Days: Discrete-Event Network Simulation on Steroids

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto

***Abstract*—As large foundation models are routinely trained with hundreds of thousands of GPU compute nodes, the need for simulating a computer network at scale has become more critical and relevant than ever. Without a doubt, packet-level discrete-event simulation (DES) offers the finest granularity, and thus the highest accuracy. Unfortunately, conventional discrete-event simulators were widely known to be slow, and thus unable to accommodate the scale of modern networks. Recent work in the literature attempted to estimate the performance of large-scale networks using deep neural network models, but such estimation inevitably leads to a loss of packet-level accuracy, when compared to the ground truth from discrete-event simulators.**

But is it really the case that discrete-event simulators are not performant at scale? In this paper, we advocate that a *process-based* design is a simpler, more scalable, and performant choice than the current event-based design. We challenge the conventional wisdom that discrete-event simulators lack scalability, and progressively introduce the design and implementation of two new DES frameworks, *ns.py* and *Days*, built using Python and Rust, and with modern development advances in generators, asynchronous programming, and stackless coroutines. Our new simulation frameworks are designed to be lean and performant, outperforming existing discrete-event simulators and performance estimators by up to three orders of magnitude.

I. Introduction

As serverless web services span multiple geographically distributed regions and large foundation models are trained with tens of thousands of GPU compute nodes [1], [2], there is a pressing need for evaluating new network protocols and resource scheduling mechanisms at scale in modern communication networks [3], [4]. For four decades, discrete-event simulation (DES) frameworks [5], [6], [7], [8], [9], [10], [11] have offered the best possible accuracy and the finest granularity, as they kept track of every packet traveling through modern networks that we wish to study. However, it has been widely accepted common knowledge that existing DES frameworks were *slow*, and failed to offer acceptable performance at the scale of modern networks. In contrast, from the perspective of raw computing power, we have witnessed speed improvements over four decades by a few orders of magnitude, made possible by advances in both single-core performance and multi-core architectures.

In the literature, there have been two pathways to improve DES performance: parallel discrete-event simulation (PDES) and network performance estimation. It has been shown since four decades ago that one can use a conservative [12] or an optimistic protocol [13] to parallelize discrete-event simulation. Despite claims to the contrary in the recent literature [14], [15], these parallelization strategies from decades ago were quite effective in our experience. More recently, network performance estimation at the packet level, represented by RouteNet [16], MimicNet [15], and DeepQueueNet [17], became the *de facto* alternative to discrete-event simulation.

But why are existing DES frameworks so slow? As they were consistently implemented with high-performance programming languages — NEST [5] and REAL [6] used C and all modern DES frameworks used C++ — it can be intuitively concluded that their lack of performance at scale was due to the inherent design of discrete-event simulation in general. Conceptually, however, discrete-event simulation is rather simple: the simulated network only changes its state at discrete points in simulation time, and migrates from one state to another upon the occurrence of an *event*. Therefore, modern DES frameworks for network simulation, including ns-3 [9], OPNET [10], and OmNeT++ [11], were all built based on the design involving an *event queue* and a *scheduler*: the event queue holds all unprocessed future events in their timestamp order, and the scheduler simply processes events in the event queue in order. As each event is processed, more events may be produced and inserted into the event queue.

However, such an *event-based* design is not the only way to build a DES framework. The classic design is *process-based* simulation, dating back six decades to Conway's abstraction of *coroutines* (1963) [18] and the semantics of *process* in the SIMULA programming language (1967) [19]. Fundamentally, such a classic abstraction of *coroutines* represents threads of execution that communicate with one another by passing messages. We advocate that the classic process-based design is simpler and promotes inherent concurrency, since each network element is represented by a *coroutine* that makes live progress over time, and is concurrent with all other network elements. Packets transmitted between network elements correspond naturally to messages passed over channels established between coroutines.

From the perspective of complexity and scalability, why do we promote a process-based design? After all, event-based designs are object-oriented as well, and each network element is represented as an object instead of a coroutine. In a nutshell, a process-based design enjoys two clear advantages. *First*, it is *simpler* to implement a new network element with a coroutine, because it is easier to conceive — and less error-prone to model — an element as an isolated entity with a fixed set of inputs and outputs, communicating with the other elements through message passing only. Fundamentally, a process-based design utilizes the *actor model* [20], [21], where each *actor* maintains its private states and can only affect one another indirectly through message passing. *Second*, it is *more performant* because coroutines (or actors) are naturally concurrent with a potentially higher degree of parallelism, while all the objects in an event-based design must share the global event queue and its scheduler within the *same* thread of execution. To accommodate a larger scale, one only needs to launch more coroutines in a process-based design, rather than partitioning the network and using elaborate and complex PDES mechanisms [12], [13], [22].

In this paper, we progressively introduce *two* new open-source DES frameworks for network simulation[1], using process-based designs. As a starting point, ns.py serves as a simple, proof-of-concept prototype developed in Python, just to explore the potential benefits of process-based designs using Python's generators, and to utilize the flexibility offered by Python's dynamic types to construct complex switches with simple elements. Despite the fact that Python is not known to be fast, ns.py shows excellent performance and outperforms ns-3, OmNeT++, and even DONS [14], the state-of-the-art DES framework. To build the most performant DES framework possible, our second DES framework, called Days, is built with Rust, taking advantage of its compiler guarantees for concurrency without data races. Days is designed to use *stackless coroutines*, utilizing the foundation of either a lock-free *single-threaded* executor for maximum single-threaded performance, or a *multi-threaded* executor for maximizing concurrency in large-scale simulation runs.

Highlights of our original contributions in this paper are twofold. From the perspective of *design*, we present *two* new DES frameworks with progressively better performance, and show convincing evidence that a *process-based* design is a simple, scalable, and performant choice for building DES frameworks of the next generation. Its stellar performance is made possible by asynchronous programming and stackless coroutines, two of the landmark advances in modern concurrent software development. From the perspective of *performance*, we present best practices on how a performant DES framework should be built with Rust, as well as our first-hand experiences building Days, which outperforms all existing work — DES frameworks and performance estimation tools alike — by at least an order of magnitude. As one example, in a FatTree-32 topology with 4096 flows, ns.py and Days delivered speedups of $29\times$ and $1574\times$, respectively, over DONS [14].

II. Motivation and Related Work

Due to its value in a wide range of engineering fields, discrete-event simulation (DES) frameworks have been widely studied and used over the past six decades. To build a DES framework, there have always been two alternative design choices: a *process-based* or an *event-based* design. The process-based design dates back to the abstraction of *coroutines* as originally envisioned by Conway [18] in 1963, and the SIMULA programming language in 1967 [19]. SIMULA, and a variety of similar later attempts such as Ada [23], have shown that coroutines are sufficient to simulate discrete-time events.

