Minha: Large-Scale Distributed Systems Testing Made Practical

Nuno Machado
Teradata & INESC TEC
nuno.machado@teradata.com

Francisco Maia
INESC TEC
franciso.a.maia@inesctec.pt

Francisco Neves
INESC TEC & U.Minho
franciso.t.neves@inesctec.pt

Fábio Coelho
INESC TEC & U.Minho
fabio.a.coelho@inesctec.pt

José Pereira
INESC TEC & U.Minho
jop@di.uminho.pt

Abstract

Testing large-scale distributed system software is still far from practical as the sheer scale needed and the inherent non-determinism make it very expensive to deploy and use realistically large environments, even with cloud computing and state-of-the-art automation. Moreover, observing global states without disturbing the system under test is itself difficult. This is particularly troubling as the gap between distributed algorithms and their implementations can easily introduce subtle bugs that are disclosed only with suitably large scale tests.

We address this challenge with Minha, a framework that virtualizes multiple JVM instances in a single JVM, thus simulating a distributed environment where each host runs on a separate machine, accessing dedicated network and CPU resources. The key contributions are the ability to run off-the-shelf concurrent and distributed JVM bytecode programs while at the same time scaling up to thousands of virtual nodes; and enabling global observation within standard software testing frameworks. Our experiments with two distributed systems show the usefulness of Minha in disclosing errors, evaluating global properties, and in scaling tests orders of magnitude with the same hardware resources.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies; Software and its engineering → Software testing and debugging

Keywords and phrases Distributed software testing; Large scale distributed systems; Simulation

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.32

Funding This work is financed by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia within project: UID/EEA/50014/2019.

1 Introduction

Formal validation and verification tools are increasingly practical and of paramount importance in developing distribute algorithms. However, they work over models rather than actual runnable code [21, 8] and with assumptions that do not hold in practice [11]. In addition to outright bugs introduced by the translation from algorithm to runnable code, such as race conditions, a major problem lies in aspects that are abstracted by models and are revealed

© Nuno Machado and Francisco Maia and Francisco Neves and Fábio Coelho and José Pereira;
licensed under Creative Commons License CC-BY

only in large scale tests [19]. A common problem is a situation in which the complexity of an operation (e.g., searching) is not apparent in testing as some data structure (e.g., a buffer) is mostly empty unless there is a large number of participants or with congestion, that never happens with small scale testing.

For example, the correctness proof of the accrual failure detector employed by Cassandra [15] assumed a negligible processing time of heartbeat messages. However, in practice, Cassandra uses variable length messages during membership changes in the cluster that may consume significant transmission and computation time. This mismatch between protocol design and implementation caused constant *flapping* problems in 500+ node deployments, preventing the cluster from stabilizing and scaling [3]. Flapping is a cluster instability problem where the status of the nodes is continuously switching between up and down.

Unfortunately, testing and debugging such systems is extremely challenging, mainly due to the following reasons:

Non-determinism and huge state space. Distributed executions often entail thousands of non-deterministic, concurrent events (e.g. message arrivals, node crashes, and timeouts) that, depending on the order in which they occur, cause the system to behave differently. Although most event sequences are correct, some incorrect timings of events result in severe damage, such as data corruption or system downtime [23]. In particular, previous work has shown that real-world distributed applications are especially prone to bugs stemming from *message races* [23, 26]. For instance, bug 2212 in ZooKeeper's issue repository reports a (rare) scenario in which a new node joining the cluster cannot become a leader due to a race condition between a message from the atomic broadcast protocol and one from the leader election protocol [4]. This bug causes a 3-node ZooKeeper cluster to stop working in the presence of a single failure, when in fact the existence of a majority of two nodes alive is sufficient for the system to operate.

Lack of resources for testing at scale. Distributed systems are typically developed to be deployed on a massive number of independent nodes (e.g. Cassandra [1], Hadoop [2], etc). Alas, as testing with close-to-production conditions can be prohibitively costly and time consuming, these applications are often debugged on small/medium-size deployments that prevent certain faulty behavior from manifesting [24]. Attempts to address scalability in testing environments include cloud computing and virtualization technologies [14, 38], but software validation is still the limiting factor in distributed software development [12].

Difficulty in checking global properties at runtime. In general, large-scale distributed protocols are designed in such a way that nodes make decisions based only on local information (or a partial view of the system). On the other hand, the correctness of these protocols often implies certain global properties or invariants to hold, which can be hard to verify without a globally-consistent snapshot of the system. For example, Pastry [33] is a distributed hash table whose routing algorithm requires each node to maintain a list with its physically closest peers. Thus, to assess the correct execution of Pastry, one needs to first obtain the complete network overlay, then collect the neighbor lists from all nodes and, finally, compute the distances in both cases to check whether the references in the lists actually correspond to the closest peers.

Contribution. In this paper we propose Minha,[1] a framework that combines virtualization and simulation techniques for making large-scale distributed systems testing practical by

[1] Minha is available as open source at `http://www.minha.pt`.

providing a unique trade-off between scale and observation completeness. In particular, Minha addresses the problems introduced in the translation from algorithm to runnable code and message races, such as experienced in Cassandra [3] and Zookeeper [4], with two contributions:

Scaling-up centralized simulation. The technique proposed in CESIUM [6] is scaled up to thousands of distributed nodes by reducing the processing and memory requirements for node isolation. Moreover, it is scaled to execute off-the-shelf distributed applications written in modern Java while simulating key environment components, reproducing the concurrency, distribution, and performance characteristics of real-world deployments.

