JADE: Just A Deterministic Emulator to Support the Verification of Protocol Implementations

Tom Rousseaux*, Alix Temmerman*, Olivier Bonaventure*†
*UCLouvain (*ICTEAM*), Louvain-la-Neuve, Belgium
†WELRI, Belgium
firstname.lastname@uclouvain.be

***Abstract*—Internet protocols continue to evolve. The verification of modern protocols such as QUIC needs to consider not only the protocol specification but also the real implementations. Verifying implementations requires the ability to execute them in controlled environments. We propose, implement, and evaluate `JADE` (Just A Deterministic Emulator), a lightweight user-space framework that is situated in the middle ground between network simulators and emulators. `JADE` enforces time determinism by interposing a minimal set of libc calls (time, randomness, and blocking I/O), advancing a global event-driven simulated clock, and buffering packet transmissions until scheduled delivery times. This selective interposition preserves binary compatibility and low extension cost. We integrate `JADE` with `Ivy`'s Network-centric Compositional Testing workflow and demonstrate deterministic, reproducible verification of picoquic, a standards-compliant QUIC implementation, including the reproduction of a known temporal bug. Compared to kernel-based emulation (`tc`), `JADE` achieves delay-agnostic execution with sub-millisecond per-packet delivery overhead; compared to the `Shadow` simulator, `JADE` offers a lower implementation complexity.**

***Index Terms*—Deterministic Emulation, Network Simulation, Protocol Verification, Model-Based Testing, QUIC**

I. Introduction

Internet protocols continue to evolve, and their validation remains an important research problem. Various techniques have been proposed and used by researchers, from mathematical models to specialized languages and support tools. Despite progress in these languages and tools, it remains important to also verify real protocol implementations. Recent efforts have addressed the verification of the Transport Layer Security protocol [1], [2].

In parallel, researchers and the IETF have developed new protocols. QUIC [3] is a new transport protocol that integrates the security features of TLS 1.3 [4] with the reliability, flow control, and congestion control features of TCP. QUIC runs above UDP and is usually implemented as a library that can be directly linked to the application code. While QUIC is a young protocol, it is already widely deployed, notably by large cloud and CDN providers [5], and serves a significant fraction of the Internet traffic. The IETF QUIC working group maintains a list of more than a dozen production QUIC implementations. Researchers have analyzed these implementations and their interoperability [6], [7].

While interoperability tests are useful, they do not allow for validating QUIC implementations. In parallel, researchers developed methodologies and supporting tools to validate protocol implementations. `Ivy` [8], [9] allows for verifying properties of protocols and implementations. It has been applied to lots of protocols (SCP [10], ABP [11], [12], TLB shootdown [11], [12], several variants of Paxos [11], [13]–[15], Chord [9], TileLink [16], several variants of Bosco [14]) and several implementations (Raft [13], Multi-Paxos [13], and QUIC [17]–[19]). It uses a Network-centric Compositional Testing (NCT) approach [18]. However, verifying QUIC implementation is not trivial [6], [20]. To verify temporal properties reproducibly requires more than a functional environment—it demands deterministic execution, where events unfold at a predictable time, in a repeatable order. Indeed, the lack of determinism implies that when a problem is identified with an implementation, there is no guarantee that it will be easy to reproduce [21]. Moreover, verifying *quantitative-time properties* (e.g., X happens less than 3s after Y) demands control on the time perceived by the processes. Meeting those requirements also brings operational soundness to NCT.

Simulation relies on abstract models of network protocols, systems, and traffic rather than executing real implementations on real systems. Because they control time progression and event scheduling, simulations are inherently deterministic and reproducible: the same sequence of events always produces the same outcome. This makes them well suited for evaluating protocol correctness, scalability, or performance across a wide design space. However, simulations trade realism for controllability, as they can only approximate the complexity of real systems.

Simulation frameworks such as `ns-2` [22], `ns-3` [23], or `omnet++` [24] are used to verify protocols. However, simulators mainly run protocol models and not complete implementations, despite some efforts to achieve this goal [25]–[27].

Section II presents these tools and their limitations when testing protocol implementations with `Ivy`. Section III presents the design of `JADE`, a deterministic emulator that overcomes the limitations of network simulators while addressing different sources of non-determinism. Section IV details the benefits and limitations of the approach. Section V argues why `JADE` is relevant for `Ivy`-testing. In Section VI, we compare the performances of `JADE` with the ones of `Shadow` and pure emulation; and we validate that `JADE`

ISBN 978-3-903176-82-9

can reproduce a temporal bug in picoquic, a standards-compliant implementation of QUIC. We conclude in Section VII.

A. Contributions

This paper makes the following contributions:

- It introduces a new lightweight method to bring determinism to time- and randomness-dependent behavior of network experimentations that use real implementations. It enforces reproducibility of experiments and allows reasoning about properties involving quantitative timing.
- An implementation of this method is proposed as a new tool called JADE. [1] It runs unmodified executables with a minimal libc interposition, while not interfering with most syscalls and delegating them to the host system, making JADE easily extensible. We compare this extensibility with other tools in Tables I
- We compare simulation time with the Shadow network simulator and pure emulation in Table II, and Figures 3 and 4
- This paper presents and justifies the soundness of this method for Ivy, a Model-Based Testing tool[2].
- This paper validates the integration of JADE within Ivy with a real usecase: verifying a real implementation of a complex protocol (picoquic, an implementation of QUIC).