Process-based design. In a process-based design, a network element to be simulated is modeled as a *coroutine*, which is simply a thread of execution that can be suspended at predefined points of execution, and resumed to a state that it left off before suspension. When coroutines wish to interact with one another, they do so by passing messages. From an implementation perspective, coroutines can be implemented in a variety of ways in modern operating systems. User-level processes naturally implement such an abstraction, and communicate with each other using inter-process communication. IBM's NEST [5] and Berkeley's REAL [6] used user-level threads that shared the same virtual address space, and communication was *implicit* via shared memory.

When using process-based simulation in DES frameworks, all coroutines must be coupled with a globally shared fictitious simulation clock, which is a double-precision floating-point variable that represents discrete points in simulation time. A dedicated *executor*, first introduced by SIMULA [19], is needed to advance the simulation clock and coordinate the execution of coroutines, analogous to a *runtime manager* designed specifically for simulations.

Event-based design. Intuitively, an *event* in a DES framework occurs at a discrete time on the simulation clock, and migrates the simulated system from one state to another. An event-based design models each network element as an object that handles events with event handler functions, and as an event is handled, more events may be produced. All unprocessed events are stored in an event queue, sorted by their timestamps. An event scheduler simply processes events consecutively by their timestamp order. The LBNL Network Simulator, *ns* version 1 (circa 1994), was the first DES framework that migrated network simulation from a process-based design used by its predecessors to an event-based design.

Design choices. From the point of view of complexity, it is typically simpler to implement a network element as a coroutine with a process-based design, rather than as an object with an event-based design. This is due to the fact that the runtime logic of a network element is a sequential algorithm, and it is cognitively easier to implement a sequential algorithm as it is designed step by step, rather than remodeling its logic to multiple handlers responding to events within a finite state machine.

Consider a Deficit Round Robin (DRR) scheduler [24], for example, in Fig. 1(a) and Fig. 1(b). To model it as a finite state machine with an event-based design, we need to separate its logic into several handlers, corresponding to events such as when an inbound packet arrives (`recv()`) and when an outbound packet finishes transmission (`timeout()`). To correctly implement these handlers, one needs to remember certain states, such as the upcoming queue to visit in a round-robin fashion, as well as the deficit counter for each queue. When existing events are handled, new events will be produced and inserted back into the event queue. In contrast, if we model it as a coroutine, the DRR scheduler can be simply

[1]The source code and full documentation of both ns.py and Days can be found at their official website: https://days.sh.

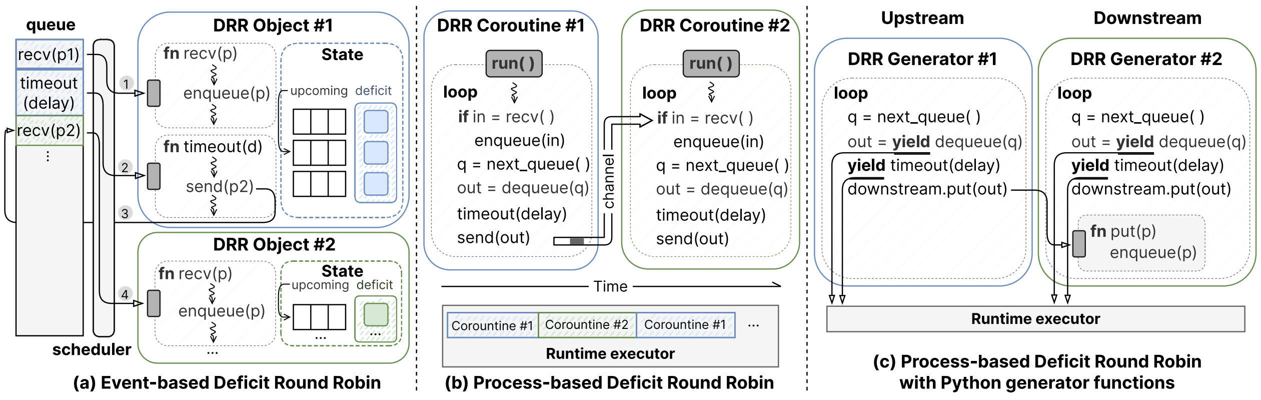


Fig. 1. Deficit Round Robin scheduler: Event-based vs. process-based designs. To implement a process-based design, one can use cooperatively scheduled coroutines (b) or generator functions supported by Python (c).

implemented as a loop. In each iteration, it receives inbound packets, moves on to the next queue, and selects and sends one or several outbound packets in a round-robin fashion. All packets are sent by passing messages between coroutines.

Data-oriented design. A recently proposed DES framework, called DONS [14], proposed a different way of increasing the degree of parallelism by using a *data-oriented* design using the Unity game engine, traditionally used for developing games [25]. To improve cache consistency, DONS proposed to store data of the same type (*e.g.*, timestamps in all the packets) together; and to improve data parallelism, it advocated processing a *batch* of data (*e.g.*, packets) concurrently using multiple threads. Unfortunately, in more complex cases such as with Weighted Fair Queueing (WFQ) schedulers [26], packets will need to be individually processed consecutively for correctness, as the choice of the next packet to be forwarded depends on previous decisions. As such, the degree of data parallelism may be limited in most simulation scenarios if discrete events are to be strictly processed in the order of their timestamps. If such strict order can be moderately relaxed, Days incorporates several optimizations — such as time quantization — that exploit data parallelism as well, albeit at a slight cost in accuracy.

Network performance estimation. In recent years, network performance estimation tools [15], [16], [17] were proposed to address the lack of scalability with DES frameworks by accepting a moderate loss of accuracy as a tradeoff, as compared to the ground truth produced by full-fidelity DES frameworks. The general strategy, shared by all performance estimation tools, is to use a deep neural network (DNN) model to represent a portion of the network topology or a single network device, and to train it using ground-truth datasets from DES frameworks. After the DNN model has been trained properly, it can be used for inference on GPUs using batches of packets as inputs, and important performance metrics — such as throughput, round-trip times, and flow completion times — can be deducted. The loss of accuracy, compared to ground truth from DES frameworks, is typically evaluated with the Wasserstein distance [27]. However, it is not clear how often the DNN models need to be retrained using new ground-truth data; it would be inconvenient, or even feasible, to train these models often, as such training may take hours to complete.

III. A Tale of Two Simulators

A. *Openings*

Complexity. In his article [28] titled "A Plea for Lean Software," Niklaus Wirth made the claim that "software's girth has surpassed its functionality," and that such complexity was due to software's monolithic design, in that all features are available all the time. We are of the opinion that current DES frameworks, such as ns-3 [9] and OmNeT++ [11], are too complex to use, to extend, and more importantly, to become performant at scale. If we use lines of code (LOC) — not including examples and tests — as a crude measure of complexity, while early frameworks such as REAL and ns-1 both had 15K LOC, the latest release of OmNeT++ had 257K LOC, ns-2 had 286K LOC, and ns-3 had 551K LOC! We advocate that one should get back to basics and, starting from a process-based design, build each network element with simplicity as a design principle. As Wirth proclaimed [28]: "Ideally, only a basic system with essential facilities would be offered, a system that would lend itself to various extensions."

