24.04, with RATTAN and the application pinned to two CPU cores.

We construct four multipath scenarios, denoted A–D, drawn from production conditions and covering shared bottlenecks, trace-driven bandwidth variation, and delay dynamics. Sce-

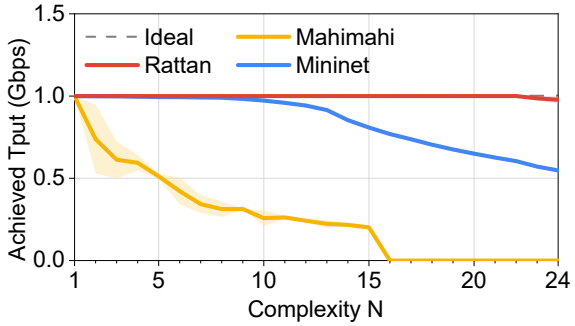
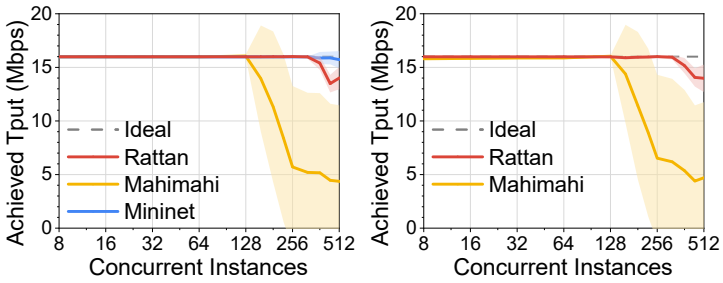


Figure 6. Emulator achieved throughput versus path complexity at a fixed 1 Gbps target. Complexity N corresponds to a path of N chained loss-bandwidth-delay sequences (0% loss, 1 Gbps, 1 ms each). The shaded bands show the standard deviation across trials. Mahimahi fails to complete the experiment beyond complexity 15.



(a) Run with static bandwidth **(b)** Run with trace bandwidth

Figure 7. Emulator achieved throughput versus concurrent instances at a fixed 16 Mbps target under a eight-core CPU budget. Lines report the mean over 512 trials (sustained load) and bands the standard deviation (reproducibility). Each instance is an independent 16 Mbps, 40 ms RTT path carrying one TCP CUBIC flow. Mininet is omitted from (b) as it does not support trace replay.

	Scenario A (Shared, Fixed BW, Low Jitter)			**Scenario B** (Indep., Fixed BW, Low Jitter)			**Scenario C** (Indep., Trace BW, Low Jitter)			**Scenario D** (Indep., Trace BW, High Jitter)		
	minRTT	BLEST	ECF	minRTT	BLEST	ECF	minRTT	BLEST	ECF	minRTT	BLEST	ECF
CUBIC	6.07 ± 0.0	6.18 ± 0.0	6.10 ± 0.0	4.07 ± 0.0	4.23 ± 0.0	4.14 ± 0.1	5.32 ± 0.1	5.48 ± 0.1	5.70 ± 0.2	6.60 ± 0.1	6.81 ± 0.1	7.47 ± 0.3
BBR	6.19 ± 0.1	6.29 ± 0.0	6.15 ± 0.0	3.56 ± 0.1	3.54 ± 0.0	3.52 ± 0.1	4.76 ± 0.1	4.78 ± 0.1	4.52 ± 0.3	4.49 ± 0.0	4.61 ± 0.1	4.45 ± 0.1
Vegas	6.12 ± 0.1	6.32 ± 0.2	**7.57 ± 0.4**	4.58 ± 0.1	4.73 ± 0.2	**6.32 ± 0.7**	5.78 ± 0.1	6.27 ± 0.0	6.74 ± 0.7	9.51 ± 0.7	10.12 ± 0.1	10.64 ± 0.8
LIA	6.16 ± 0.0	6.21 ± 0.0	6.83 ± 0.6	4.56 ± 0.0	4.76 ± 0.0	5.27 ± 0.6	6.68 ± 0.0	6.68 ± 0.0	6.79 ± 0.4	7.74 ± 0.1	7.71 ± 0.2	7.99 ± 0.4
OLIA	6.15 ± 0.1	6.19 ± 0.0	**6.70 ± 0.4**	4.57 ± 0.0	4.76 ± 0.0	**7.88 ± 0.8**	6.88 ± 0.3	7.12 ± 0.1	6.75 ± 0.1	7.71 ± 0.0	7.76 ± 0.1	8.38 ± 0.5
BALIA	6.22 ± 0.0	6.20 ± 0.0	9.22 ± 2.0	4.73 ± 0.0	4.80 ± 0.0	9.97 ± 3.0	**6.83 ± 0.4**	**6.67 ± 0.1**	**6.96 ± 0.6**	**7.63 ± 0.2**	**8.03 ± 0.0**	**7.78 ± 0.2**