II. Background and State of the Art

Ivy [8], [9] is a tool for specifying and verifying distributed systems. An Ivy model can be transpiled into a C++ program that is compiled and executed against real implementations of a protocol. It performs black-box testing over the wire with its Model-Based Testing (MBT) approach. But Ivy alone does not guarantee the reproducibility of its results. Its high computational cost can affect the time determinism of its tests [18], [19] and its capacity to verify *quantitative-time properties* [19]. Ivy is deterministic in terms of input-output, but we look for *time determinism*, i.e., given the same sequence of input events at the same timestamps, the outputs will occur at the same timestamps each run [28]. Network simulators provide controlled environments that enforce such time determinism.

ns-3 [23] is a feature-rich discrete-time network simulator widely used by the networking community [29]. It is extensible by design with user-provided extensions modeling new elements, such as additional protocols [30] or specific physical layers [31]. As this simulator relies on models, it cannot execute real-world implementations. ns-3 allows a hybrid approach by connecting emulated implementations with a simulated network through sockets [32], but this does not bring determinism to those implementations.

Direct Code Execution (DCE) [27] is an extension to ns-3 that allows the execution of real-world implementations directly within the simulator. It leverages an old Linux kernel with a modified libc[3]. Syscalls related to the network, the file system, or time are intercepted to update the state of the simulator accordingly. While DCE solves the main limitation of plain ns-3, it also has limitations. First, this extension relies on a specific version of the Linux kernel. Supporting a newer version would require heavy modifications. Second, when executing new applications for the first time in this framework, some syscalls may not be implemented yet. This forces the user to extend DCE with the appropriate syscall interception code, which requires some degree of knowledge of the ns-3 internals. Third, DCE encounters scalability limitations on medium-sized topologies [33].

Shadow [26] is a discrete-time network simulator designed to run real-world applications. The applications are run as plug-ins and require no modifications. Shadow intercepts the system calls of the applications and provides a simulated environment for them to run in. This raises the same concerns as ns-3 DCE, as it requires intercepting lots of system calls. System calls that are not natively supported need to be implemented before running the application. This can lead to a high overhead in extending the simulator to support new executables. Another limitation of Shadow is that it is designed to be used with a specific set of network protocols and may not be suitable for all use cases (e.g., raw sockets are not supported, which prevents the simulation of layer-3 protocols).

The complexity of those tools makes them difficult to extend, particularly when support for additional system calls is needed. Indeed, implementing them requires familiarity with the simulation infrastructure.

On the other hand, network emulation provides a middle ground between pure simulation and experimentation on physical testbeds. Rather than modeling protocols or systems, emulation executes real implementations while introducing artificial network conditions such as latency, loss, or bandwidth limits. For example, tools like mininet [34] leverage kernel network namespaces and tc to instantiate entire topologies on a single machine. This allows developers to run unmodified applications in a controlled environment, while preserving the realism of the execution. However, emulation remains fundamentally tied to wall-clock time: delays are actually waited, the execution progresses in real time, and resource contention on the host can directly impact the course of events. As a consequence, reproducibility cannot be guaranteed, limiting emulation's suitability for deterministic testing of time-sensitive behaviors.

We introduce a new environment called JADE. This stands for Just A Deterministic Emulator, which reflects its position between network simulation and pure emulation. JADE aims at bringing reproducibility to network emulation, with a focus

[1] it is available as open source at https://github.com/FlyearthR/JADE/tree/networking-26

[2] Ivy specifically implements Network-centric Compositional Testing (NCT), which is detailed in V-A

[3] The latest supported kernel version is 5.10 and is proposed here: https://ns-3-dce-linux-upgrade.github.io/

on implementation verification through Ivy. Precisely, we designed JADE to run Ivy-generated implementations against real hand-written ones with five goals in mind: *(i)* to be easy-to-extend to support new implementations, *(ii)* to provide determinism and reproducibility for experiments (as network simulators [19]), *(iii)* to enable Ivy to verify *quantitative-time properties* (as network simulators [19]), *(iv)* to restore operational soundness to Ivy's testing methodology, and *(v)* to remain usable as a standalone tool beyond the Ivy workflow.

III. System Design

We identified the primary sources of non-determinism in process execution. These fall into five broad categories:

Multithreading errors, such as data races, which introduce execution-order non-determinism [35]. JADE does not address them directly because they can be reliably detected using tools such as Helgrind [36] or CHESS [35].

C-style memory errors, including uninitialized memory and buffer overflows, which lead to undefined behavior. Similarly, they should be uncovered using tools like Valgrind [37] and are not supported by JADE.

Environmental variation: external system state (e.g., file contents, environment variables) changes across runs [38]. JADE mitigates those by executing the processes within a container to isolate them from host-side variability.

External sources of non-determinism, such as calls to `random()`, `time()`, or non-deterministic packet arrival timing [21]. These are addressed by JADE, which intercepts and deterministically simulates the relevant libc calls.

Computational time could vary around the internal timeout of a protocol, leading to non-deterministic exceeding of it [39]. This is abstracted by the event-driven clock of JADE, which is needed to verify *quantitative-time properties*.

This results in a time-deterministic emulator that interposes on randomness- and time-related libc calls while delegating all other calls to the container libc and host kernel. It enables black-box execution of dynamically linked binaries in a temporally controlled environment, without requiring application modification.

To achieve this, JADE introduces an event-driven simulated clock that advances in discrete steps in response to scheduled events, such as timeouts or packet arrivals. This controlled clock governs the progression of time across processes, enabling reproducible timing behavior.

As Figure 1 shows, time determinism is enforced by selectively intercepting a subset of libc calls—specifically those involving randomness, time, and network I/O. These calls are intercepted using the LD_PRELOAD mechanism [40], allowing JADE to override their behavior at runtime. An override only simulates the timing and random-related aspects of a libc call. Their functional part is delegated unmodified to the actual libc, e.g., writing to a socket or polling file descriptors. For instance, `poll()` is intercepted to control timeout progression in simulated time, but it still invokes the actual libc call with a modified (zero) timeout. This allows JADE to control time while leaving kernel interaction untouched. This selective interposition minimizes implementation effort (as discussed in Section IV) and maintains compatibility with unmodified dynamically linked binaries.