Providing meta-interfaces for observation and automation. Minha provides a programming interface to orchestrate large-scale virtual deployments of complete programs or standalone middleware layers with application stubs, aimed at being used within standardized testing frameworks. The same interface eases the collection of consistent snapshots and traces from distributed executions, suitable for visualization or evaluating global properties.

We evaluated Minha on a peer sampling service protocol and a large-scale key-value store. The results show that Minha is not only able to assess properties over a coherent, global snapshot of the system without imposing runtime overhead while at the same time allowing tests with a large number of nodes to run in cost effective way.

The rest of this paper is structured as follows. Section 2 outlines the state-of-the-art in simulation and emulation for testing distributed systems. Section 3 describes how the design and implementation of Minha provide a new trade-off between scale and observation completeness and scale. Sections 4 and 5 evaluate Minha. Section 6 concludes the paper.

2 Related Work

The ideal approach to test distributed systems software for race conditions is the use of implementation-level model checkers. Model checkers, such as MaceMC [18], Demeter [13], MoDist [39], dBug [35] and SAMC [22], intercept non-deterministic events of local and distributed programs (e.g. message arrivals, node crashes, and timeouts), and permute their ordering in systematic runs. This approach is very effective in discovering concurrency bugs, as it virtually explores the whole system state space. However, for large-scale applications, distributed model checking becomes unpractical and starts suffering from scalability issues due to state space explosion [22].

The next best approach is to run tests and then evaluate system-wide properties by logging the local state of each node independently along the execution and, periodically, send those logs to a centralized machine to be combined into a globally-consistent snapshot of the system that allows checking the desired predicates. For deployed systems, D^3S [27] proposes a simple language for writing distributed predicates that are checked on-the-fly at runtime. DCatch [26], in turn, aims to detect distributed concurrency bugs by resorting to a *happens-before (HB)* model. This work encodes the event causality into HB rules and builds a graph representing the timing relationships of several distributed concurrency and communication mechanisms. This approach has some drawbacks though. First, the execution details to be recorded at runtime must be defined *a priori*. Second, monitoring the nodes' local state is intrusive and induces both performance and space overhead, and finding the sweet spot between overhead and the necessary amount of information to be traced is far from trivial [27]. Third, setting up and running large scale tests requires a corresponding large scale distributed infrastructure to be available.

An important class of bugs introduced in the translation from algorithms to implementations, for instance, caused by the complexity of library operations and data structures used, can often be disclosed by running the system at large scale. Distributed systems researchers have been building platforms to address this challenge. EmuLab [16] provided a set of dedicated computer nodes and networking hardware that could be reconfigured to mimic different large scale systems. PlanetLab [32] uses a decentralized approach, therefore enabling a much larger and realistic platform to run off-of-the-shelf code. Unfortunately, experiments in PlanetLab are cumbersome to configure, deploy, and run. Splay [25] aims at easing the task of configuring the environment and running large-scale experiments, but limits the development to a specific framework and the Lua language.

The use of virtual machines and containers in public clouds, together with state-of-the-art orchestration software [12] makes it much easier for any developer to set up and run large tests. However, virtual nodes compete over the same physical resources, hence impacting negatively the performance of the system being evaluated and the accuracy of the measurements [34], possibly hiding the problems. Moreover, even if possible, it is still expensive to conduct extensive testing in public clouds.

Given the cost of running large scale tests and the difficulties in obtaining reliable, reproducible results, a number of proposals have focused on exploiting simulation to test actual implementations. Simulation is used extensively for distributed systems research, allowing simplified models to be tested in a very large scale [31], but they don't capture timeliness properties of implementation decisions and require code to be written in event-driven style, hence, with inversion of control. As an example, Neko [36] offers the ability to use simulation code as actual code, as long as its event-driven API is used in place of the standard Java classes.

An interesting trade-off is achieved by JiST [7] (Java in Simulation Time), a simulation kernel that allows event-driven simulation code to be written as Java threaded code, but avoids the overhead of a native thread for each simulated thread by using continuations. Unfortunately, this simulation kernel does not virtualize Java APIs and thus cannot be used to run most of the existing Java code. Moreover, it does not reflect the actual overhead of Java code in simulation time.

CESIUM [6] proposes the centralized simulation approach, in which the time effectively used to execute implementation code for each event is measured and reflected into simulation time. This is useful to detect issues such as the Cassandra failure detector bug [3], as it captures the timeliness properties of implementation code. This does not however address the general Java platform API and thus does not allow general code to be run. Moreover, by using Java class loaders for virtualization it imposes a large memory overhead and restricts simulations to a small number of nodes. UMLsim [5] is a similar proposal at the operating system level, that virtualizes Linux while providing a simulated timeline and network. By requiring a full Linux installation for each node, it restricts attainable scale even more that CESIUM.

The ideal approach would thus have the ability to run unmodified implementation code, such as possible by using cloud computing, with the frugality and scalability of simulation, the ability to reproduce timeliness properties of CESIUM, and the ability to capture distributed snapshots of distributed debuggers. This the challenge addressed in this paper with Minha.