Table 2. Flow completion time (s, mean±std over 10 runs; lower is better) of 18 scheduler–CCA combinations under four multipath scenarios. Highlighted cells mark the comparisons discussed in §5.2.

nario A models two heterogeneous paths that merge at a shared bottleneck: path 1 provides 60 Mbps bandwidth, time-varying delay with a 35 ms mean, and 0.1% loss, while path 2 provides 90 Mbps bandwidth, time-varying delay with a 45 ms mean, and no loss; the two paths then feed a 75 Mbps shared bottleneck, and all three bandwidth stages use BDP-sized drop-tail queues. Scenario B removes the shared bottleneck and keeps the two paths fully independent, reducing to the conventional two-path setup. Scenarios C and D instead drive both paths with bandwidth traces and introduce no packet loss. In C, path 1 has a 60 Mbps mean bandwidth with high variance, a shallow buffer, and a fixed 40 ms delay, while path 2 has a 60 Mbps mean bandwidth with lower variance, a deep buffer, and a delay that switches between 22.5 ms and 50 ms every 2 s. Scenario D keeps path 2 unchanged but turns path 1 into a high-jitter path whose delay switches between 22 ms and 56.5 ms every 10 ms. For each scenario, we sweep three MPTCP schedulers (minRTT [45], BLEST [21], and ECF [38]) and six congestion-control algorithms (CUBIC [26], BBR [16], Vegas [15], LIA [44], OLIA [33], and BALIA [56]), repeat each of the resulting 18 configurations 10 times, and report the mean and standard deviation of flow completion time (FCT).

Results. Table 2 reports the results. No scheduler or congestion-control algorithm dominates: combinations that perform comparably in one scenario diverge sharply in another, and their relative order can reverse outright once the path structure changes. We highlight three representative findings.

First, coupled congestion control pays off only with a shared bottleneck. Under the ECF scheduler, OLIA outperforms Vegas in Scenario A but falls behind it in Scenario B, where the shared bottleneck is removed. We infer the reason is that OLIA couples the congestion window growth of the two subflows and shifts the increase toward the path it deems in better condition, so when both subflows traverse a shared bottleneck, they compete less against each other. Vegas instead grows each subflow's window independently; its two subflows contend at the shared bottleneck, build up the queue, and induce additional losses, degrading its performance in Scenario A. Once with independent paths, the trade-off reverses: independent window growth lets Vegas saturate both paths, whereas OLIA's coupling caps the aggregate growth rate and leaves capacity unused.

Second, minRTT ranks among the best schedulers across many scenarios and congestion-control algorithms. This could be due to BLEST and ECF refining minRTT to mitigate head-of-line (HoL) blocking: when the faster subflow has no available congestion window, they may defer transmission rather than send immediately on the slower subflow. For long bulk transfers, however, HoL blocking is largely irrelevant, while every deferral directly idles available capacity. minRTT keeps transmitting on whichever subflow has congestion window, sustaining higher aggregate utilization.

Third, delay jitter reorders the schedulers. Under BALIA, the three schedulers perform within run-to-run variance of one another in Scenario C, but separate clearly in Scenario D, where BLEST falls behind ECF and minRTT. This may reflect their different responses to delay variations. In C, path 1 exhibits a stable delay, so all three schedulers keep both subflows active and behave similarly. In D, path 1's delay oscillates violently; BLEST treats it as a persistent HoL-blocking risk and largely stops scheduling traffic onto it, leaving the path underutilized. ECF instead incorporates the delay variance of both paths: when either path fluctuates, the fast–slow distinction becomes ambiguous, and ECF falls back to using the slower subflow much as minRTT does, so both outperform BLEST. ECF nevertheless trails minRTT slightly, possibly because its defer-then-release behavior emits bursts on the fast path, which disturb the underlying congestion controller and lower the throughput.

Takeaway. These ranking reversals show that shared bottlenecks and time-varying conditions can change conclusions drawn from independent, statically configured paths. Mininet can construct shared-bottleneck topologies and approximate constant link conditions, but it does not natively support fine-grained trace replay. Mahimahi supports trace-driven conditions, but its linear process nesting does not directly express branching and merging paths that contend for a shared queue. By composing these path structures and time-varying effects within a channel graph, Rattan enables offline evaluation to reveal how the relative performance of transport variants depends on path interactions that simpler configurations omit.