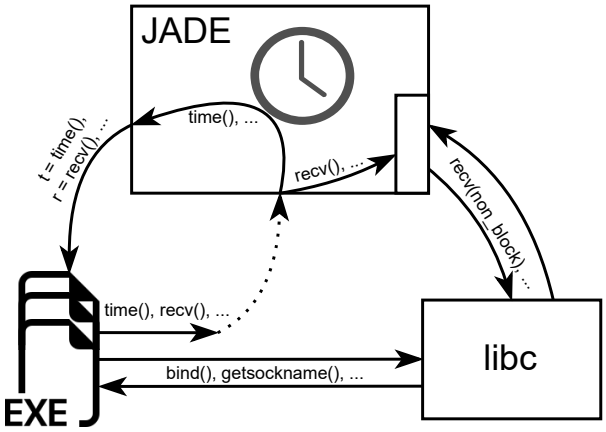

Fig. 1. Most of the libc calls behave normally, and JADE intercepts others. The time and random parts of these syscalls are simulated while the rest is delegated to the actual libc.

The intercepted functions fall into three broad categories, further detailed in Section III-B. These include functions that (*i*) interact with randomness or time-based variability, (*ii*) wait on timer expiration, or (*iii*) wait for or produce network traffic, e.g., while polling for incoming packets. By reimplementing only such functions, we ensure that JADE controls all sources of temporal uncertainty and external non-determinism, without needing to simulate the full system.

A. Event-Driven and Discrete Time

Figure 2 shows that JADE maintains a simulated clock that advances in discrete, non-constant steps, each corresponding to the earliest upcoming event in the system (e_t, e_{t+1}, and e_{t+2} in the Figure). These events may be scheduled timeouts (e.g., from `nanosleep()`, `poll()`, or `select()`) or expected network deliveries (i.e., the scheduled receipt of a packet). To preserve determinism across multiple processes, each discrete time step is divided into two distinct phases.

Network IO Phase (left bracket). As there is virtually no real delay between sending a packet and receiving it, packets that are scheduled to arrive at the current time are sent one at a time in a deterministic order. In the figure, JADE waits for packet p_γ to be sent before saying to send p_ϵ. This ensures that the order of receiving packets is consistent across executions.

Processing Phase (right bracket). Once all packets have been sent, all processes are woken up simultaneously. At this point, the simulated system clock is updated to reflect the current time, and processes may receive packets to process or timeout events corresponding to that time.

Each discrete time step concludes when all processes are blocked—or finished—, waiting for incoming data or for their timeout to expire. By ensuring that no computation or communication occurs between discrete steps, JADE maintains

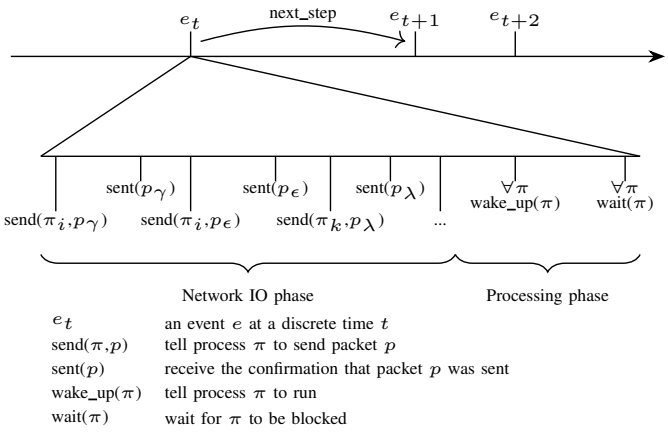

Fig. 2. Unfolding of a time step in JADE.

a clean separation between application behavior and time advancement.

The pseudocode in Algorithm 1 sketches how these phases are linked within JADE. The main emulation loop keeps running as long as there are processes that have not yet terminated *and* there remain pending events; the second condition is important because it allows JADE to terminate cleanly if the event list becomes empty (for example, if an emulated server blocks indefinitely waiting for a new connection that never arrives).

Each iteration begins by dequeuing the next event. Processing then unfolds in the two clearly separated phases. In the **Network_IO Phase** (lines 7-10), as we've seen, JADE iterates over the processes attached to the current event, instructing one at a time to send its pending packet and waiting for confirmation of the send.

Then the global simulated clock is advanced to the event's time, and every process that was previously blocked is woken simultaneously and JADE enters the **Processing Phase** (lines 12-25): it consumes messages from the processes and updates the emulation state accordingly, until all non-finished processes have either become blocked again or have terminated. Messages can indicate that a process has blocked, that it has finished execution, or that it needs to schedule a future event (e.g., for sending a packet or reaching a timeout).

In the last case (line 23), the helper function `add_event_or_process` is called. It manages the cases where either there is no event at that time yet, or there is already an event, or there is already an event and a packet from this process.

The version of the algorithm shown above is deliberately simplified. Real communication in JADE is more complex: there are many different message types used to compute per-packet delays, to make the random number generator deterministic, etc. In addition, the actual implementation contains more bookkeeping to enforce determinism and to handle corner cases. The pseudocode should therefore be read as an overview of the control flow: its two-phase structure is representative of how JADE interleaves network I/O and process execution in a deterministic manner.

Algorithm 1 JADE Simplified Algorithm

```
1:  current_event ← (time = 0, ready_to_send = ∅)
2:  event_list ← [current_event]
3:  finished ← ∅                          ▷ set of finished processes
4:  blocked ← ∅                           ▷ set of blocked processes
5:  while finished ≠ all_processes ∧ event_list ≠ [] do
6:      current_event ← event_list.dequeue()
7:      for process p in current_event.ready_to_send do
8:          Tell p to send its packet
9:          Wait for p sending confirmation
10:     end for
11:     simulated_clock ← current_event.time
12:     for all process p ∈ blocked do
13:         Wake up p
14:         blocked ← blocked \ {p}
15:     end for
16:     while blocked ≠ all_processes \ finished do
17:         m ← top message of the queue
18:         if m = "p is blocked" then
19:             blocked ← blocked ∪ {p}
20:         else if m = "p has finished" then
21:             finished ← finished ∪ {p}
22:         else if m = "p has to send at time t" then
23:             event_list.add_event_or_process(t, p)
24:         end if
25:     end while
26: end while
```

B. Categorizing Intercepted `libc` Calls

JADE intercepts only a minimal subset of `libc` calls, classified according to how they introduce non-determinism into a program execution. We define three categories.