3 Minha Framework

Minha is a practical framework for testing large-scale distributed systems. Minha virtualizes

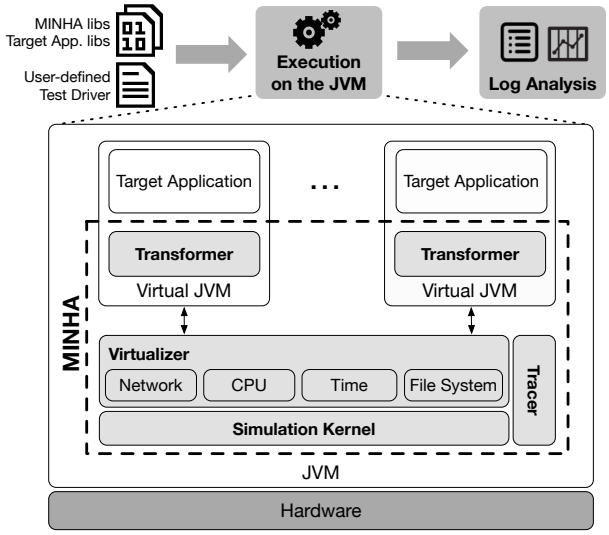

Figure 1 Execution flow and architecture of MINHA.

multiple JVM instances within a single JVM and simulates key environment components, thus allowing reproducing the concurrency, distribution, and performance characteristics of an actual distributed system at a much lower resource cost.

The usage of MINHA is shown at the top of Figure 1. From left to right, a *test driver* defines the execution scenario, such as the number of instances of the target application to be created and the global properties to be checked. The test driver is then executed, along with the target application and MINHA libraries, on an off-the-shelf JVM. Properties can be checked in runtime and logs stored for further off-line checking and visualization.

At runtime, MINHA acts as an interposition layer that takes control of the execution and steers it according to the testing scenario defined in the test driver. As show also in Figure 1, the main components of MINHA are: the *simulation kernel*, the *virtualizer*, the *transformer*, and the *tracer*. Briefly, the transformer converts application and middleware code into an event-driven simulation that interacts with models provided by the virtualizer. Both run on an event-driven simulation kernel. The tracer collects information for off-line use. The remainder of this section describes how the design and implementation of MINHA address the twin challenge of scale and observability of distributed systems software.

3.1 Achieving scale

Simulating multiple processes in the JVM requires isolating their code and preventing their executions from interfering with each other. Prior work typically addresses this issue by using a separate Java class loader for each process [6], that provides private copies of code and data for each virtual process. However, this approach severely limits attainable numerical scale, as each virtual process has to load, transform, compile, and store its own copy of each class.

A second aspect of scale is the size and complexity of the application and middleware that can be loaded. Current systems make use of large portions of the Java platform API in addition to the basic networking and time interfaces that have been intercepted in previous proposals [6]. Simply replacing all Java API with simulation models would lead to a very large development effort while impairing compatibility.

MINHA's transformer addresses these challenges by using single class loader for all virtual

processes and converts the original platform libraries to use simulation models of external resources, as happens for user provided middleware. As process isolation still needs to be enforced and the JVM forbids class loaders from rewriting native classes, MINHA uses the ASM Java bytecode manipulation and analysis framework [9] to perform the following bytecode modifications:

Redirecting references to virtualized classes. Direct references to classes that are replaced by simulation models (e.g., `java.net.Socket`) are rewritten. Simulation models need then to be written, with same same interfaces, and containing simulation logic to reproduce their behaviors. This can however be done incrementally: As new applications demand new platform interfaces that haven't yet been modeled, they fail and report the problem. Experience shows that having implemented models for a moderate subset of the platform API supports many interesting distributed middleware components.

Redirecting static instance variables. This transformation moves static instance variables to regular instance variables in an auxiliary class. It then creates and uses static setter and getter methods for each of them. These methods use a map in the simulation model of each process to store and retrieve the correct instance, thus enforcing isolation.

Redirecting references to renamed classes and methods. Transformed platform classes are renamed to a separate package, thus circumventing the restriction to modifying them. Therefore, references to these classes and to classes that are replaced with simulated versions are re-directed to the new package by transforming their respective callers. Individual static methods in platform classes that do not need to be replaced as a whole (e.g., `System.currentTimeMillis()`) are simply redirected to simulated versions.

Moreover, a subset of classes, containing the simulation kernel and environment models, are kept *global* by delegating their load to the system's class loader. In contrast to isolation provided by using multiple class loaders and the Java security manager, the isolation between virtualized nodes in MINHA is not designed for containing any malicious code or attack. This is however not needed, as MINHA is used only for testing and all nodes are inherently trusted. This has the advantage of providing a controlled channel for virtual JVMs to interact with each other, that can easily be exploited by the user-defined test driver.

Finally, the last challenge to scale is in the discrete event simulation kernel. It keeps a list of future events per simulated timeline, scheduled to execute at target simulation instants. To scale up to large simulations, MINHA's simulation kernel supports multiple timelines for parallel execution in multi-core systems. Timelines in a single simulation are conservatively synchronized with a time window [28]: current time for events executing in parallel in different timelines differs by at most a constant W. This exploits the fact that MINHA maps one or more simulated hosts to each timeline and that the network latency is at least W. Therefore, message delivery events scheduled on a different timeline are guaranteed to be properly ordered.

Parallel simulation in MINHA is optimized to use one timeline for each available processor core. This is achieved by using a concurrent non-blocking data structure to store the event list and a non-blocking algorithm to keep track of the synchronization window. In detail, each thread tries to update the global lowest time of an event executing across all timelines, spinning until its next event handler can be safely executed. This approach works well in cases where events are evenly spread across all timelines, as typically happens when simulating large distributed systems in a few processor cores.