As a starting point, few programming languages are simpler or more ubiquitous than Python. But does it offer suitable language-level support for a process-based design? Coroutines, which coexist in the same kernel thread and are cooperatively scheduled, have been supported since Python 3.4, and the `async`/`await` keywords have been supported since Python 3.5. Alternatively, one can use *generator iterators*, as coroutines and generator iterators are flip sides of the same coin: With a generator iterator created by a *generator* function containing `yield` calls, each `yield` temporarily suspends processing, remembering the current execution state. When the generator iterator resumes, it restarts execution where it left off [29], just as a cooperatively scheduled coroutine would do.

The first DES framework we have designed for network simulation, ns.py, uses the facility of *generator* functions in Python to realize the process-based design. It is simple to define generator functions, called `run()`, in network elements in ns.py, as one needs to *yield* to other coroutines in two cases only: waiting for an inbound packet, or for a timeout before sending out the next outbound packet, as we illustrate in Fig. 1(c). The following First-Come-First-Served (FCFS) scheduler is even simpler, including only an infinite loop:

```
1 def run(self):
2     while True:
3         p = yield self.store.get()
4         yield self.env.timeout(p.size * 8.0 / self.rate)
5         self.out.put(p)
```

At line 3, `yield` is used to suspend the execution of this coroutine until a new packet can be retrieved from the scheduler's own `store`, which is a FCFS queue. After execution resumes at line 4, the packet `p` is guaranteed to be retrieved, and we only need to wait for a period of time, which is the packet's transmission delay, to send it out. We use `yield` again for such waiting, suspending execution until the globally shared *simulation clock* is advanced beyond the specified timeout value. At that time, execution would resume again, and we send it out to the next-hop network element, defined as `self.out`, by calling its `put()` function.

All network elements implement `put()`, and the FCFS scheduler is no exception:

```
def put(self, packet):
    self.store.put(packet)
```

Of course, the actual FCFS scheduler implementation in ns.py needs to implement more features, such as a FCFS queue with a bounded buffer size. The logic, however, remains the same and strikingly simple: when an upstream element calls `put()`, the scheduler only needs to place it in its own queue (when certain conditions are satisfied). The scheduler executes this function, however, in its upstream element's coroutine, rather than its own `run()` coroutine. Calling the `put()` function of another element, therefore, is just a simple and ingenious mechanism to pass messages between coroutines by taking advantage of dynamic types — a Pythonic way of writing code. To connect two network elements, only one line of code is needed. For example, to connect an upstream DRR scheduler with a downstream packet sink:

```
    drr_scheduler.out = sink
```

Composability. As Python is a dynamically typed language, types are only determined at runtime. ns.py takes advantage of dynamic types to support a unique feature that more complex network elements can be composed by connecting multiple simpler elements, each of which can have no buffers at all. For example, one may *compose* a fair output-queued packet switch with a finite buffer size and a fair scheduler, such as a DRR scheduler, simply by defining a FCFS scheduler with a finite buffer size and a packet dropping strategy, and connecting it to a downstream DRR scheduler with no buffer at all — by specifying the option that `zero_buffer` ← `True`. The upstream element would also need to have the knowledge that its downstream element has no buffers, by specifying `zero_downstream_buffer` ← `True`. Despite the fact that it is deceptively simple to use this feature, its implementation is non-trivial, and infeasible without using Python's dynamic types.

Simplicity. With the use of `yield` calls in generator functions, our experience is that it becomes much simpler to extend ns.py with a new network element, by implementing its `run()`

Category	Network Elements
Scheduling	FCFS, DRR, WFQ, Static Priority, Virtual Clock
Packet dropping	Tail drop, Random Early Detection
Traffic generation	Distribution-based, trace-based, TCP Reno, TCP CUBIC, BBRv3
Demultiplexing	Flow Information Base, random, flow-based
Routing	Random simple path, shortest path
Traffic shaping	Token bucket, two-rate three-color token bucket

TABLE I
ESSENTIAL NETWORK ELEMENTS SUPPORTED IN ns.py.

coroutine with a process-based design. Unlike event-based designs, there are no event handlers to implement and no state to keep track of after resuming execution. It is not unusual to simply use an infinite loop as in our example, since the runtime executor, `env`, only runs for a finite amount of simulation time. Within the loop, one only needs to *yield* the CPU to other coroutines until a future event — even the choice of this keyword makes sense!

Through the lens of lines of code (LOC), a crude measure of complexity, ns.py should be considered simpler than all existing DES frameworks. It includes a curated collection of essential network elements, shown in Table I, with only **3235** LOC. To implement a complete simulation with a FatTree topology, only **54** LOC is necessary. With popular simulators hovering at 200-500K LOC, this level of simplicity can genuinely be called "lean software."

B. Adjournment

With ns.py's simplicity and ease of use, it is primed for fast prototyping of new designs at relatively small network scales. Yet, before v3.14, Python code runs single-threaded due to the Global Interpreter Lock (GIL), and performance is widely known to be lacking. As we concluded in Section II, the main advantage of the process-based design is its ease of achieving a higher degree of parallelism using multiple threads and additional CPU cores: one only needs to launch new coroutines. ns.py would not be able to enjoy such an advantage as it is forced to be single-threaded by Python. We are back to square one: would it be feasible to design a new DES framework with the best of both worlds: *performant* with the *simplicity* of a process-based design?

Let us revisit Conway's abstraction of *coroutines* [18]. In OmNeT++, process-based design is supported using coroutines *with dedicated stacks* [30]. With such stack-based coroutines, context switches between different coroutines are inefficient due to the presence of the local call stack. If coroutines could be implemented without using stacks, the lack of runtime efficiency would "vanish entirely," as proclaimed by Weber *et al.* [31]. At the high level, such *stackless coroutines* advocate that local data within a coroutine is best stored as fields in an active *instance* of the coroutine, rather than in a stack frame. Suspending execution in a stackless coroutine is, therefore, mapped to an ordinary `return` statement, and context switches become as fast as simple function calls.

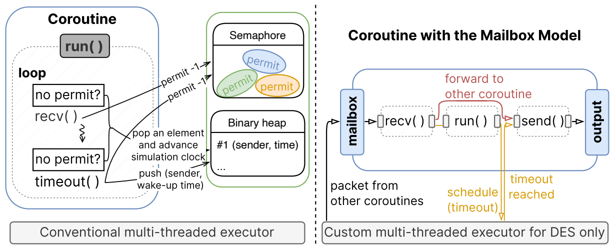
While planning started initially in 2016 [32], stackless coroutines were only officially supported in the stable Rust programming language with the advent of `async/await` recently [33]. The Rust compiler natively supports capturing local fields and the program counter in the `Future` structure. Various runtime executors are supported by third-party libraries, such as *tokio* [34], by "polling" *lazy* coroutines. Dubbed *zero-cost futures* [33], there is *zero* runtime overhead: a stackless coroutine in Rust is just a fancier way of creating a finite state machine [35] — a hallmark of event-based simulation!