Randomness or Time-based Variability. This category includes functions that return values derived from system state outside the program's control. That is the case, for example, for `getrandom()`, `gettimeofday()`, or `clock_gettime()`. Their output can vary across executions, even with identical inputs. This is due to dependence on external sources such as the wall clock or kernel random number generator. For such functions, JADE fully replaces the result with a deterministic value, drawn from a seeded pseudo-random generator or computed from the simulated clock.

Blocking for a Fixed Amount of Time. Functions such as `sleep()` or `nanosleep()` belong to this category. These are fully simulated by appending a future event to the simulated clock and waiting until that event is reached, without invoking any delay in the real system. As a result, the passage of time is virtualized, decoupling application behavior from host performance.

Waiting for or Producing Network Traffic. This category includes calls such as `send()`, `recv()`, `poll()`, or `select()`. Unlike the previous categories, these calls are

only partially simulated. Their blocking behavior is adjusted to align with the simulated time, but their core functionality—sending or receiving data—relies on the actual `libc` and kernel implementation. For instance, `send()` is intercepted to delay packet transmission until the scheduled delivery time, but the actual `libc send()` function is used to transmit the data. Similarly, `recv()` is rewritten and reinvoked with a non-blocking flag at each simulated time step until a packet is received.

Some functions fall into multiple categories. In such cases, each behavioral component is handled according to its category. For example, `poll()` waits either for a network event or a timeout. Its network part is delegated to the real kernel in a non-blocking fashion. In practice, `libc poll()` is called with a timeout of zero at each simulated time step until an event occurs. The timeout part is simulated with a scheduled event. When `JADE`'s clock reaches the timeout event, the intercepted function returns. This modular treatment allows controlling the perceived time without reimplementing the full system behavior.

C. Modeling Link Delays

Simulating network latency in a user-space emulator requires control over when packets are delivered. Kernel-level mechanisms such as `tc` (traffic control) cannot be used in this context because they operate entirely within the kernel and are unaware of the simulated time. Instead, `JADE` buffers outbound packets in user space and delays their actual transmission until the simulated clock reaches the packet's scheduled receive time. Specifically, when a process issues a call to `send()` or similar, the data is captured and placed into a delay queue inside the same process. The actual transmission of the packet occurs later, when `JADE` determines—according to the simulated link delay and the global simulated time—that the packet has reached its destination.

This mechanism allows full control over network communication timing, including the simulation of asymmetric delays or jitters. It also ensures that packet delivery is both repeatable and deterministic. Currently, links throughput are considered infinite, but their value could be taken into account in the computation of receiving time. This is left as future work.

IV. Implementing JADE

The main goals of `JADE` are controlling non-determinism and abstracting away computational time *via* minimal `libc` interposition and real system execution outside the temporal and random domains. Its core is implemented in Rust (1977 LOC), and it provides a C API (383 LOC) to help implement `libc` calls (652 LOC of C). `JADE` is available at https://github.com/FlyearthR/JADE/tree/networking-26.

The approach enables this lightweight architecture presented in Section III, which avoids the complexity of traditional full-system simulators discussed in Section II. We discuss the main benefits and limitations of our simple architecture in the remainder of this section.

Low Barrier for Extending Functionality. `JADE` is designed to be easily extensible with minimal developer overhead. To support a new `libc` call, a developer only needs to understand (*i*) the documented behavior of that specific call (its man page), (*ii*) how it interacts with time or randomness, and (*iii*) high-level operations of an event-driven simulation. There is no need to inspect the implementation of the function or reason about low-level system behavior, as either the `libc` call to implement is very simple—e.g., `sleep()` just checks the current time and advertises the process as blocked until the end of the timeout— or the complex part remains implemented in the `libc`—e.g., `recv()` only calls the `libc recv()` with a non-blocking flag at each time step. Furthermore, since `JADE` does not simulate the kernel, developers do not need to understand the internals of packet transmission or system call scheduling.

To assess the simplicity of `libc` functions implementation, we compared `JADE` against `Shadow` and `ns-3 DCE`. Table I reports, for each tool, the number of `libc` functions implemented in common with `JADE`, the implementation language, and the average lines of code (LOC) per function. We also include the corresponding average LOC required in `JADE` for the same functions. Our tool consistently requires fewer lines of code and has an average of 21 LOC per intercepted `libc` call.

	Shadow[4]		ns-3 DCE	JADE
Number of functions	10	4	22	31
Language	Rust	C	C/C++	C
Average lines of code per function	56	84	25	21
JADE lines of code per function	24	26	21	21

TABLE I
COMPARISON OF LIBC FUNCTIONS IMPLEMENTED IN SHADOW, NS-3 DCE, AND JADE, IN TERMS OF THE NUMBER OF COMMON FUNCTIONS, IMPLEMENTATION LANGUAGE, AVERAGE LINES OF CODE (LOC) PER FUNCTION, AND JADE LOC FOR THE COMMON FUNCTIONS.

Ease of Keeping in Sync with `libc`. Another advantage of the approach is to simplify staying up to date with the evolving `libc`. Since `JADE` delegates the majority of system behavior to the actual `libc` and kernel, only the simulated portions—primarily related to time and randomness—require attention when `libc` updates are released. Bug fixes or changes in the behavior do not need to be reflected in the implementation of the `libc` calls, as they rely on the `libc` implementation. Calls that introduce randomness or have a timeout are the only calls simulated, and they are unlikely to change behavior, due to their simplicity. Moreover, only changes in the syscall interface could have an impact on simulated (parts of) `libc` calls, which would break `libc` backward compatibility assurance [41]. In practice, this means that maintenance is localized to a narrow and well-understood subset of the `libc` interface[5]. Developers need only review

[4] `Shadow` mixes Rust and C, so the `libc` functions were grouped by language.