5.3 Real-World Deployment Experience

We have deployed Rattan at a large CDN provider for offline evaluation of transport and application-layer optimizations. The deployment has been in continuous use for 21 months and has executed 20,000,000 experiments, totaling 1.4 million channel-hours of emulation. Here, an experiment evaluates one combination of workload, system variant, and emulated path configuration; repeated runs of the same combination to obtain stable results count as one experiment. Rattan is integrated into the provider's pre-deployment evaluation pipeline covering both TCP and QUIC products: engineers use it to replay representative path conditions, evaluate congestion-control variants (new mechanisms or parameter settings), sweep application parameters (e.g., tuning adaptive bitrate algorithms), and reproduce regressions before exposing changes to production traffic. Campaigns typically use 10–50 worker machines, with the allocation adjusted according to campaign size and completion deadlines. The orchestrator distributes experiments across these workers as described in §3.3.

This deployment changed the provider's congestion-control rollout workflow. Before Rattan, evaluating a new congestion-control change required relying heavily on staged A/B tests and ad hoc diagnosis, taking an average of 24 weeks from candidate implementation to deployment decision. With Rattan, the same cycle takes only 18 days. In this workflow, engineers can run large offline campaigns to screen out clear regressions and enable their reproduction for diagnosis, reducing the number of candidates requiring online A/B evaluation for final validation before live rollout. Candidates that regress offline are occasionally included in online tests to check agreement. In one representative campaign, Rattan evaluated 15 congestion-control variants over 83,732 path configurations and identified reproducible performance regressions in 11 variants. We also evaluated all 15 variants in online A/B tests: the same 11 variants exhibited performance regressions, while the remaining four achieved comparable or better performance.

The deployment also validates Rattan's extensibility. Engineers at the provider added 12 custom cells for production-specific path effects, including traffic policing, packet reordering, and wireless packet aggregation. On average, an engineer with no prior experience with Rattan internals implemented a new cell in 2.2 days, suggesting that the cell interface is simple enough for fast extension. In a concrete development example, a developer who had used Rattan through its command-line interface but had not previously read its source code implemented a packet-reordering cell in less than one day (349 lines of code) and a token-bucket cell in 3.5 days (1,453 lines of code). For the token-bucket cell, approximately two days were spent studying the Linux kernel's token-bucket implementation and parameter definitions, and the remaining 1.5 days implementing the cell in Rattan. This breakdown distinguishes the effort of understanding the emulated mechanism from that of implementing it through the cell interface. Through our production deployment, no cell has required inspection of packet payloads: packet lengths, timestamps, and selected headers have been sufficient for their emulation decisions. Cells that require payload data can access the needed byte ranges on demand through RVNIC (§3.2). Overall, the deployment shows that Rattan can support long-running, large-scale evaluation workflows while remaining flexible enough for engineers to adapt it to new production demands quickly.

6 Discussion

Why not use formal analysis for performance evaluation? Formal analysis offers strong guarantees for specific properties, and recent work has used it to prove impossibility results and worst-case bounds for congestion control [2, 3, 6, 7]. Applying it to production transport innovations on the modern Internet, however, is blocked on both sides of the formulation. On the network side, real path conditions are a joint, time-varying mixture of effects compounded across heterogeneous elements, admitting no stable distribution or compact model. On the innovation side, what is deployed is not a compact algorithm but a full stack,

whose behavior hinges on implementation details, parameter choices, and interactions across layers and competing flows—far beyond the small set of variables and constraints a solver can discharge. And even where analysis succeeds, it yields worst-case bounds, whereas a deployment decision rests on the performance distribution across realistic conditions. Emulation therefore remains necessary: it runs unmodified implementations as production testing does, under the control and coverage that systematic evaluation requires, exposing behaviors that abstracted formulations omit.

Limitations of Rattan. Rattan has three main limitations. First, it emulates the transport- and application-visible properties of a path rather than the lower-layer mechanisms that produce them: routing dynamics, wireless medium access, handovers, and satellite motion are not modeled directly, though their transport-level consequences—RTT shifts, capacity variation, loss bursts, reordering, and shared bottlenecks—can be captured by cells or traces. Second, a channel's data plane is a static DAG of cells. Although cell state, parameters, and branch selection can change dynamically (§3.1), the abstraction does not support cyclic packet flow or arbitrary addition or removal of cells and edges during a run. Across 21 months of production use, we have not encountered a transport- or application-level condition that could not be represented within this abstraction. Third, Rattan improves the emulation substrate but does not by itself guarantee model fidelity. It executes the specified cell graph with controlled packet timing and overload detection, while running unmodified transport and application implementations. Whether the resulting experiment predicts real-world behavior depends on how well the cells, traces, and parameters represent the target paths. If a model omits a relevant real-world effect, executing it accurately and reproducibly cannot compensate for that omission.