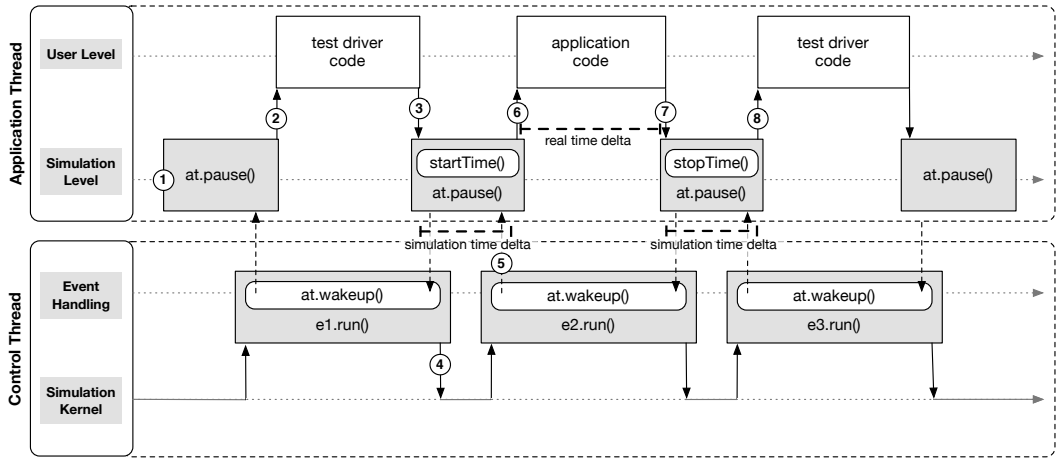

Figure 2 Example of how simulation time and execution time are coordinated in MINHA.

3.2 Achieving observability

MINHA's programming interface is targeted at automated tests and scripting and combines reflection, to represent entities in the simulation domain, with an interface to inject events and interact with those entities.

To set up and control the simulation, the test driver API provides the following entities that describe and manipulate the simulated system: `World` represents the system as a whole, keeping track of global simulated time and allowing hosts to be created and enumerated. A `Host` describes a simulated host with processing and storage resources and attached to the network with an address. It allows creating and enumerating simulated processes. A `Process` keeps a private address space with its own copy of `static` variables and allows invocation of methods. Both hosts and processes can be terminated, to simulate crash faults.

Execution in the context of a simulated process is started by creating an `Entry` proxy for some interface and scheduling an invocation. Each entry point corresponds to a thread in the simulated process. Using the interface, it can specify arbitrary delays or an absolute schedule and if it is executed synchronously, implicitly running the simulation until the invocation returns, or asynchronously, providing a future to wait for and retrieve the result whenever the simulation has advanced enough. The simulation can also be run for predetermined periods of time, easing periodic observation. The API also provides the ability to exit the simulation to execute code that performs global observation or logging. This is achieved with an `Exit` proxy that ensures that the simulation is stopped on invocation and restarted on return. An example using the API is shown in Figure 3 and further discussed in Section 4.

The main challenge addressed is ensuring that only consistent global states can be inspected, regardless of the simulation containing multiple threads in a number of virtual processes. First, all blocking operations, such as synchronization and calls to the platform, are replaced with calls to simulated synchronization primitives by modifying the compiled bytecode at load time. Second, the transformed code executes consistently with simulation time. This ensures that: *i)* the application thread advances only in the context of an simulation event; *ii)* the execution time observed is reflected in the usage of a simulated CPU core; and *iii)* the waiting time (e.g., when reading from disk) is computed by the simulation and not by actual contention.

Figure 2 shows in detail how this is achieved. From the bottom to the top, the Control

Thread (CT) originates in the Simulation Kernel and executes discrete Event Handling procedures. In contrast to common discrete event simulation practice, these do not directly modify model variables. Instead, they allow an Application Thread (AT) that executes test driver and application code at User Level to advance. An AT thus progresses as follows: (1) When started, the AT stopped using the `pause()` method to wait for its turn; when the corresponding event is scheduled by the Simulation Kernel, the CT signals the AT, that can then execute test driver code (2). When the test driver is done, (3) it starts accounting real time with `startTime()`. This pauses the AT and returns control to the Simulation Kernel, to wait for its turn (4). When simulation time has advanced, the AT is finally signaled to start executing application code (6). The time elapsed while executing application code is measured with the CPU cycle counter in `stopTime()` (7) and used to advance simulation time accordingly. This means that further events, either in application or test driver code (8) will be scheduled appropriately in simulation time.

This allows the passing of real time while executing code to influence simulation time, thus reproducing the performance characteristics as needed to disclose scaling bugs. However, since it is an event driven simulation, global state is consistent and can be observed by direct inspection while the simulation is stopped. Moreover, as logging is done in user-defined test driver code, outside the periods that measure real time, it has no impact in measured performance and does not disturb the system under test.

Note that the test driver thread could get blocked in synchronization primitives within the target application and middleware, leading to a deadlock. However, as synchronization primitives have been replaced by simulated counterparts, these will recognize that the invoking thread is the test driver and avoid blocking it. The developer writing the test driver has however to be careful to make sure that the code used for observation is not susceptible to inconsistent internal state.

In addition to directly checking properties, the tracer component allows logging events of interest at runtime for off-line processing. Currently, Minha is configured to trace events regarding: thread synchronization (i.e. *fork/join/start/end* events), inter-node communication (i.e. socket *send/receive* events), and *read/write* accesses to variables indicated by the programmer, if any. Recall that capturing memory accesses requires instructing the transformer to dynamically instrument the target application's bytecode with calls to the tracer, which may incur additional overhead during the execution. In contrast, all the remaining events are captured by Minha at the simulator side (i.e., outside the application), hence they do not impose any slowdown, as opposed to what happens in a real deployment.

An additional feature of the traces produced by Minha is that events are logged in a coherent global order, due to the framework's centralized and virtualized nature. This is particularly useful for debugging. In fact, Minha comes with a built-in diagram generator that provides a visual representation of the execution according to the information stored in the trace file. Section 4.2 illustrates the benefits of this feature using the Cyclon example of Section 4.