[5] During our tests and evaluations of `JADE`, we met 129 different `libc` calls. We had to implement 23 of them, leaving 106 pass directly to the `libc`.

changes to the specification of relevant calls (e.g., if the behavior of `get_time()` or `nanosleep()` is updated) and adjust JADE's logic accordingly. The rest of JADE remains agnostic to changes in internal implementations.

Binary Compatibility and Black-Box Testing. Since JADE operates via dynamic interposition, it supports black-box testing of unmodified binaries. This is useful in regression testing, fuzzing, and conformance-checking scenarios where source code may be unavailable or undesirable to modify. It also enables testing of real-world implementations without the need for recompilation or source-level instrumentation.

A. Limitations

While JADE proposes a practical solution for enforcing time determinism in internet system testing, it has several limitations.

Dynamic Linking Requirement. JADE depends on dynamic linking—as `ns-3 DCE` and `Shadow`—to interpose on `libc` functions. Indeed, static compilation embeds system calls directly in the binary, bypassing the `libc` interface. While this is effective for many applications, especially those written in C/C++ that dynamically link the `libc` by default, further analysis should evaluate the methods with Go, Rust, and interpreted languages.

Non-blocking Sockets and Infinite Polling. The current simulation model does not natively support non-blocking sockets used in busy-loops. In such configurations, applications may enter tight polling loops that stall the simulated time progression, potentially resulting in infinite loops. `Shadow` has an option to address such cases, incrementing the simulated clock on each system call[6], which could be implemented in JADE.

Lack of Kernel-level Time Interposition. JADE cannot intercept or influence kernel-level timing behavior. For example, TCP relies on timers managed within the kernel (e.g., for retransmissions), and their behaviors are not controllable through `libc` interposition alone. Consequently, our tool cannot deterministically simulate protocols with kernel-managed timing logic. One direction for addressing this limitation is to replace kernel TCP with a user-space TCP stack (as [42]), enabling full control over temporal behavior.

Sensitivity to Bad Memory Management. Because JADE enforces determinism only at the level of time and randomness, it assumes that the processes are free from memory access bugs, such as data races or bad pointers. In multithreaded applications, data races could reintroduce non-deterministic behaviors, but existing tools—Valgrind and Helgrind—can detect such programming errors.

V. JADE works well with Ivy

While the construction of JADE abstracts computational time and introduces determinism, it also enables another interesting property in the context of Ivy. Indeed, it repairs an implicit timing gap in the original Ivy soundness statement: Ivy's randomized compositional testing assumes that every enabled event has a non-zero probability of being sampled, but this fails in practice when solver computational time exceeds protocol deadlines. By bringing virtualized timing into Ivy, JADE closes this gap and restores operational soundness in the presence of time-sensitive failures. This extension relies on a fundamental property of JADE: its explicit control of timing, which abstracts away host-dependent computational delays.

A. NCT

Network-centric Compositional Testing (NCT) [18] is a methodology for expressing formal wire-level specifications of Internet protocols and testing implementations against them. It builds directly on Ivy, which provides the formal specification language and implements the randomized testing mechanism. NCT departs from assume–guarantee reasoning to escape per-process specifications; instead, it relies on a single global specification ϕ of the protocol, expressed as a deterministic guarded-command system over traces of network events. It relies on a theorem stating that if no componenent π of a closed system Π breaks ϕ, then their composition guarantees ϕ.

$$\frac{\forall \pi \in \Pi : \quad \langle \phi \rangle \pi \langle \phi \rangle}{\Pi \models \phi} \tag{1}$$

Corollarily, if the global specification breaks, then at least one component has broken it. As Ivy models follow the specification by design, it is the tested component that breaks it. This global perspective is important in a network setting, where processes cannot be reliably identified, addresses may change dynamically, and even malicious spoofing is possible. From such global specifications, Ivy generates mirrors—adversarial environments that stimulate an implementation under test while checking compliance. Thus, Ivy operationalizes NCT by compiling formal global specifications into executable randomized testers.

B. Soundness

McMillan & Zuck define NCT soundness in the following sense: if the closed system violates the global safety property ϕ, then some finite failure trace exists; and if the sampling distribution over enabled events is everywhere non-zero, this failure will eventually be sampled with probability one [18]. However, this argument breaks down in practice. Timing-sensitive failures[7] introduce deadlines Δ that may be shorter than Ivy's computational cost[8] T_{solve}. When $P[T_{solve} \leq \Delta] = 0$, entire classes of timed failures become unreachable in the sampling space, voiding the soundness guarantee. Thus, Ivy's compositional testing over real implementations is not sound.

This happens because equation 1 requires a closed system, and Ivy testers and implementations under tests do not form

[6]https://shadow.github.io/docs/guide/limitations.html#busy-loops

[7]A time-sensitive failure can be caused by the violation of a non-time-sensitive property as well as a time-sensitive one. Indeed, a non-time-sensitive property could break in a path of execution followed only for a specific timing.

[8]Ivy uses constraint solving to produce packets. Its solver takes a significant computational time [18].

one as soon as the specification contains *quantitative-time-properties*. Those are very common for networked applications and especially for network protocols, which have to deal with link latencies. A closed system should integrate the system clock Σ:

$$\frac{\forall \pi \in \Pi : \quad \langle \phi(t < \Delta) \rangle \ \pi \ \langle \phi(t) \rangle \qquad \langle \phi(t) \rangle \ \Sigma \ \langle \phi(t + T_{solve}) \rangle}{\Pi \| \Sigma \models T_{solve} < \Delta \implies \phi} \tag{2}$$

With that, `Ivy` loses the property that a break of the specifications is attributable to the tested component, as it could also be attributed to the evolution of the system clock.