7 Related Work

Network models. A line of work pursues emulation fidelity by deriving path models from real traffic rather than handset parameters, and the models have grown progressively richer: from recording and replaying the packet traces of a measured link [42], to calibrating path parameters against measurements of live Internet paths [59], to learning data-driven models of middlebox-laden paths [9, 10] and reactive models that assign each packet an individual delay [5]. These efforts sharpen *what* an emulator should reproduce, but all of them run on top of an emulation substrate and thus inherit its expressiveness and efficiency. Rattan addresses precisely this substrate, offering such models an expressive and efficient base to execute on.

Network emulators. Existing emulators realize network conditions on three kinds of substrates. Hardware emulators such as Keysight [32] and HoloWAN [29] implement fixed and simple condition models on specialized ASICs or FPGAs and reach line rate. Kernel-based software emulators, including DummyNet [47], NetEm [27], ModelNet [55], Emulab [28], and Mininet [35], compose rigid in-kernel queueing disciplines over virtual NICs into emulated topologies. User-space software emulators such as Mahimahi [42] manipulate packets freely in nested processes, supporting trace replay and richer per-packet effects. As analyzed in §2.4, each substrate binds expression and execution to a single unit and thereby trades expressiveness against efficiency and reproducible scale; Rattan instead decouples the two with the cell abstraction and satisfies all three requirements at once.

Experiment platforms. A long line of infrastructure supports network experiments at scale, at two layers. At the resource layer, shared testbeds such as PlanetLab [17], Emulab [58], DETER [13], GENI [14], CloudLab [20], Chameleon [31], and FABRIC [11] provision controlled nodes, topologies, and programmable wide-area links. On top of these resources, orchestration frameworks manage the experiment lifecycle: OMF-F [46], SPLAY [36], and Klonet [39] describe and deploy experiments across federated testbeds, while MACI [23], pos [24], and NPF [12] drive large campaigns through parameter sweeps, remote execution, and metric collection. These systems target experiments on real or testbed networks and thus inherit their cost, limited control, and weak observability; offline experiments on Rattan complement them with controllability, reproducibility, and high parallelism. Their orchestrators, moreover, treat experiments as generic tasks, whereas Rattan exploits the distinctive characteristics of emulation itself to better provision and dispatch campaigns.

8 Conclusion

Modern transport evaluation requires emulators that can express realistic Internet paths, execute them efficiently, and support reproducible experiments at scale. This paper presents Rattan, a modular Internet path emulator built around the cell abstraction. By composing cells into channel graphs, Rattan makes complex path structures such as asymmetric links, trace-driven dynamics, multipath, and shared bottlenecks natural to express and extend. By executing the same graph with metadata-oriented packet I/O, graph-aware scheduling, and managed resource isolation, Rattan keeps the cost of modularity low enough for high-throughput paths and dense concurrent experiments. Our evaluation and deployment experience show that Rattan can sustain demanding emulation workloads, reveal conclusions that other emulators may miss, and serve as a practical substrate for large-scale pre-deployment evaluation.

Acknowledgments

We sincerely thank our shepherd, anonymous reviewers, and labmates in the Routing Group at Tsinghua University for their valuable feedback. We also thank Keith Winstein for his helpful discussions and insightful suggestions on this work. This work is supported by the National Natural Science Foundation of China (Grant No. 62221003 and 62372261) and the Future Internet Technology Infrastructure (FITI).

References

[1] Soheil Abbasloo, Chen-Yu Yen, and H. Jonathan Chao. 2020. Classic Meets Modern: a Pragmatic Learning-Based Congestion Control for the Internet. In *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020*, Henning Schulzrinne and Vishal Misra (Eds.). ACM, 632–647. doi: 10.1145/3387514.3405892

[2] Anup Agarwal, Venkat Arun, Devdeep Ray, Ruben Martins, and Srinivasan Seshan. 2022. Automating network heuristic design and analysis. In *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*. 8–16.

[3] Anup Agarwal, Venkat Arun, and Srinivasan Seshan. 2026. Frcc: Towards provably fair and robust congestion control. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26), Renton, WA*.