Stackless coroutines aside, Rust enjoys many other advantages as a modern programming language, such as zero-cost abstraction, memory safety, as well as conservative ownership rules enforced by the compiler. With Rust, it is guaranteed that there will not be any data races across multiple threads, each corresponding to a CPU core and accommodating a large number of coroutines. Needless to say, as we set out to design and build Days, our new, simple and performant DES framework, Rust is our natural choice.

Unfortunately, though they have been planned for a future release, Rust does not yet support generator functions as Python does. This makes it unlikely to directly migrate our design and implementation from ns.py to Days. To start from the ground up, we selected *tokio* [34], a well-maintained multi-threaded executor in Rust, as our executor of choice. As *tokio* is primarily designed for networking I/O, it is not only multi-threaded — each hosting a large number of coroutines — by default, but also supports common thread synchronization primitives such as a *semaphore* [36], which maintains a set of *permits* that are used to synchronize access to a shared resource. In addition, it provides excellent support for highly performant *channels* [37]; messages can easily be passed between coroutines with *tokio*'s multi-producer, single-consumer (MPSC) channels, which are able to buffer an unbounded number of messages for future delivery.

In network simulation, there are only two cases in which the simulation clock may need to be advanced: ① when a coroutine is waiting to receive a message from its MPSC channel; and ② when it needs to time out for a period of simulation time by going to sleep. But when do we advance the simulation clock? The key insight is that the simulation clock should *only* be advanced when the simulation reaches a state of *quiescence* and *no coroutines are active*; *i.e.,* they are all blocked waiting for a message or a timeout. This is fundamentally a *deadlock* across coroutines, which is to be avoided at all costs in a conventional multi-threaded executor such as *tokio*.

With such a conundrum between the need to avoid or to seek a deadlock, we argue that it is still feasible to simulate discrete events using *tokio*, a conventional executor. Fig. 2(a) illustrates how we use a globally shared trio, involving a binary heap sorted by wake-up time, the current simulation time, and a single semaphore, to advance the simulation clock correctly. The binary heap keeps track of a collection of *(sender, wake-up time)* tuples, sorted by wake-up times. The number of available permits in the semaphore is used to keep track of the number of *live* coroutines; only when it is reduced to zero should we advance the simulation clock to the minimum wake-up time popped from the binary heap. We check if the simulation clock should be advanced whenever a coroutine blocks itself, either to receive a packet or to time out.


Coroutine
run( )
loop
no permit?
recv( )
no permit?
timeout( )
permit -1
permit -1
pop an element and advance simulation clock
push (sender, wake-up time)
Semaphore
permit
permit
permit
Binary heap
#1 (sender, time)
...
Conventional multi-threaded executor
Coroutine with the Mailbox Model
forward to other coroutine
mailbox
recv( )
run( )
send( )
output
packet from other coroutines
schedule (timeout)
timeout reached
Custom multi-threaded executor for DES only

(a) Failed first cut design using a globally shared binary heap and semaphore. **(b) *Days* with the mailbox model and its custom multi-threaded executor.**

Fig. 2. (a) A failed first cut design. (b) Days with the mailbox model and its custom multi-threaded executor.

In our first-cut design, the simulation clock is only advanced when the number of available permits is 0, which implies that no coroutines are currently alive and running, and a state of deadlock has been reached. When the simulation clock is advanced, it is advanced to the next wake-up time of the nearest coroutine in time, decided by popping an event from the binary heap.

The binary heap includes a collection of *(sender, wake-up time)* tuples, sorted by their wake-up times. Each entry in the binary heap is pushed right before a coroutine attempts to time out, and to sleep until a specific wake-up time is reached. When this occurs, a coroutine would create a one-shot channel using *tokio* (our runtime executor), and store the sender of the channel as the first element of the tuple to be pushed onto the binary heap.

There are two occasions when a coroutine goes to sleep: ① when it needs to receive a packet, arriving from other coroutines via an MPSC channel; and ② when it times out waiting for a specific wake-up time in the future. In both occasions, the coroutine would need to acquire a permit from the globally shared semaphore before going to sleep, and release the permit back to the semaphore after waking up. When a coroutine is initialized, it adds a new permit to the semaphore; and when it is terminated, it permanently removes one permit. This way, the number of available permits in the semaphore will accurately reflect coroutines that are currently alive.

Performance. Much to our dismay, despite the fact that it has access to multiple CPU cores, our first-cut design underperformed ns.py by $6\times$ in a FatTree-4 topology with 8 flows, achieving a mean running time of 1.898 seconds and a Standard Error of Mean (SEM) of 0.004 over 5 runs, as compared to 0.32 seconds (SEM 0.005) with ns.py, as well as an abysmal CPU utilization of only 10.3%. This is due to the fact that it spent the vast majority of its time *waiting* for the globally shared context, to access the binary heap, the semaphore, and the simulation clock. Since coroutines are typically very simple in their logic in network simulations — a FCFS scheduler only needs to dequeue a packet and send it out — as the number of coroutines scales up, the time competing for shared states would play an overwhelming role. Though the first cut failed, our efforts were far from futile: we understood that to maximize parallelism, the fundamental challenge is to share as little state as possible, as we advance the simulation clock only when a deadlock among coroutines emerges and a state of quiescence is reached.

C. Fork

Mailbox model and custom executor. Now that we are back to square one, the only remaining alternative is to build our own custom multi-threaded executor, with the sole purpose of supporting discrete-event network simulations. This is also an excellent opportunity to revisit our concurrent programming abstraction to realize process-based simulation: how, after all, should coroutines run and interact with one another in a simple and performant way?

We decided that the process-based design should be best implemented with what we refer to as the *mailbox* model, where each coroutine still maintains its own states during its execution, but only interacts with the external world via our new custom multi-threaded executor using a *mailbox*. With the mailbox model, each network element is an isolated coroutine with *one* mailbox and an arbitrary number of outputs connecting to other network elements. Our custom executor supports an Application Programming Interface (API) that allows a coroutine to be initialized with a new mailbox, to connect its output to public interfaces — such as `recv()` for inbound packets — in other coroutines via their mailboxes, and to schedule a timeout with a callback when it is reached. All possible performance optimizations are to be implemented within the executor, and the entire network topology is built upon its launch.