The event trace is produced in JSON format, which can later be consumed by external tools. Events are stored as JSON objects, containing fields regarding: the thread identifier, the type of event (as indicated in tracer's description in Section 3), the timestamp, and, for inter-node communication events, a message unique identifier, and the source and destination node addresses. Minha's built-in visualizer consists of a Javascript module that uses the `SVG.js` library to generate a graphical representation of the event trace as a space-time diagram [20]. An example is shown in Figure 4 and discussed in the next section.

Algorithm 1 – Cyclon Active Thread at Node p

```
Init:
V_size ← size of p's partial view
view ← p's view containing V_size references to other peers
S_size ← number of peers exchanged during a shuffling
         operation (S_size ≤ V_size)
for every T time units do
    // increase by one the age of all neighbors
    ageGlobal(view)
    // pick node q with highest age among all neighbors
    q = getOldestPeer(view)
    // select a random subset of S_size neighbors
    peers = selectPeers(view, S_size)
    // send list of peers to q, indicating that this message is a request for a shuffle
    send(q, {REQ, p, peers})
end for
```

4 Use Case: Peer Sampling Service

In this section, we show how Minha can be used to test the properties of the overlays generated by the peer sampling service (PSS) Cyclon [37, 29]. A PSS is a mechanism, widely used by gossip-based peer-to-peer systems, that provides each node of the system with a partial *view* of the network. The overlay network used by the gossip protocol to spread information is thus defined by the logical graph that emerges from the union of all nodes' views at a given instant. Since the effectiveness and efficiency of the dissemination depends heavily on the properties of the overlay, the PSS refreshes each node's view from time to time to account for peer joins and leaves.

4.1 Test application

For the purpose of this example, we will focus on a particular PSS named Cyclon [37]. In a nutshell, the Cyclon protocol consists in a series of *shuffle* rounds, where pairs of neighbor nodes exchange a subset of randomly sampled peers from their views. The shuffle procedure works as follows. Each node assigns an *age* value to the node references in its view. This value is incremented by one at the beginning of a new exchange round, every T units of time. Upon starting a new shuffle, nodes pick the neighbor with highest age and send to it a random subset of other peers in their view, replacing the oldest node's reference with a self-reference of age 0. In turn, when a shuffle message is received, nodes first reply with a subset of peers randomly selected from their own view, and then update their views with the references received, replacing the entries previously sent. As a result of shuffle operations, nodes alive have their views refreshed and nodes that left the system are eventually removed from every view. This way, Cyclon is able to cope with dynamism.

The experimental results in Cyclon's original paper [37] shown that this protocol is able to generate network overlays with properties similar to random graphs, even starting from non-random topologies. This ability is particularly relevant for peer-to-peer systems that have to handle high churn, as random overlays exhibit low diameter and are able to maintain connectivity even in the presence of massive node failures.

The results presented in the original paper were obtained solely from simulations though, so it remains unclear how would the protocol behave in an actual distributed system. In such a real-world setting, the Cyclon protocol can be implemented by means of two execution threads: an *active thread* that is responsible for initiating a new shuffle with a neighbor, and a *passive thread* that operates as a message handler, which receives and processes the

Algorithm 2 – Cyclon Passive Thread at Node p

while *true* **do**
  // receive a list of peers sent by node q
  $\{t, q, peersRcv\} = \text{RECEIVE}()$
  if $t = REQ$ **then**
    // select a random subset of S_{size} neighbors
    $peersSnd = \text{SELECTPEERS}(view, S_{size})$
    // send list of peers to q, indicating that this message is a reply to the shuffle
    $\text{SEND}(q, \{REP, p, peersSnd\})$
  end if
  // incorporate list of peers received into its own view
  $\text{UPDATEVIEW}(view, peersRcv)$
end while

```
1  //create a new simulation with 500 Cyclon peers
2  World world = new Simulation();
3  Entry<ICyclon>[] peers = world.createEntries(500, ICyclon.class, CyclonImpl.class.
       getName());
4
5  //start Cyclon peers
6  for (int i = 0; i < 500; i++){
7      peers[i].queue().run();
8  }
9
10 //allow nodes to bootstrap
11 world.run(1,TimeUnit.SECONDS);
12
13 //let the application code execute for 5s
14 //and then observe the overlay
15 for(int i=0; i <= 100; i++) {
16    world.run(5 , TimeUnit.SECONDS);
17    for(ICyclon c : peers){
18        //inspect node's view and store it into log
19        logView(c.getView());
20    }
21 }
```

Figure 3 Test driver for running Cyclon on Minha

messages sent by other nodes. The portions of Cyclon executed by the two threads are detailed in Algorithms 1 and 2, respectively.

4.2 Test driver

To conduct the experiment in Minha, one can use a test driver like the one presented in Figure 3. The test driver starts by creating a new simulation scenario with 500 hosts, each corresponding to a Cyclon node, using Minha's API (lines 2-3). Nodes are then prepared for execution by enqueuing the method `run()` in the simulator (lines 6-8). This method is responsible for spawning Cyclon's passive and active threads, as well as initiating the node's view with references to its 11 subsequent neighbors.