[4] Jeffrey G Andrews, Stefano Buzzi, Wan Choi, Stephen V Hanly, Angel Lozano, Anthony CK Soong, and Jianzhong Charlie Zhang. 2014. What will 5G be? *IEEE Journal on selected areas in communications* 32, 6 (2014), 1065–1082.

[5] Divyam Anshumaan, Sriram Balasubramanian, Shubham Tiwari, Nagarajan Natarajan, Sundararajan Sellamanickam, and Venkat N. Padmanabhan. 2023. Simulating Network Paths with Recurrent Buffering Units. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*. AAAI Press, 6684–6692.

[6] Venkat Arun, Mohammad Alizadeh, and Hari Balakrishnan. 2022. Starvation in end-to-end congestion control. In *SIGCOMM '22: ACM SIGCOMM 2022 Conference, Amsterdam, The Netherlands, August 22 - 26, 2022*, Fernando Kuipers and Ariel Orda (Eds.). ACM, 177–192. doi: 10.1145/3544216.3544223

[7] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. 2021. Toward formally verifying congestion control behavior. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*, Fernando A. Kuipers and Matthew C. Caesar (Eds.). ACM, 1–16. doi: 10.1145/3452296.3472912

[8] Venkat Arun and Hari Balakrishnan. 2018. Copa: Practical Delay-Based Congestion Control for the Internet. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. USENIX Association, 329–342. https://www.usenix.org/conference/nsdi18/presentation/arun

[9] Sachin Ashok, Sai Surya Duvvuri, Nagarajan Natarajan, Venkata N. Padmanabhan, Sundararajan Sellamanickam, and Johannes Gehrke. 2020. iBox: Internet in a Box. In *HotNets '20: The 19th ACM Workshop on Hot Topics in Networks, Virtual Event, USA, November 4-6, 2020*. ACM, 23–29. https://doi.org/10.1145/3422604.3425935

[10] Sachin Ashok, Shubham Tiwari, Nagarajan Natarajan, Venkata N. Padmanabhan, and Sundararajan Sellamanickam. 2022. Data-Driven Network Path Simulation with iBox. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1 (2022), 6:1–6:26.

[11] Ilya Baldin, Anita Nikolich, James Griffioen, Inder Monga, Kuang-Ching Wang, Tom Lehman, Paul Ruth, and Ewa Deelman. 2019. FABRIC: A National-Scale Programmable Experimental Network Infrastructure. *IEEE Internet Comput.* 23, 6 (2019), 38–47. doi: 10.1109/MIC.2019.2958545

[12] Tom Barbette. 2024. Poster: NPF: orchestrate and reproduce network experiments. In *2024 ACM Conference on Reproducibility and Replicability*.

[13] Terry Benzel, Robert Braden, Dongho Kim, Anthony D. Joseph, B. Clifford Neuman, Ron Ostrenga, Stephen Schwab, and Keith Sklower. 2007. Design, Deployment, and Use of the DETER Testbed. In *DETER Community Workshop on Cyber Security Experimentation and Test 2007, Boston, Ma, USA, August 6-7, 2007*, Terry V. Benzel and George Kesidis (Eds.). USENIX Association. https://www.usenix.org/conference/deter-2007/design-deployment-and-use-deter-testbed

[14] Mark Berman, Jeffrey S. Chase, Lawrence H. Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. 2014. GENI: A federated testbed for innovative network experiments. *Comput. Networks* 61 (2014), 5–23. doi: 10.1016/J.BJP.2013.12.037

[15] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. 1994. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proceedings of the ACM SIGCOMM 1994 Conference on Communications Architectures, Protocols and Applications, London, UK, August 31 - September 2, 1994*, Jon Crowcroft (Ed.). ACM, 24–35. doi: 10.1145/190314.190317

[16] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-Based Congestion Control. *ACM Queue* 14, 5 (2016), 20–53. doi: 10.1145/3012426.3022184

[17] Brent N. Chun, David E. Culler, Timothy Roscoe, Andy C. Bavier, Larry L. Peterson, Mike Wawrzoniak, and Mic Bowman. 2003. PlanetLab: an overlay testbed for broad-coverage services. *Comput. Commun. Rev.* 33, 3 (2003), 3–12. doi: 10.1145/956993.956995

[18] Mo Dong, Tong Meng, Doron Zarchy, Engin Arslan, Yossi Gilad, Brighten Godfrey, and Michael Schapira. 2018. PCC Vivace: Online-Learning Congestion Control. In *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. USENIX Association, 343–356. https://www.usenix.org/conference/nsdi18/presentation/dong

[19] Dexuan Duan, Xinshuo Wang, Lin Li, and Lei Liu. 2024. FPGA-Based High-Performance Network Impairment Emulator. *Electronics* 13, 24 (2024). doi: 10.3390/electronics13244998