As we illustrate in Fig. 2(b), when a packet arrives at a network element, its `recv()` processes it by forwarding to another element or enqueueing it locally, and then calls `run()` if needed. In `run()`, we call our custom executor's `schedule()` to send out the packet to the coroutine's output at a later time, and the output is connected to a downstream network element during initialization. To implement `run()`, we may use a loop or, alternatively, schedule to call itself *recursively* after a timeout.

Under the hood, rather than exposing the binary heap for blocked coroutines and the simulation clock for global sharing, the custom multi-threaded executor is now responsible for maintaining these states locally in its own dedicated thread. The packets exchanged between network elements are implemented as `async` closures in Rust, which are forwarded by the custom executor via an asynchronous and bounded MPSC channel. Whenever all coroutines complete their local computations and a deadlock is reached, the custom executor is responsible for advancing the simulation clock, by managing its own priority queue containing all future events. It incorporates many performance optimizations, including a work-stealing strategy similar to that of *tokio*, which balances the workload across CPU cores. Days also supports a single-threaded executor to eliminate most runtime overhead related to mutex locks when multiple threads contend for shared states, which is useful when running less complex simulations.

Simplicity. As shown in Fig. 2(b), Days' mailbox model is *simple*, as all performance-optimized machinery has been moved into the custom multi-threaded executor itself. As no states are shared between network elements, there is no fear for data races, and no need for either mutual exclusion locks or atomic reference counting. From the perspective of users, Days is fully configurable using `toml` configuration files. Despite its simplicity, Days' mailbox model is *versatile* as well, as we are able to incorporate essential network elements shown in Table II, including complex scheduling disciplines such as Weighted Fair Queueing (WFQ) [26], as well as several congestion control variants (such as TCP CUBIC [38], BBRv3 [39], and DCQCN [40]). While the custom executor consists of 13244 LOC (a heavily customized fork from nexosim [41]), Days itself is built with only **8375** LOC — or 157K tokens, fitting comfortably into the context window of most frontier models.

Category	**Network Elements**
Topologies	FatTree, 2D and 3D Torus, custom
Scheduling	FCFS, DRR, WRR, Static Priority, WFQ, Virtual Clock
Packet processing	ECN, Tail Drop, Random Early Detection
Traffic generation	Distribution, TCP Reno, TCP CUBIC, DCQCN, BBRv3
Demultiplexing	Flow Information Base
Routing	Random simple path, shortest path, ECMP
Link Layer	Priority-based Flow Control

TABLE II
ESSENTIAL NETWORK ELEMENTS SUPPORTED IN Days.

Lean software. A further design choice that contributes to Days' efficiency is that selected protocols have been implemented behind compile-time feature gates in Rust. Protocols such as explicit congestion notification (ECN) [42] and priority-based flow control (PFC) [43] are fairly complex and implemented across several network elements; experiments that do not require them should not need to pay a price at runtime. By excluding them entirely from the compiled binary by default, this design preserves conceptual economy: the default build exposes only a basic set of essential elements, while specialized mechanisms are introduced explicitly when needed. Days enforces a strict "pay-for-what-you-use" cost model: when a protocol is disabled, its code paths are not merely bypassed at runtime but are absent at compile time, so it incurs no additional branches or state maintenance.

D. End Game

In large-scale simulation runs, Days may progress through hundreds of thousands of distinct timestamps where the global simulation clock advances. Each such timestamp — a "step" — induces a recurring sequence of runtime operations: extracting the next set of events at the minimum deadline, materializing the corresponding task groups, executing them to quiescence as a deadlock is reached, and repeating. When step counts are high, even modest per-step overhead becomes a dominant term in wall-clock time. To further improve Days' performance, we therefore concentrated on reducing ① the number of *actions* that interact with globally shared states, ② the number of times such interactions require synchronization, and ③ the number of steps for which the full quiescence barrier must be paid.

Minimizing contention for globally shared states. There are only two actions in a network element that may contend for globally shared states: ① reading the global simulation clock, and ② scheduling events for future execution in a shared global event queue. To minimize the number of reads from the global simulation clock, each read should be saved and shared between network elements as much as possible. Since packets are always forwarded between network elements, they are excellent vehicles for carrying the current simulation time: once a read is saved by an upstream element, its downstream counterpart can extract the current simulation time directly from a received packet, rather than from the global simulation clock.

To minimize contention for the global event queue, we attempt to fuse the scheduling of multiple events into a single request as much as possible. In a packet scheduler, for example, each packet is forwarded by scheduling two immediately consecutive events after a timeout: one to transmit a packet, another to run a new iteration of the scheduling discipline. In Days, we not only fuse these consecutive actions into a single scheduling request, but also attempt to fuse the scheduling of a *batch* of packets into a single request. Operationally, some network elements, such as FIFO schedulers, can compute a bounded lookahead window of departures and submit that window as a batch. In the same spirit as data-oriented design [14], this strategy preserves packet-level behavior while substantially reducing the frequency of synchronization for inserting into the global event queue.

Local event buffers with deterministic flushing. Bulk insertion reduces the number of global queue lock acquisitions; but concurrent worker threads in the executor may still produce scheduled events at high rates. Days' design incorporates per-worker *local event buffers*: Rather than inserting every newly produced event immediately into the global event queue, workers append events to local buffers, which are flushed at step boundaries when the executor reaches a deadlock and is quiescent. With this design, most scheduling activity remains local during a step, minimizing global lock traffic while the deterministic ordering for same-timestamp events is still preserved. Equally important, the flushing rule is semantic: local event buffers are drained only when no model tasks are executing, aligning with the correctness requirements of conservative parallel discrete-event simulation, where global progress is coordinated at well-defined boundaries [12], [44].

Time quantization. When the number of distinct timestamps is excessively large, the quiescence barrier is paid too frequently. To reduce the total number of steps taken, timestamps can be *quantized* in Days when the executor schedules actions to take place next. With time quantization, event deadlines are snapped to a grid. This coalesces near-identical deadlines that arise from floating-point drift, accumulated rounding error, or fine-grained pacing timers, thereby increasing the number of actions executed per step and improving data parallelism in a similar vein as data-oriented design [14]. When opted in for a potential performance boost, the granularity of time quantization — which may affect simulation accuracy — can also be configured and threaded through topology construction.

Keeping threads warm across step boundaries. Reducing the number of steps mitigates the number of barriers; the barrier itself must also be efficient. A substantial component of barrier cost in Days' multi-threaded executor is the repeated transition of workers into and out of OS-level sleep states. In workloads with many short steps, frequent sleep/wakeup transitions can dominate execution time. Days therefore implements a warm-pool policy with several complementary mechanisms: ① At the start of each run, the executor proactively wakes up several "hot" worker threads, ready to run tasks. ② Before sleeping, the main thread briefly spins to avoid a full sleep transition when new work is likely to arrive imminently. ③ On worker threads, a linger-then-sleep strategy keeps a designated subset of workers in a standby state for a short interval. If the next step begins promptly, workers can resume without a wakeup. Overall, Days' warm-pool policy reduces the overhead of reactivation across adjacent steps.