The test driver proceeds with an instruction to let the simulation run for one second, thus allowing the Cyclon nodes to bootstrap (line 11). The core of the test run consists of 100 simulation cycles, in which Minha lets the application code of each instance execute for 5 seconds, before giving back the control to the test driver code (lines 15-21). At this point, the test driver consults the local state of each node, namely the composition of its view, and logs it into a trace file (lines 17-20). We highlight that these observations are performed

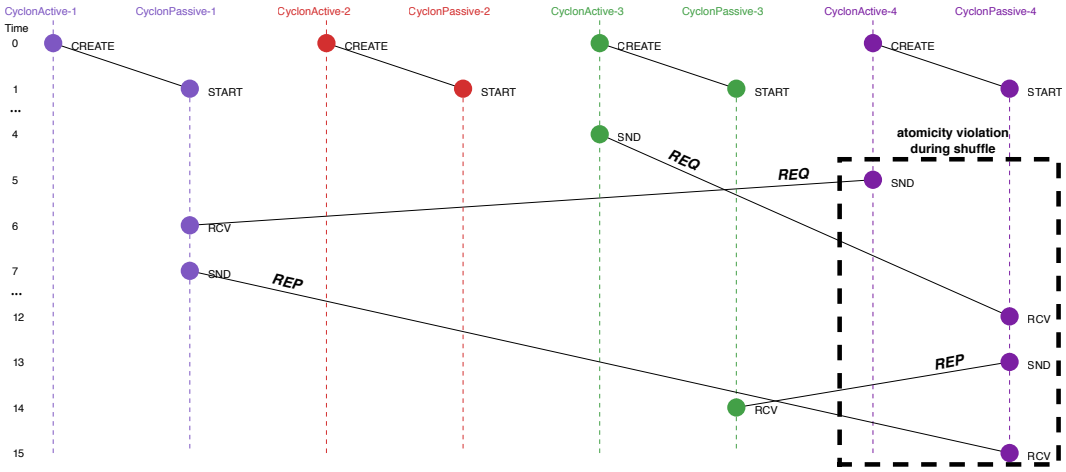

Figure 4 Space-time diagram generated by MINHA's visualizer from the event trace captured during the simulation of Cyclon. The values on the left represent logical time ticks, whereas events *CREATE*, *START*, *SND*, *RCV*, denote, respectively, the creation and beginning of a thread, and the send and receive of a message. Messages labeled as *REQ* indicate shuffle requests and as *REP* indicate shuffle replies. The dashed box highlights the fact that atomicity is not guaranteed during a view exchange procedure, as new shuffle requests may arrive in the meantime.

transparently to the application and without incurring runtime overhead. Since all nodes are *paused*, the local views are obtained from a global snapshot of the system, thus allowing reenacting the actual overlay network at that exact moment.

4.3 Log analysis

Off-line log analysis is particularly well suited to discover message race conditions. As an example, we search logs obtained with MINHA for cases in which a node is involved in more that one concurrent shuffle operations. This fact is not contemplated in the original Cyclon algorithm, nor in the simulations presented in the original paper [37].

On the other hand, these atomicity violations can easily occur in real settings, and may hamper the properties of the overlay generated, especially in highly dynamic environments. Figure 4 illustrates a detail of a time-space diagram of the execution plotted from the event trace captured during the simulation for our experiment with Cyclon. To improve readability, we only depict the timelines of the two Cyclon threads (namely, the active thread and the passive thread) for the first four nodes. From the diagram, it can be observed that the active thread starts by spawning the passive thread and then proceeds to sending shuffle requests. In turn, the passive thread is responsible for receiving incoming messages and reply back to complete the view exchange, as defined by the protocol.

This time-space diagram confirms the interesting scenario: The atomicity of a shuffle operation is not guaranteed in practice (see the dashed box in Figure 4). In fact, a node can receive a new shuffle request from a third node in-between swapping views with a neighbor. Since the view updates performed by the passive thread upon receiving shuffle requests and replies are not commutative, it results in a view that is not anticipated in the algorithm. A possible solution to address this issue is to store incoming requests in a queue until the awaited reply arrives [17].

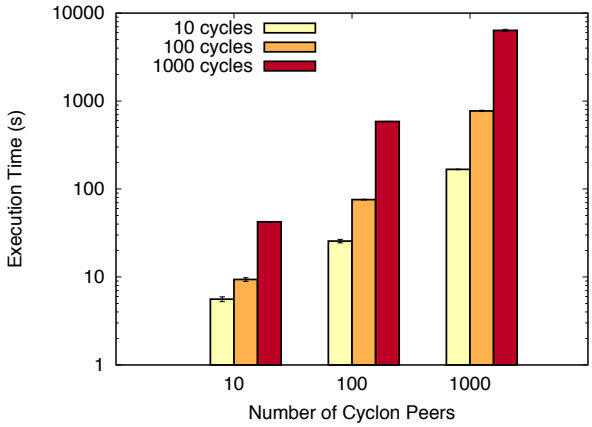

Figure 5 Minha's execution time (in log scale) for simulations considering different configurations of Cyclon.

4.4 Scalability

This section evaluates how Minha's performance scales with the number of nodes simulated. In particular, we measured the execution time of Minha when varying both the number of nodes and the number of cycles considered in the simulation within the range $\{10, 100, 1000\}$. The experiments were performed on a machine with a 3.4 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk. The results, averaged for three runs, are depicted in Figure 5.

As expected, the figure shows that Minha's execution time grows proportionally to the size of the simulation both in terms of number of Cyclon peers and number of cycles, making it efficient and practical for in-house testing of large-scale distributed systems as a simulation of the Cyclon protocol with 1000 nodes and 100 cycles (which suffice to assess most properties of the overlay, as shown in Section 4.2) takes only around 13 minutes with a minimal hardware configuration.

5 Use Case: Data Store

In this section, we provide a second example of Minha using the DataFlasks peer-to-peer database system.[2] This is used to show that Minha copes with an unmodified larger application that makes use of more features of the Java platform and external libraries. Moreover, we compare the scalability and cost of tests with the common alternatives of setting up a real distributed system or using virtual machines.