C. Back to soundness

`JADE` resolves this limitation through two mechanisms. First, it virtualizes computational time: the simulated clock advances independently of host solver latency ($T_{solve} = 0$), so deadlines Δ are always respected in simulated time. This ensures that timed behaviors remain reachable, regardless of solver delays. The system clock can be modeled as $\langle \phi(t) \rangle \Sigma \langle \phi(t) \rangle$. Second, `JADE` schedules packet deliveries using delays drawn from a distribution with everywhere non-zero probability. This satisfies the "full support" hypothesis required in the original NCT soundness proof [18], guaranteeing that all finite failure traces remain observable with non-zero probability. So the system clock can be refined as $\langle \phi(t) \rangle \Sigma \langle \phi(t + \tau) \rangle$, with full control over the value of τ and $\forall \delta : P[\tau \leq \delta] > 0 \wedge P[\tau \geq \delta] > 0$. This allows `Ivy` to deterministically test pre- and post-deadline behaviours while being aware of it. Together, these properties restore the operational soundness of `Ivy` in testing timing-sensitive network protocol implementations.

VI. Evaluation

Table I illustrates the implementation simplicity of `JADE` compared to network simulators. In this section, we evaluate its effectiveness quantitatively and qualitatively. The experiments were on a server running Ubuntu 22.04 with 2 Intel Xeon CPU E5-2687W v3 and 128 GB of RAM. Quantitatively, we benchmark `JADE` against pure emulation using `tc` and pure simulation using `Shadow`[9]. Qualitatively, we demonstrate that `JADE` supports deterministic, reproducible Model-Based Testing (MBT) with `Ivy` using its *quantitative-time-property* extension. In particular, we show that `JADE` can deterministically reproduce a known temporal bug in `picoquic`, a QUIC protocol implementation, using an `Ivy`-generated test case [19].

Performance Comparison with `tc` and `Shadow`. First, we compare the execution performance of `JADE` against that of pure emulation using `tc` and pure simulation using `Shadow`. Both implementations under test follow a simple client–server architecture based on a request–response exchange.

[9]Due to the `ns-3 DCE` constraint on the kernel version, we couldn't run `JADE`, `Shadow`, and `ns-3 DCE` on the same setup and chose to compare `JADE` with `Shadow`, which is more recent and still maintained.

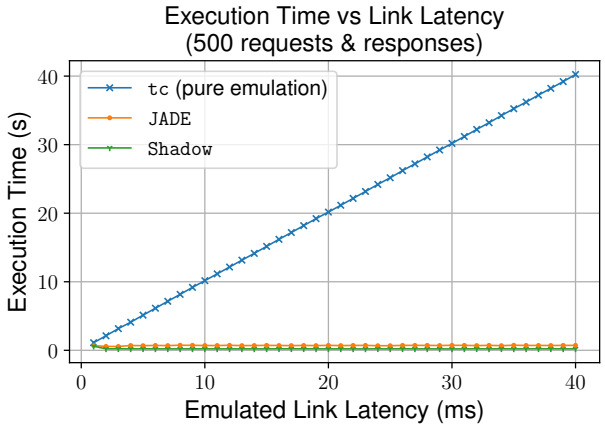

Fig. 3. Execution time of `JADE`, `Shadow`, and `tc`-based emulation as a function of link delay.

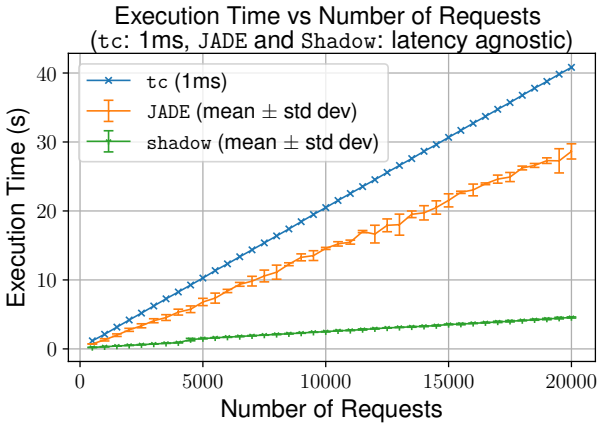

Fig. 4. Execution time of `JADE`, `Shadow`, and `tc`-based emulation with increasing number of requests.

Figure 3 shows the total execution time as a function of link latency. With `tc`, the execution time increases linearly with the delay, since the process must wait in real time before each packet is delivered. By contrast, `JADE` and `Shadow` exhibit delay-agnostic performance: they immediately advance to the next discrete time step, regardless of the latency value.

In a second experiment (Figure 4), we assess how `JADE` scales with the number of request-response pairs. The three curves follow roughly a theoretical line expressed by $T(n_{req}) = T_{init} + 2 * n_{req} * (T_{link} + T_{comput})$ with T_{init} the initialization time of the setup, n_{req} the number of requests, T_{link} the packet delivery time, and T_{comput} the computation time of processes.

In Figure 3, we see that `tc` follows this rule perfectly with $T_{link}+T_{comput}$ matching the link latency (the implementations being simplistic, T_{comput} seems negligible) and that `JADE` performs a sub-millisecond packet transfer. Its performance, as its design, is clearly positioned in the middle ground between

emulation and simulation.