IV. Performance Evaluation

While Days is a natural choice when performance is needed, ns.py may very well stand on its own when we wish to rapidly prototype the design of new protocols at smaller network scales. We now evaluate the performance and scalability of both frameworks, in comparison with the state-of-the-art in both discrete-event simulation and network performance estimation tools.

Baseline benchmarks. As initial baseline benchmarks, we choose FatTree topologies configured with FCFS schedulers, which were widely used in the literature [14], [15], [17]. Each flow in our evaluation sends 1500 packets of 1000 bytes in size. In this context, we use the following four benchmarks for our comparisons: ① FatTree-4 (*i.e.,* $k = 4$) and 8 flows; ② FatTree-8 ($k = 8$) and 64 flows; ③ FatTree-16 ($k = 16$) and 512 flows; and ④ FatTree-32 ($k = 32$) and 4096 flows[2]. We carried out our experiments on a consumer-grade server with Intel i7-13700K (16 CPU cores), 128 GB of physical memory, and an NVIDIA GeForce RTX 4090. We used ns-3 3.40 and OmNeT++ 4.5; ns.py's runs used Python 3.13. Each experiment involved 5 runs, with the mean and SEM shown in Fig. 3. Due to substantial performance differences across-the-board, the running time on the y-axis uses a logarithmic scale.

The scale of the first two benchmarks is relatively small, which serves as a fitting starting point. Despite their complexity and numerous claims to the contrary (*e.g.*, [14], [15], [17]), ns-3 and OmNeT++ performed surprisingly well, after we spent a considerable amount of time fine-tuning their implementations for fair comparisons. Rather than over two hours for a FatTree-4 topology as claimed in the literature [17], OmNeT++ took only 1.91 seconds in our first benchmark. Rather than a 3× speedup claimed in [14], DONS performed

[2]We chose these benchmark configurations because they were the ones that DONS can work with. It turned out that, though its source code has been publicly available, DONS' authors confirmed that the executable compiled from its source code failed to run. For this reason, we had to run the provided pre-compiled executable, which *only* supported these baseline benchmarks. All benchmarking configurations in ns.py and Days have been made available as open source for more reproducible research.

slightly worse than both ns-3 and OmNeT++. Surprisingly, despite the performance penalty imposed by Python, ns.py outperformed both ns-3 and OmNeT++ in the FatTree-4 topology, and outperformed DONS with a 12× speedup.

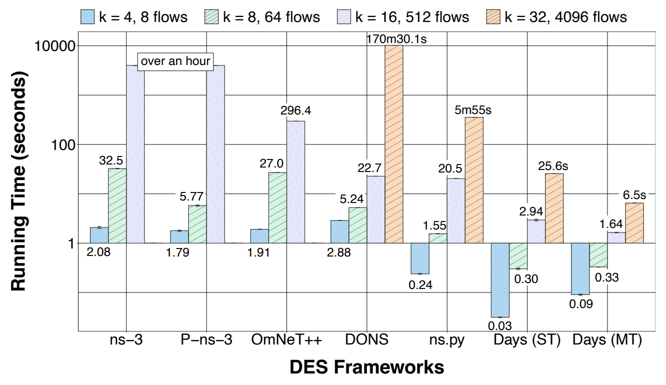


Fig. 3. ns.py outperformed all existing DES frameworks in baseline benchmarks, including DONS. Packing even more performance, Days outperformed ns.py by up to 55×, and outperformed DONS by 32×, 17×, 14×, and 1574× in these baseline benchmarks, respectively.

When we moved to the FatTree-8 topology, ns-3 with a single process — as well as OmNeT++ — started to slow down quite substantially, but its parallelized counterpart, with 9 logical processes running on 9 different CPU cores and communicating with one another using the Message Passing Interface (MPI), performed admirably well, almost offering a linear speedup. Our results again diverged from experimental results reported in the literature [14]. DONS, while having the opportunity to use all our 16 CPU cores, outperformed parallelized ns-3 slightly, but its performance was a far cry from a speedup of 3× over parallelized ns-3 as it originally claimed [14]. ns.py, while running with only one thread on a single CPU core, was 15× faster than OmNeT++ and 3.4× faster than DONS. In the FatTree-16 benchmark, ns.py was marginally faster than DONS; and interestingly, neither vanilla nor parallel ns-3 were scalable at all: they both took over an hour before we terminated the jobs.

We now turn to Days, and show its performance with both single-threaded (ST) and multi-threaded (MT) executors. In the case of FatTree-4, while the previously fastest framework, ns.py, took 0.32 seconds, Days (ST) took only 0.03 seconds, representing a speedup of 96× over DONS. In the case of FatTree-8 and FatTree-16, Days (ST) outperformed ns.py by 5× and 7×, respectively; it also achieved a speedup of 17× and 8× over DONS, respectively, and a speedup of 100× over OmNeT++ with FatTree-16. But with FatTree-16, the true winner is Days (MT), achieving a 1.8× speedup over Days (ST), and a 14× speedup over DONS.

Finally, with the FatTree-32 topology and 4096 flows, we only evaluated DONS and ns.py against Days, as it was not feasible to wait for the other contestants. This was a benchmark where Days (MT) excelled: a 55× speedup over ns.py, and a 1574× speedup over DONS — we double-checked our configurations and they were indeed this fast! Our performance optimizations on Days' multi-threaded executor have also catapulted it to a significant 4× speedup over the single-threaded executor. Suffice it to say, Days was fast.

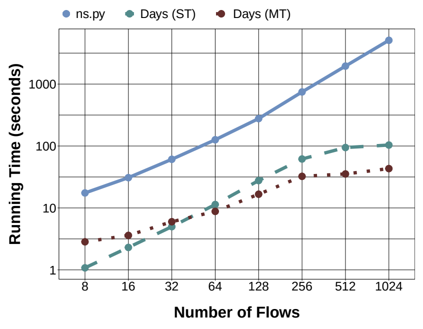
Benchmark	MimicNet	ns.py	Days ST	Days MT
FatTree-4, 8 flows	9.46s (± 0.11s)	0.24s (± 0.005s)	0.03s (± 0.0006s)	0.09s (± 0.0024s)
FatTree-8, 64 flows	23.32s (± 0.27s)	1.55s (± 0.02s)	0.30s (± 0.011s)	0.33s (± 0.002s)
FatTree-16, 512 flows	567.5s (± 2.50s)	20.3s (± 0.10s)	2.94s (± 0.119s)	1.64s (± 0.026s)

TABLE III
NOT INCLUDING THE TRAINING TIME, MIMICNET IS STILL NOT COMPETITIVE AGAINST ns.py AND Days.