[20] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuang-Ching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 1–14. https://www.usenix.org/conference/atc19/presentation/duplyakin

[21] Simone Ferlin, Özgü Alay, Olivier Mehani, and Roksana Boreli. 2016. BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks. In *2016 IFIP Networking Conference, Networking 2016 and Workshops, Vienna, Austria, May 17-19, 2016*. IEEE Computer Society, 431–439. doi: 10.1109/IFIPNETWORKING.2016.7497206

[22] Alan Ford, Costin Raiciu, Mark J. Handley, Olivier Bonaventure, and Christoph Paasch. 2020. TCP Extensions for Multipath Operation with Multiple Addresses. RFC 8684. doi: 10.17487/RFC8684

[23] Alexander Frömmgen, Denny Stohr, Boris Koldehofe, and Amr Rizk. 2018. Don't repeat yourself: seamless execution and analysis of extensive network experiments. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*, Xenofontas A. Dimitropoulos, Alberto Dainotti, Laurent Vanbever, and Theophilus Benson (Eds.). ACM, 20–26. doi: 10.1145/3281411.3281420

[24] Sebastian Gallenmüller, Dominik Scholz, Henning Stubbe, and Georg Carle. 2021. The pos framework: a methodology and toolchain for reproducible network experiments. In *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*, Georg Carle and Jörg Ott (Eds.). ACM, 259–266. doi: 10.1145/3485983.3494841

[25] Google, Deloitte Digital, and Fifty-Five. 2020. Milliseconds Make Millions. https://www.deloitte.com/ie/en/services/consulting/research/milliseconds-make-millions.html

[26] Sangtae Ha, Injong Rhee, and Lisong Xu. 2008. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Oper. Syst. Rev.* 42, 5 (2008), 64–74. doi: 10.1145/1400097.1400105

[27] Stephen Hemminger et al. 2005. Network emulation with NetEm. In *Linux.conf.au*, Vol. 5. 2005.

[28] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. 2008. Large-scale Virtualization in the Emulab Network Testbed. In *Proceedings of the 2008 USENIX Annual Technical Conference, USENIX ATC 2008, Boston, MA, USA, June 22-27, 2008. Proceedings*. USENIX Association, 113–128.

[29] HoloWAN Team. 2026. HoloWAN Network Emulator: Network Emulation & Performance Testing. https://www.msytest.cn/en/HoloWAN_Emulator.html. [Accessed: 2026-06-10].

[30] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2013. B4: experience with a globally-deployed software defined wan. In *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013*. ACM, 3–14. doi: 10.1145/2486001.2486019

[31] Kate Keahey, Joe Mambretti, Paul Ruth, and Dan Stanzione. 2019. Chameleon: A Large-Scale, Deeply Reconfigurable Testbed for Computer Science Research. In *27th IEEE International Conference on Network Protocols, ICNP 2019, Chicago, IL, USA, October 8-10, 2019*. IEEE, 1–2. doi: 10.1109/ICNP.2019.8888067

[32] Keysight Technologies. 2026. Network Emulators. https://www.keysight.com/us/en/products/network-emulators.html. Accessed: 2026-06-10.

[33] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. 2013. MPTCP Is Not Pareto-Optimal: Performance Issues and a Possible Solution. *IEEE/ACM Trans. Netw.* 21, 5 (2013), 1651–1665. doi: 10.1109/TNET.2013.2274462

[34] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan R. Iyengar, Jeff Bailey, Jeremy Dorfman, Jim Roskind, Joanna Kulik, Patrik Westin, Raman Tenneti, Robbie Shade, Ryan Hamilton, Victor Vasiliev, Wan-Teh Chang, and Zhongyi Shi. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. ACM, 183–196. doi: 10.1145/3098822.3098842

[35] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM Workshop on Hot Topics in Networks. HotNets 2010, Monterey, CA, USA - October 20 - 21, 2010*. ACM, 19.

[36] Lorenzo Leonini, Etienne Rivière, and Pascal Felber. 2009. SPLAY: Distributed Systems Evaluation Made Simple (or How to Turn Ideas into Live Systems in a Breeze). In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*, Jennifer Rexford and Emin Gün Sirer (Eds.). USENIX Association, 185–198. http://www.usenix.org/events/nsdi09/tech/full_papers/leonini/leonini.pdf

[37] Fangfan Li, Arian Akhavan Niaki, David R. Choffnes, Phillipa Gill, and Alan Mislove. 2019. A large-scale analysis of deployed traffic differentiation practices. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019*. ACM, 130–144.