In Table II, we calculate the estimators of those parameters from measurements. We compute a least square regression for the pure emulation data, knowing T_{link}. This gives us one estimate for $T_{compute}$, which we assumed was the same for JADE and Shadow, and one for T_{init}. Then we can use the same regression to find the values for JADE and Shadow. Although JADE is 6.4 times slower than Shadow, we see

	Shadow	JADE	emulation
$\hat{T}_{init}$ [ms]	235	115	136
$\hat{T}_{link}$ [ms]	0.1	0.64	T_{link}
$\hat{T}_{comput}$ [ms]	0.02	0.02	0.02

TABLE II
ESTIMATORS OF EXECUTION TIME OF SHADOW, JADE, AND PURE EMULATION FROM THE FUNCTION
$T(n_{req}) = T_{init} + 2 * n_{req} * (T_{link} + T_{comput})$

that it indeed performs a sub-millisecond packet transfer, being significantly faster than emulation. The difference in initialization time can be explained *(i)* because Shadow has to set up a more complex architecture than JADE (which only sets up network namespaces and veth pairs), and *(ii)* because our emulation script has to set up the tc configuration and has unoptimized netlink calls.

Compared to Shadow, JADE incurs a higher packet delivery time because packets are actually transmitted through the host network stack rather than being simulated in user space. Each message traverses the full kernel networking path, introducing overhead that Shadow avoids by short-circuiting packet delivery inside the simulator. Shadow was designed from the outset to simulate large-scale networks [26] and has undergone years of optimization [43].

Deterministic MBT of Temporal Properties. Next, we evaluate JADE through the testing of several implementations of a toy protocol designed to exhibit simple temporal properties. The protocol follows a client-server architecture over UDP, where the client sends requests and the server must respond within 3 seconds. That makes it a suitable target for evaluating time determinism: it is easy to reason about the correctness of each event in the exchange and to compare their timing across runs.

We use Ivy to generate model-based test cases to test various implementations, including intentionally faulty variants that delay their responses beyond the deadline. JADE allows for detecting these bugs consistently. Since it virtualizes time and controls the delivery schedule of packets, every test execution proceeds identically, reproducing the same series of timeouts and packet exchanges. Faults due to late responses are consistently uncovered.

To evaluate JADE on a real-world use case, we reproduced the experiments of [19], which combined Ivy's timing extension with Shadow to check *quantitative-time properties* of QUIC. In that study, the authors used picoquic, a standards-compliant QUIC implementation, and uncovered a temporal bug that caused some connections to close prematurely due to the miscalculation of a timeout. We recreated the same setup but replaced Shadow with JADE. The bug was reproduced deterministically: packet logs, encryption keys, and timings were identical across runs, and the Ivy model consistently triggered the same error. This shows that JADE provides the same time determinism guarantees as Shadow, while requiring significantly less implementation complexity (as quantified in Table I). These results validate JADE's intended use case: enabling Ivy to test *quantitative-time properties* of Internet protocol implementations deterministically and reproducibly while being easy-to-extend to support new implementations.

VII. CONCLUSION

We presented JADE, a simple deterministic emulator designed to help test the temporal behavior of network protocol implementations. JADE interposes on a small subset of libc calls to simulate only time and randomness, delegating other calls to the host. This approach enables black-box, reproducible testing of real-world binaries with minimal developer effort.

	ns-3 DCE	Shadow	JADE	Emulation
Extension difficulty	hard	hard	easy	none
libc functions subset	medium	big	small	none
Execution speed	slow[10]	fastest	fast	slowest
Kernel version	5.10	5.10+	3.10+	2.6.24+
Deterministic	✓	✓	✓	✗
Statically linked binaries	✗	✗	✗	✓
Time abstraction	✓	✓	✓	✗
Layer 3 protocols	✓	✗	✓	✓
Layer 4 protocols	✓	✓	UDP-based[11]	✓

TABLE III
COMPARISON OF JADE WITH NS-3 DCE, SHADOW AND PURE EMULATION.

This offers a practical middle ground between network simulators and pure emulation. Table III shows that JADE draws positive aspects of both worlds (ease of extension from emulation; determinism and temporal efficiency from simulation) while addressing weaknesses of specific simulators (no fixed kernel version as ns-3 DCE, and layer-3 protocol support unlike Shadow). The design of JADE also comes with drawbacks: no native support for in-kernel timouts (e.g., the ones in TCP), and the requirement of dynamic linking for emulated processes.

We also demonstrated JADE's effectiveness through both synthetic and real-world use cases. In particular, it supports deterministic model-based testing using Ivy and successfully reproduced a known timing bug in picoquic, a standards-compliant QUIC implementation. It brings soundness back to Ivy. Microbenchmark results further confirmed that JADE scales well with both packet count and latency, offering faster-than-real-time simulation with sub-millisecond packet transfer.

[10]Because of environement incompatibility, we did not compare ns-3 DCE with JADE and Shadow, but we rely on the result of [33].

[11]The UDP-based support of JADE is a design choice to simplify the implementation, but it could be extended to support TCP-based protocols as well. See IV-A for details.

While JADE is limited by its reliance on dynamic linking and its inability to control kernel-level timing behavior, its simplicity and extensibility make it a compelling option for testing time-sensitive network protocols.

It met our five goals: *(i)* to be easy-to-extend to support new implementations, *(ii)* to provide determinism and reproducibility for experiments, *(iii)* to enable Ivy to verify *quantitative-time properties*, *(iv)* to restore operational soundness to Ivy's testing methodology, and *(v)* to remain usable as a standalone tool beyond the Ivy workflow.

REFERENCES

[1] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the tls 1.3 standard candidate," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 483–502.

[2] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu, "Verified cryptographic implementations for tls," *ACM Transactions on Information and System Security (TISSEC)*, vol. 15, no. 1, pp. 1–32, 2012.

[3] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[4] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[5] J. Zirngibl, F. Gebauer, P. Sattler, M. Sosnowski, and G. Carle, "Quic hunter: Finding quic deployments and identifying server libraries across the internet," in *International Conference on Passive and Active Network Measurement*. Springer, 2024, pp. 273–290.

[6] K. K. Ang, G. Farrelly, C. Pope, and D. C. Ranasinghe, "An automated blackbox noncompliance checker for quic server implementations," *arXiv preprint arXiv:2505.12690*, 2025.