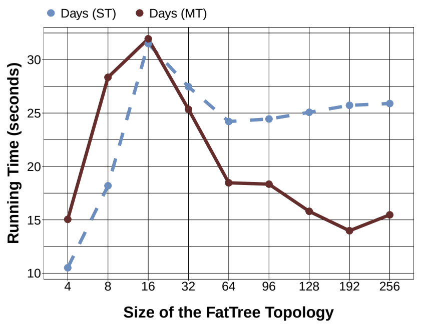
Network performance estimation. We then turned our attention to recent tools for network performance estimation — MimicNet [15] and DeepQueueNet [17] — by running their vanilla source code without modifications. It turns out that the source code for DeepQueueNet only supports the FatTree-4 topology; and with 5 test runs, the mean running time we measured was 174.63 seconds, with an SEM of 2.66 seconds. This was substantially slower than all DES frameworks. Granted, the single NVIDIA GPU on our server did not satisfy its 4-GPU hardware requirement, and inference was performed on the CPUs. Still, even with the benefit of 4 GPUs, the performance of our frameworks, at 0.24, 0.03, and 0.09 seconds, may still be quite competitive.

The source code of MimicNet [15] can indeed support all three benchmarks, and we again carried out 5 test runs each, using CPU cores to perform both training and inference, as the CUDA version supported by MimicNet no longer supports current-generation NVIDIA GPUs. As we show in Table III, it was not competitive in performance, which is the primary motivation for designing network performance estimators. Even ns.py outperformed MimicNet by 20×, not to mention Days. In addition, our MimicNet results did not include the time needed for training, which took an average of 211 minutes and 24.3 seconds for the FatTree-16 topology.

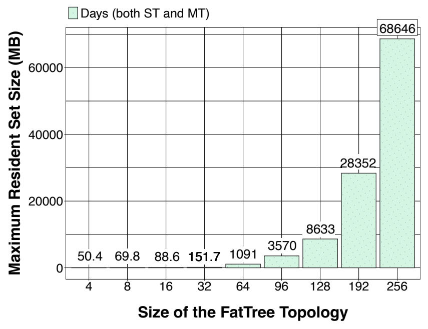
Scaling up the number of flows. Now that ns.py and Days have both outperformed existing DES frameworks and performance estimators, we continue with a more in-depth analysis on their scalability with respect to the number of flows in a FatTree-32 topology. Flows in each simulation ran for 10 seconds, produced traffic at 10 Mbps, and used TCP CUBIC as their congestion control protocol. The link capacities were set as 100 Mbps. The running time on the y-axis, again, uses a logarithmic scale. As Fig. 4(a) shows, ns.py's running times increased exponentially with the exponential increase in the number of flows simulated; the line graph is almost linear when plotted using the logarithmic scale. This shows that the runtime overhead was relatively fixed, in that no extra workload was performed per flow as more flows were simulated. In contrast, Days ST and Days MT were much more performant — both overall and on a per-flow basis — as the number of flows scales up. In particular, exactly as we expected due to its multi-threaded executor, Days MT craved more workload, and was able to catch up with its single-threaded sibling as the number of flows reached beyond 64. At these larger scales, the speedup offered by parallelism caught up with the synchronization overhead. It is also worth noting



(a) Varying the numbers of flows in FatTree-32.



(b) Scaling up the topologies with 128 flows.



(c) Memory footprint as topologies scale up.

Fig. 4. Performance of ns.py, Days (ST) and Days (MT) at scale.

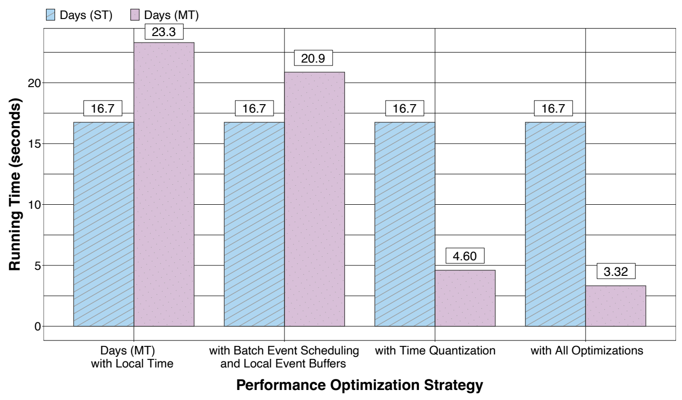


Fig. 5. Optimizing the performance of Days' multi-threading: how various strategies helped.

that with fewer than 64 flows, both Days (ST) and Days (MT) took less than 10 seconds to finish a simulation with a 10-second duration in simulated time!

Scaling up the size of the topology. In a similar vein, we wish to evaluate scalability with respect to the size of the network topology, moving from a FatTree-4 topology up, through FatTree-64 (the largest ever tested in the literature) all the way to FatTree-256 (with 4,194,304 hosts and 81,920 switches). Our results — with ns.py excluded as it was unable to accommodate some of these scales — are shown in Fig. 4(b), with 128 flows simulated, link capacities of 100 Mbps, and TCP CUBIC for congestion control. It was evident that both Days (ST) and Days (MT) scaled well all the way up to FatTree-256; Days (ST) was faster at smaller scales, but was outperformed by Days (MT) beyond FatTree-16. Since shortest paths between hosts were of similar lengths as the topology scales up, the running times remained steady.

Peak memory footprint. As we show in Fig. 4(c), Days (both ST and MT) offered excellent efficiency regarding its peak memory footprint, as measured by the maximum resident set size (RSS) (obtained with `/usr/bin/time -v`), as the network scales up from FatTree-4 to FatTree-256 using the same experimental settings as Fig. 4(b). For example, Days consumed only 2.96 GB with FatTree-96 and 8.43 GB with FatTree-128. This level of memory footprint can be easily accommodated by modern consumer-grade computers.

Performance optimizations. Finally, we show how our performance optimization strategies helped Days (MT) outperform its single-threaded counterpart. Fig. 5 shows the running times of a simulation using a FatTree-96 topology with 512 flows. With locally maintained simulation times using packets — which reduced contention for the global simulation clock — Days (MT) was about 40% slower than Days (ST). By adding batch event scheduling and local event buffers, contention for the global event queue was reduced, improving the performance by 11%; but it was still 25% slower than Days (ST). It was the addition of time quantization that substantially helped multi-threading performance, as more parallelism can be achieved by snapping timestamps to a grid; yet such a performance gain comes with a (controllable) loss of simulation accuracy. Ultimately, when everything is said and done — including warm threads across step boundaries — Days (MT) is 5× faster than Days (ST), a feat not to be dismissed lightly.