5.1 Global Property Checking

The DataFlasks distributed data store was designed to ensure availability of data in the presence of varying levels of node churn [30]. To assess this property, we ran DataFlasks on Minha with different levels of churn and wrote a test driver to compute the number of data replicas stored in the system at a given instant Churn is easily injected in using

[2] We used the version of DataFlasks publicly available at `github.com/fmaia/dataflasks`

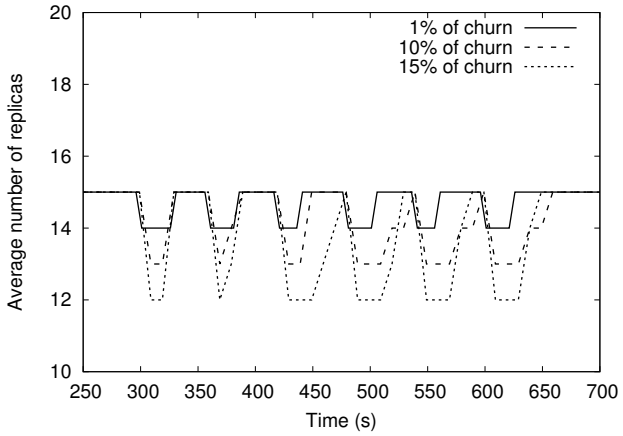

Figure 6 MINHA's dynamic property checking applied to the evaluation of DATAFLASKS' replica maintenance properties.

the test driver API by invoking `close()` on `Host` objects, to remove them from the system, or by creating new `Host` instances to bring them back. Computing the number of keys in the system requires inspecting the storage of each node to check the data it is holding and combining such information with that of the other nodes.

In the experiments, we instructed MINHA to run the verification code and compute the average number of replicas that the system holds at execution intervals of 1 second. We plot the results of the experiments in Figure 6 for a simulation of 700 seconds. The results show that, as expected, the average number of data replicas varies with churn (the higher the churn, the fewer replicas exist in the system), although DATAFLASKS is always able to eventually recover the keys lost. In fact, the system ends the experiment maintaining the expected mean number of data replicas. Checking the same property in a real-world deployment would require extensive logging and cumbersome synchronization mechanisms.

5.2 Performance and Resource Usage

For this experiment, we considered three different three different deployment configurations for DATAFLASKS:

Configuration 1 (Commodity). We considered a deployment built from a set of several commodity hosts. Each commodity host is randomly selected at startup time from a pool of resources equipped with either *i)* a 3.1 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk, *ii)* a 3.4 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a 7200 RPMs SATA disk, or *iii)* a 3.7 GHz Dual-Core Intel i3 Processor, 8 GB of RAM and a SSD disk. All hosts are interconnected through a switched Gigabit Ethernet. The total number of commodity hosts available at deployment time was limited to 36, which aims to represent a scenario where the available resources for testing are not comparable with the equivalent to those expected in a production deployment.

Configuration 2 (Virtual Machines). We considered a deployment built from a single server grade machine equipped with an 2.3 GHz Intel Xeon E5-2670 v3 Processor, 94GB of RAM and a 7200 RPMs SATA disk. This machine was configured to deploy a set of 32 virtual machines interconnected through a virtualized switched Ethernet.

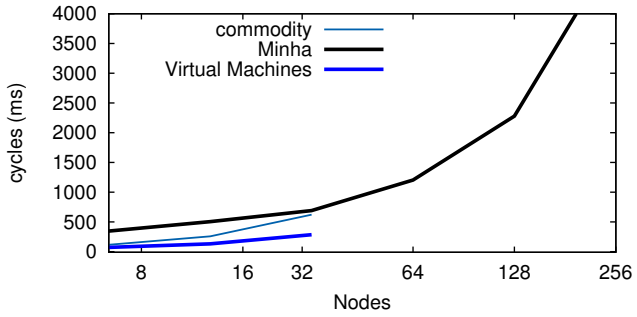

Figure 7 Time to execute the workload in the DataFlasks key-value store for the three deployment configurations.

Configuration 3 (Minha). We considered a single commodity host, extracted from the pool of resources described in the first scenario, on which we deployed Minha.

For each one of the previous configurations, we deployed DataFlasks with an increasing number of nodes within the range $\{8, 16, 32, 64, 128, 256\}$. We then performed the experiments by running a simple write workload on top of DataFlasks using YCSB [10], in a total of 5 independent runs for each configuration. For each experiment, we measured the time required to replicate data across all the necessary replicas and used it to compare the different configurations. Figure 7 depicts the experimental results.

Figure 7 shows that besides scaling to larger system sizes than a configuration with multiple virtual machines, Minha is able to do it on a single commodity host that represents 1/36 of the cost of a real deployment depicted in either the first or second configurations. These results support the claim that Minha is able to accurately simulate large-scale distributed systems with much less resources than traditional approaches.

6 Conclusions and Future Work

Distributed systems are notoriously hard to test, mainly due to their large state space and the difficulty in deploying large infrastructures. This paper addresses the challenges of scale and observability in Minha, a simulation framework aimed at easing the burden of testing large-scale distributed systems. Minha virtualizes multiple JVM instances within a single JVM, while simulating key environment components, thus reproducing the concurrency, distribution, and performance characteristics of an actual system. Minha also helps assessing the application's correctness by allowing checking distributed properties on globally-consistent snapshots of the system. Moreover, due to time virtualization, these system-wide assertions can be performed transparently to the target application and without affecting its runtime performance.

Our experiments with a large-scale key-value store and a peer sampling service show that Minha can accurately reproduce the characteristics of real-world deployments with fewer resources than traditional approaches, and is effective in assessing system-wide properties.