[38] Yeon-Sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. 2017. ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2017, Incheon, Republic of Korea, December 12 - 15, 2017*. ACM, 147–159. doi: 10.1145/3143361.3143376

[39] Tie Ma, Long Luo, Hongfang Yu, Xi Chen, Jingzhao Xie, Chongxi Ma, Yunhan Xie, Gang Sun, Tianxi Wei, Li Chen, Yanwei Xu, and Nicholas Zhang. 2024. Klonet: an Easy-to-Use and Scalable Platform for Computer Networks Education. In *21st USENIX Symposium on Networked Systems Design and Implementation, NSDI 2024, Santa Clara, CA, April 15-17, 2024*, Laurent Vanbever and Irene Zhang (Eds.). USENIX Association. https://www.usenix.org/conference/nsdi24/presentation/ma

[40] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*. ACM, 197–210. doi: 10.1145/3098822.3098843

[41] François Michel, Martino Trevisan, Danilo Giordano, and Olivier Bonaventure. 2022. A first look at starlink performance. In *Proceedings of the 22nd ACM Internet Measurement Conference*. 130–136.

[42] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: Accurate Record-and-Replay for HTTP. In *Proceedings of the 2015 USENIX Annual Technical Conference, USENIX ATC 2015, July 8-10, Santa Clara, CA, USA*. USENIX Association, 417–429.

[43] Ookla. 2024. Speedtest by Ookla Global Fixed and Mobile Network Performance Maps. AWS Open Data Registry. https://registry.opendata.aws/speedtest-global-performance/

[44] Costin Raiciu, Mark Handley, and Damon Wischik. 2011. Coupled Congestion Control for Multipath Transport Protocols. *RFC* 6356 (2011), 1–12. doi: 10.17487/RFC6356

[45] Costin Raiciu, Christoph Paasch, Sébastien Barré, Alan Ford, Michio Honda, Fabien Duchene, Olivier Bonaventure, and Mark Handley. 2012. How Hard Can It Be? Designing and Implementing a Deployable Multipath TCP. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, Steven D. Gribble and Dina Katabi (Eds.). USENIX Association, 399–412. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/raiciu

[46] Thierry Rakotoarivelo, Guillaume Jourjon, and Max Ott. 2014. Designing and orchestrating reproducible experiments on federated networking testbeds. *Comput. Networks* 63 (2014), 173–187. doi: 10.1016/J.BJP.2013.12.033

[47] Luigi Rizzo. 1997. Dummynet: a simple approach to the evaluation of network protocols. *Comput. Commun. Rev.* 27, 1 (1997), 31–41.

[48] Pramod Sanaga, Jonathon Duerig, Robert Ricci, and Jay Lepreau. 2009. Modeling and Emulation of Internet Paths. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. USENIX Association, 199–212.

[49] Yixin Shen, Ruihua Chen, Bo Wang, Jing Chen, Haochen Zhang, Minhu Wang, Mingwei Xu, and Zili Meng. 2026. Mortise: Auto-tuning Congestion Control to Optimize QoE via Network-Aware Parameter Optimization. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2026, Renton, WA, May 4-6, 2026*, Srikanth Kandula and Hakim Weatherspoon (Eds.). USENIX Association, 2303–2321. https://www.usenix.org/conference/nsdi26/presentation/shen-yixin

[50] Bruce Spang, Veronica Hannan, Shravya Kunamalla, Te-Yuan Huang, Nick McKeown, and Ramesh Johari. 2021. Unbiased experiments in congested networks. In *IMC '21: ACM Internet Measurement Conference, Virtual Event, USA, November 2-4, 2021*. ACM, 80–95. doi: 10.1145/3487552.3487851

[51] Thomas Stockhammer. 2011. Dynamic adaptive streaming over HTTP -: standards and design principles. In *Proceedings of the Second Annual ACM SIGMM Conference on Multimedia Systems, MMSys 2011, Santa Clara, CA, USA, February 23-25, 2011*. ACM, 133–144. doi: 10.1145/1943552.1943572

[52] Lizhuang Tan, Nguyen Van Tu, Xinhang Wang, Peiying Zhang, and James Won-Ki Hong. 2026. SDNIE: A Software-Defined Approach to High-Performance Network Impairment Emulation Using Programmable Switches. *IEEE Transactions on Network and Service Management* 23 (2026), 4970–4981. doi: 10.1109/TNSM.2026.3694388

[53] Farzad Tashtarian, Abdelhak Bentaleb, Hadi Amirpour, Sergey Gorinsky, Junchen Jiang, Hermann Hellwagner, and Christian Timmerer. 2024. {ARTEMIS}: adaptive bitrate ladder optimization for live video streaming. In *21st USENIX Symposium on Networked Systems Design*

and Implementation (NSDI 24). 591–611.