V. Concluding Remarks

This paper reflects five years' worth of research towards building simpler and more performant discrete-event network simulation frameworks, as we roll back to the basics and revisit every design decision. When we penned and committed the first line of code five years ago, we never imagined such handsome payoffs — brought to us by a more intuitive, decades-old process-based design, as well as modern advances in concurrent programming, with generator functions, runtime executors, stackless coroutines, and fearless concurrency offered by Rust, all ushered in by the recent desire of building highly performant cloud services.

While we are pleasantly impressed by the raw performance of both ns.py and Days as they outperformed all existing DES frameworks and network performance estimation tools by at least an order of magnitude, the highlight of this work lies in its inherent simplicity, by paring decades of work down to the bare essentials, and then building them back up. We are true believers that lean software is easier to use and to extend, and with 3235 LOC in ns.py and 8375 LOC in Days, we argue that the foundation for discrete-event network simulation is no exception. Though these frameworks are still in their early days and new network elements will be added brick by brick, with a refreshingly simple and highly performant foundation, the bright days of discrete-event network simulation are yet to come.

References

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

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

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

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

[5] A. Dupuy, J. Schwartz, Y. Yemini, and D. Bacon, "NEST: A Network Simulation and Prototyping Testbed," *Commun. ACM*, vol. 33, no. 10, p. 63–74, Oct. 1990.

[6] S. Keshav, "REAL: A Network Simulator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-88-472, Dec. 1988. [Online]. Available: https://www2.eecs.berkeley.edu/Pubs/TechRpts/1988/CSD-88-472.pdf

[7] S. McCanne, S. Floyd, and K. Fall, "ns version 1: LBNL Network Simulator," https://ee.lbl.gov/ns/, 1995.

[8] T. Issariyakul and E. Hossain, "Introduction to Network Simulator 2 (NS2)," in *Introduction to Network Simulator 2 (NS2)*. Springer, 2009, pp. 1–18.

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

[10] Z. Lu and H. Yang, *Unlocking the Power of OPNET Modeler*. Cambridge University Press, 2012.

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

[12] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations," *Commun. ACM*, vol. 24, no. 4, p. 198–206, Apr. 1981.

[13] D. R. Jefferson, "Virtual Time," *ACM Trans. Program. Lang. Syst.*, vol. 7, no. 3, p. 404–425, Jul. 1985.

[14] K. Gao, L. Chen, D. Li, V. Liu, X. Wang, R. Zhang, and L. Lu, "DONS: Fast and Affordable Discrete Event Network Simulation with Automatic Parallelization," in *Proc. ACM SIGCOMM 2023 Conference*. ACM, 2023, p. 167–181.

[15] Q. Zhang, K. K. W. Ng, C. Kazer, S. Yan, J. Sedoc, and V. Liu, "MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning," in *Proc. ACM SIGCOMM 2021 Conference*. ACM, 2021, p. 287–304.

[16] K. Rusek, J. Suárez-Varela, P. Almasan, P. Barlet-Ros, and A. Cabellos-Aparicio, "RouteNet: Leveraging Graph Neural Networks for Network Modeling and Optimization in SDN," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 10, pp. 2260–2270, 2020.

[17] Q. Yang, X. Peng, L. Chen, L. Liu, J. Zhang, H. Xu, B. Li, and G. Zhang, "DeepQueueNet: Towards Scalable and Generalized Network Performance Estimation with Packet-level Visibility," in *Proc. ACM SIGCOMM 2022 Conference*. ACM, 2022, p. 441–457.

[18] M. E. Conway, "Design of a Separable Transition-diagram Compiler," *Commun. ACM*, vol. 6, no. 7, p. 396–408, Jul. 1963.

[19] O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA 67. Common Base Language*, ser. Report at the Norwegian Computing Center (743). Report at the Norwegian Computing Center, 1984.

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

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

[22] K. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 5, pp. 440–452, 1979.

[23] G. Lomow and B. Unger, "The Process View of Simulation in Ada," in *Proc. 14th Winter Simulation Conference (WSC)*, 1982, p. 77–86.

[24] M. Shreedhar and G. Varghese, "Efficient Fair Queuing Using Deficit Round-Robin," *IEEE/ACM Transactions on Networking*, vol. 4, no. 3, pp. 375–385, 1996.

[25] J. D. Bayliss, "The Data-Oriented Design Process for Game Development," *Computer*, vol. 55, no. 05, pp. 31–38, May 2022.

[26] A. Demers, S. Keshav, and S. Shenker, "Analysis and Simulation of a Fair Queueing Algorithm," in *Proc. ACM SIGCOMM*. ACM, 1989, pp. 1–12.

[27] A. Frohmader and H. Volkmer, "1-Wasserstein Distance on the Standard Simplex," *Algebraic Statistics*, vol. 12, no. 1, pp. 43–56, 2021.

[28] N. Wirth, "A Plea for Lean Software," *Computer*, vol. 28, no. 2, pp. 64–68, Feb. 1995.

[29] A. M. Kuchling, "Functional Programming HOWTO," https://docs.python.org/3/howto/functional.html, Jan. 2024.

[30] "OMNeT++ Simulation Manual — Activity," https://doc.omnetpp.org/omnetpp/manual/#sec:simple-modules:activity, Jan. 2024.

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

[32] "Official Support for a New Coroutine Runtime in Rust," https://internals.rust-lang.org/t/official-support-for-a-new-coroutine-runtime-in-rust/3612, Mar. 2019.

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

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

[35] S. Sartor, "Generalizing Coroutines in Rust ," https://samsartor.com/coroutines-1/, Nov. 2019.

[36] Tokio, "Tokio: Semaphore," https://docs.rs/tokio/latest/tokio/sync/struct.Semaphore.html, Jan. 2024.

[37] ——, "Tokio: Channels Tutorial," https://tokio.rs/tokio/tutorial/channels, Jan. 2024.

[38] S. Ha, I. Rhee, and L. Xu, "CUBIC: a New TCP-friendly High-speed TCP Variant," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, p. 64–74, Jul. 2008.

[39] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, "BBR: Congestion-Based Congestion Control," *ACM Queue*, vol. 14, no. 5, p. 20–53, Oct. 2016.

[40] Y. Zhu, D. Firestone, C. Guo, J. Padhye, S. Raindel, M. Zhang, Y. Liron, H. Eran, M. H. Yahia, and M. Lipshteyn, "Congestion Control for Large-Scale RDMA Deployments," in *Proc. ACM SIGCOMM 2015 Conference*. ACM, 2015, p. 523–536.

[41] S. Barral, "Nexosim: A High-Performance, Discrete-Event Computation Framework for System Simulation," https://docs.rs/nexosim/latest/nexosim/, 2025.

[42] K. K. Ramakrishnan, S. Floyd, and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP," https://www.rfc-editor.org/rfc/rfc3168, Sep. 2001.

[43] "IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks Amendment: Priority-based Flow Control," IEEE Std 802.1Qbb-2011, 2011.

[44] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, no. 10, p. 30–53, Oct. 1990.