[7] M. Piraux, Q. De Coninck, and O. Bonaventure, "Observing the evolution of quic implementations," in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*, 2018, pp. 8–14.

[8] K. L. McMillan and O. Padon, "Ivy: A multi-modal verification tool for distributed algorithms," in *CAV 2020*. Springer, 2020, pp. 190–202.

[9] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: safety verification by interactive generalization," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 614–630.

[10] G. Losa and M. Dodds, "On the formal verification of the stellar consensus protocol," in *2nd Workshop on Formal Methods for Blockchains (FMBC 2020)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 9–1.

[11] O. Padon, J. Hoenicke, G. Losa, A. Podelski, M. Sagiv, and S. Shoham, "Reducing liveness to safety in first-order logic," in *POPL*, vol. 2. ACM New York, NY, USA, 2017, pp. 1–33.

[12] O. Padon, J. Hoenicke, K. L. McMillan, A. Podelski, M. Sagiv, and S. Shoham, "Temporal prophecy for proving temporal properties of infinite-state systems," *Formal Methods in System Design*, vol. 57, pp. 246–269, 2021.

[13] M. Taube, G. Losa, K. L. McMillan, O. Padon, M. Sagiv, S. Shoham, J. R. Wilcox, and D. Woos, "Modularity for decidability of deductive verification with applications to distributed systems," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 662–677.

[14] I. Berkovits, M. Lazić, G. Losa, O. Padon, and S. Shoham, "Verification of threshold-based distributed algorithms by decomposition to decidable logics," in *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*. Springer, 2019, pp. 245–266.

[15] O. Padon, G. Losa, M. Sagiv, and S. Shoham, "Paxos made epr: decidable reasoning about distributed protocols," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–31, 2017.

[16] K. McMillan, "Modular specification and verification of a cache-coherent interface," in *FMCAD 2016*. IEEE, 2016, pp. 109–116.

[17] C. Crochet, T. Rousseaux, M. Piraux, J.-F. Sambon, and A. Legay, "Verifying quic implementations using ivy," in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, 2021, pp. 35–41.

[18] K. L. McMillan and L. D. Zuck, "Compositional testing of internet protocols," in *2019 IEEE Cybersecurity Development (SecDev)*. IEEE, 2019, pp. 161–174.

[19] T. Rousseaux, C. Crochet, J. Aoga, and A. Legay, "Network simulator-centric compositional testing," in *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 2024, pp. 177–196.

[20] Y. Zhou, "Towards scalable automated program verification for system software," 2025.

[21] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble, "Ddos: taming nondeterminism in distributed systems," *ACM SIGPLAN Notices*, vol. 48, no. 4, pp. 499–508, 2013.

[22] N. Simulator, "The network simulator–ns-2," *URL: http://www.isi.edu/nsnam/ns*, 2009.

[23] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[24] A. Varga, "A practical introduction to the omnet++ simulation framework," in *Recent advances in network simulation: the OMNeT++ environment and its ecosystem*. Springer, 2019, pp. 3–51.

[25] J. S. Ivey, B. P. Swenson, and G. F. Riley, "Simulating networks with ns-3 and enhancing realism with dce," in *2017 Winter Simulation Conference (WSC)*. IEEE, 2017, pp. 690–704.

[26] R. Jansen and N. Hooper, "Shadow: Running tor in a box for accurate and efficient experimentation," Tech. Rep., 2011.

[27] H. Tazaki, F. Uarbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous, "Direct code execution: Revisiting library os architecture for reproducible network experiments," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013, pp. 217–228.

[28] S. Fischmeister and I. Lee, "Temporal control in real-time systems: Languages and systems," *Handbook of Real-Time and Embedded Systems*, pp. 10–1, 2007.

[29] L. Campanile, M. Gribaudo, M. Iacono, F. Marulli, and M. Mastroianni, "Computer network simulation with ns-3: A systematic literature review," *Electronics*, vol. 9, no. 2, p. 272, 2020.

[30] A. De Biasio, F. Chiariotti, M. Polese, A. Zanella, and M. Zorzi, "A quic implementation for ns-3," in *Proceedings of the 2019 Workshop on ns-3*, 2019, pp. 1–8.

[31] D. Magrin, S. Avallone, S. Roy, and M. Zorzi, "Validation of the ns-3 802.11 ax ofdma implementation," in *Proceedings of the 2021 Workshop on ns-3*, 2021, pp. 1–8.

[32] J. Iyengar and M. Seemann, "quic-network-simulator," https://github.com/quic-interop/quic-network-simulator, 2025.

[33] N. Rybowski and O. Bonaventure, "Evaluating ospf convergence with ns-3 dce," in *Proceedings of the 2022 Workshop on ns-3*, 2022, pp. 120–126.

[34] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010, pp. 1–6.

[35] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *OSDI*, vol. 8, no. 2008, 2008.

[36] "Helgrind: a thread error detector," https://valgrind.org/docs/manual/hg-manual.html, accessed: 2025-09-15.

[37] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," *ACM Sigplan notices*, vol. 42, no. 6, pp. 89–100, 2007.

[38] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, "Deterministic process groups in {dOS}," in *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010.

[39] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *International Workshop on Embedded Software*. Springer, 2001, pp. 166–184.

[40] K. Pulo, "Fun with ld_preload," in *linux. conf. au*, vol. 153, 2009, p. 103.

[41] I. Free Software Foundation, "The gnu c library," https://www.gnu.org/software/libc/, 2023, accessed: 2025-05-23.

[42] H. J. Chu and Y. Liu, "User space tcp-getting lkl ready for the prime time," *Linux Netdev*, vol. 1, 2016.

[43] R. Jansen, J. Newsome, and R. Wails, "Co-opting linux processes for {High-Performance} network simulation," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 327–350.