[54] The Linux Kernel Community. [n. d.]. AF_XDP. Linux kernel documentation. https://docs.kernel.org/networking/af_xdp.html

[55] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeffrey S. Chase, and David Becker. 2002. Scalability and Accuracy in a Large-Scale Network Emulator. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*. USENIX Association.

[56] Anwar Walid, Qiuyu Peng, Jaehyun Hwang, and S Low. 2016. Balanced linked adaptation congestion control algorithm for MPTCP. *Internet Engineering Task Force, Internet-Draft draft-walid-mptcp-congestion-control-04* (2016).

[57] Yifan Wang, Minzhao Lyu, and Vijay Sivaraman. 2024. Characterizing User Platforms for Video Streaming in Broadband Networks. In *Proceedings of the 2024 ACM on Internet Measurement Conference, IMC 2024, Madrid, Spain, November 4-6, 2024*. ACM, 563–579. doi: 10.1145/3646547.3688435

[58] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An Integrated Experimental Environment for Distributed Systems and Networks. In *5th Symposium on Operating System Design and Implementation (OSDI 2002), Boston, Massachusetts, USA, December 9-11, 2002*, David E. Culler and Peter Druschel (Eds.). USENIX Association. http://www.usenix.org/events/osdi02/tech/white.html

[59] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Alexander Levis, and Keith Winstein. 2018. Pantheon: the training ground for Internet congestion-control research. In *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*. USENIX Association, 731–743.

[60] Chen-Yu Yen, Soheil Abbasloo, and H. Jonathan Chao. 2023. Computers Can Learn from the Heuristic Designs and Master Internet Congestion Control. In *Proceedings of the ACM SIGCOMM 2023 Conference, ACM SIGCOMM 2023, New York, NY, USA, 10-14 September 2023*. ACM, 255–274. doi: 10.1145/3603269.3604838

[61] Xuan Zeng, Haoran Xu, Chen Chen, Xumiao Zhang, Xiaoxi Zhang, Xu Chen, Guihai Chen, Yubing Qiu, Yiping Zhang, Chong Hao, and Ennan Zhai. 2025. Learning Production-Optimized Congestion Control Selection for Alibaba Cloud CDN. In *22nd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2025, Philadelphia, PA, USA, April 28-30, 2025*, Theophilus A. Benson and Radhika Niranjan Mysore (Eds.). USENIX Association, 827–845. https://www.usenix.org/conference/nsdi25/presentation/zeng

[62] Xumiao Zhang, Shuowei Jin, Yi He, Ahmad Hassan, Z. Morley Mao, Feng Qian, and Zhi-Li Zhang. 2024. QUIC is not Quick Enough over Fast Internet. In *Proceedings of the ACM on Web Conference 2024, WWW 2024, Singapore, May 13-17, 2024*. ACM, 2713–2722. doi: 10.1145/3589334.3645323

[63] Zhilong Zheng, Yunfei Ma, Yanmei Liu, Furong Yang, Zhenyu Li, Yuanbo Zhang, Jiuhai Zhang, Wei Shi, Wentao Chen, Ding Li, Qing An, Hai Hong, Hongqiang Harry Liu, and Ming Zhang. 2021. XLINK: QoE-driven multi-path QUIC transport in large-scale video services. In *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*. ACM, 418–432. doi: 10.1145/3452296.3472893

[64] Shengtong Zhu, Yan Liu, Lingfeng Guo, and Jack Yiu-Bun Lee. 2026. R-TCP: A Framework to Optimize TCP Performance Over Rate-Limiting Networks. In *23rd USENIX Symposium on Networked Systems Design and Implementation, NSDI 2026, Renton, WA, May 4-6, 2026*, Srikanth Kandula and Hakim Weatherspoon (Eds.). USENIX Association, 609–630. https://www.usenix.org/conference/nsdi26/presentation/zhu

[65] Shengtong Zhu, Yan Liu, Lingfeng Guo, Rudolf K. H. Ngan, and Jack Y. B. Lee. 2023. mBBR - Improving BBR Performance Over Rate-Limited Mobile Networks. In *31st IEEE International Conference on Network Protocols, ICNP 2023, Reykjavik, Iceland, October 10-13, 2023*. IEEE, 1–11. doi: 10.1109/ICNP59255.2023.10355605