We believe that Minha opens a number of interesting research opportunities in the field of distributed systems testing and debugging. For example, Minha can be extended with model checking capabilities to systematically explore the execution space of distributed systems and discover new bugs. Furthermore, Minha's event traces, which are totally ordered and logged without any runtime overhead for the target application, can be combined with bug

detection techniques to also detect problems automatically.

References

1. Apache Cassandra. `http://cassandra.apache.org`.
2. Apache Hadoop. `http://hadoop.apache.org`.
3. Bug CASSANDRA-6127: vnodes don't scale to hundreds of nodes. `https://issues.apache.org/jira/browse/CASSANDRA-6127`.
4. Bug ZOOKEEPER-2212: distributed race condition related to qv version. `https://issues.apache.org/jira/browse/ZOOKEEPER-2212`.
5. Werner Almesberger. umlsim-a uml-based simulator. In *10th International Linux System Technology Conference (Linux-Kongress 2003)*, pages 202–213, 2003.
6. G. A. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *SRDS '97*, pages 35–42, Oct 1997.
7. Rimon Barr, Zygmunt J Haas, and Robbert van Renesse. Jist: An efficient approach to simulation using virtual machines. *Software: Practice and Experience*, 35(6):539–576, 2005.
8. Y Bertot and P Castéran. Interactive theorem proving and program development—coq'art: The calculus of inductive constructions (2004).
9. Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.
10. Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC '10*, pages 143–154, New York, NY, USA, 2010. ACM.
11. Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *EuroSys '17*, New York, NY, USA, 2017. ACM.
12. F. Gortázar, M. Gallego, M. Donato, E. Pages, A. Edmonds, G. Tuñón, A. Bertolino, G. De Angelis, A. Cervantes, T. Bohnert, A. Willner, and V. Gowtham. The ElasTest platform: Supporting automation of end-to-end testing of large complex applications. ElasTest project whitepaper, November 2018. URL: `https://elastest.io/resources/ElasTest_white_paper.pdf`.
13. Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *SOSP '11*, pages 265–278. ACM, 2011.
14. Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.
15. Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The phi accrual failure detector. In *SRDS*. IEEE Computer Society, 2004.
16. Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the emulab network testbed.
17. Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.
18. Charles Killian, James W Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI '07*, 2007.
19. Ignacio Laguna, Dong H. Ahn, Bronis R. de Supinski, Todd Gamblin, Gregory L. Lee, Martin Schulz, Saurabh Bagchi, Milind Kulkarni, Bowen Zhou, Zhezhe Chen, and Feng Qin. Debugging high-performance computing applications at massive scales. *Commun. ACM*, 58(9):72–81, August 2015.
20. Leslie Lamport. Time clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.
21. Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

22 Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *OSDI '14*. USENIX Association, 2014.

23 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *ASPLOS '16*. ACM, 2016.

24 Tanakorn Leesatapornwongsa, Cesar A. Stuardo, Riza O. Suminto, Huan Ke, Jeffrey F. Lukman, and Haryadi S. Gunawi. Scalability bugs: When 100-node testing is not enough. In *HotOS '17*, pages 24–29. ACM, 2017.

25 Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI*, volume 9, pages 185–198, 2009.

26 Haopeng Liu, Guangpu Li, Jeffrey F. Lukman, Jiaxin Li, Shan Lu, Haryadi S. Gunawi, and Chen Tian. DCatch: Automatically detecting distributed concurrency bugs in cloud systems. In *ASPLOS '17*. ACM, 2017.

27 Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D3s: Debugging deployed distributed systems. In *NSDI '08*. USENIX Association, 2008.

28 B. D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Commun. ACM*, 32(1):111–123, January 1989.

29 Nuno Machado, Francisco Maia, Miguel Matos, and Rui Oliveira. BuzzPSS: A dependable and adaptive peer sampling service. In *LADC '16*. IEEE Computer Society, 2016.

30 Francisco Maia, Miguel Matos, Ricardo Vilaça, José Pereira, Rui Oliveira, and Etienne Rivière. Dataflasks: Epidemic store for massive scale systems. In *SRDS '14*. IEEE Computer Society, 2014.

31 A. Montresor and M. Jelasity. Peersim: A scalable p2p simulator. In *International Conference on Peer-to-Peer Computing*, 2009.

32 Larry Peterson and Timothy Roscoe. The design principles of planetlab. *ACM SIGOPS operating systems review*, 40(1):11–16, 2006.

33 Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware '01*, London, UK, UK. Springer-Verlag.

34 Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: Observing, analyzing, and reducing variance. *Proc. VLDB Endow.*, 3(1-2):460–471, September 2010. URL: `http://dx.doi.org/10.14778/1920841.1920902`, `doi:10.14778/1920841.1920902`.

35 Jiri Simsa, Randy Bryant, and Garth A Gibson. dbug: Systematic evaluation of distributed systems. In *SSV '10*, 2010.

36 Peter Urban, Xavier Défago, and André Schiper. Neko: A single environment to simulate and prototype distributed algorithms. In *Information Networking, 2001. Proceedings. 15th International Conference on*, pages 503–511. IEEE, 2001.

37 Spyros Voulgaris, Daniela Gavidia, and Maarten van Steen. CYCLON: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

38 Yang Wang, Manos Kapritsos, Lara Schmidt, Lorenzo Alvisi, and Mike Dahlin. Exalt: Empowering researchers to evaluate large-scale storage systems. In *NSDI'14*, pages 129–141, Berkeley, CA, USA, 2014. USENIX Association.

39 Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent model checking of unmodified distributed systems. In *NSDI '09*. USENIX Association, 2009.