Practical Evaluation of Large Scale Applications

Tiago Jorge, Francisco Maia, Miguel Matos, José Pereira, Rui Oliveira

▶ **To cite this version:**

Tiago Jorge, Francisco Maia, Miguel Matos, José Pereira, Rui Oliveira. Practical Evaluation of Large Scale Applications. 15th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2015, Grenoble, France. pp.124-137, 10.1007/978-3-319-19129-4_10 . hal-01775035

HAL Id: hal-01775035

https://inria.hal.science/hal-01775035v1

Submitted on 24 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Practical Evaluation of Large Scale Applications

Tiago Jorge, Francisco Maia, Miguel Matos, José Pereira, and Rui Oliveira

INESC TEC & U. Minho
{tiago.jorge,francisco.maia,miguelmatos,jop,rco}@di.uminho.pt
http://www.haslab.pt

Abstract. Designing and implementing distributed systems is a hard endeavor, both at an abstract level when designing the system, and at a concrete level when implementing, debugging and evaluating it. This stems not only from the inherent complexity of writing and reasoning about distributed software, but also from the lack of tools for testing and evaluating it under realistic conditions. Moreover, the gap between the protocols' specifications found on research papers and their implementations on real code is huge, leading to inconsistencies that often result in the implementation no longer following the specification. As an example, the specification of the popular Chord DHT comprises a few dozens of lines, while its Java implementation, OpenChord, is close to twenty thousand lines, excluding libraries. This makes it hard and error prone to change the implementation to reflect changes in the specification, regardless of programmers' skill. Besides, critical behavior due to the unpredictable interleaving of operations and network uncertainty, can only be observed on a realistic setting, limiting the usefulness of simulation tools. We believe that being able to write an algorithm implementation very close to its specification, and evaluating it in a real environment is a big step in the direction of building better distributed systems. Our approach leverages the MINHA platform to offer a set of built in primitives that allows one to program very close to pseudo-code. This high level implementation can interact with off-the-shelf existing middleware and can be gradually replaced by a production-ready Java implementation. In this paper, we present the system design and showcase it using a well-known algorithm from the literature.

Keywords: testing and evaluation; distributed systems, simulation and emulation

1 Introduction

Real distributed systems are often built around several collaborating middleware components such as a membership or coordination service. The correctness and performance of these systems depends not only on the particular algorithm used to solve the problem, but also on the interactions among the supporting middleware components. Despite its importance and criticality, experimentally assessing such distributed systems in a large scale setting is a daunting task.

Unfortunately, interesting behavior - and bugs - often arise exclusively in large scale settings where intra-component concurrency, the interleaving among components' operations and network uncertainty, expose the system to previously overlooked issues. This is aggravated by the fact that the very conditions that cause the problems to appear in the first place are often hard to determine, and harder to reproduce. Simulators such as ns-2 [12] or PeerSim [11] partially address this problem, but their usefulness is limited to validating the design and specification, not production code. This requires maintaining the simulation and real implementations in tandem, which due to the huge gap in complexity between them, becomes error-prone and time consuming as we have witnessed first-hand several times [10]. Test beds such as PlanetLab [3] or a cloud infrastructure allow to perform very large scale deployments, but system observability and reproducibility of testing conditions on normal and faulty conditions poses several challenges. As a matter of fact, not only a coherently global observation is physically impossible, but also the system's behavior remains largely unpredictable and unreproducible. Other tools allow to run real code but are limited to a particular framework and language [8], or are limited in scope [2, 14, 5], thus precluding the integration with required off-the-shelf middleware components and providing only rough estimates of system behavior and performance.

In this paper, we take a different approach to the problem. Instead of building yet another simulator, we rely on the MINHA [1] platform [4] and extend it with capabilities to write distributed algorithms at a high abstraction level. Briefly, MINHA virtualizes multiple Java Virtual Machines (JVM) instances in a single JVM while simulating key environment components, reproducing the concurrency, distribution, and performance characteristics of a much larger distributed system. By virtualizing time, it is possible to get a global observation of all operation and system variables, while simulation models make it possible to reproduce specific testing conditions. This allows to run and evaluate unmodified Java code in real, yet reproducible conditions.

In this work we extend MINHA to provide a simplified API that features common distributed systems functionality allowing to write algorithms very concisely. Despite this, such code still runs as real code and can interact with off-the-shelf middleware components. This allows one to not only develop and test algorithms faster but also to incrementally replace the concise implementations, close to pseudo-code, with fully fledged Java implementations for critical parts of the system. Since MINHA accurately reflects the cost of executing real code, succedaneum components can be used with the guarantee that its behavior can be analyzed, controlled and evaluated under precise reproducible conditions, thus minimizing measurements fluctuations. Arguably, working closer to the specification allows better reasoning about the problem at hand and therefore easier detection and correction of problems. This is achieved through the JSR-223 Java Scripting API [7] which allows to run scripting languages inside the JVM. We consider the Python programming language in particular, but our approach lends itself to be used in any language supported by JSR-223.

[1] `www.minha.pt`

The rest of this paper is organized as follows. We begin by providing background on important characteristics of MINHA and JSR-223 in Section 2. Then we describe our system in Section 3 and present a concrete use case by providing the implementation of the Chord DHT in Section 4. Related work is debated in Section 5 and, finally, Section 6 concludes the paper and discusses future work.

2 Background

In this section we provide some background and context for our framework. We describe two frameworks on which our own work relies, MINHA and JSR-223.

The MINHA [4] framework is capable of virtualizing multiple JVM instances in a single JVM. It is able to do so simulating a real distributed environment by virtualizing the network, CPU scheduling and by virtualizing most of the standard Java APIs. As a consequence, it is possible to run multiple instances of any Java application in a single machine. Each instance believes it is running in its own machine and runs without the need to adapt any of its code. By running multiple instances of an application in a single JVM, MINHA reduces significantly the resources typically required for evaluating it in a large scale scenario. This makes large scale evaluation practical.

One critical advantage of MINHA is the fact it virtualizes time. Once time is virtualized, it is possible to perform a global system observation at any moment of the simulation. Moreover, contrary to execution in a real environment, there is no overhead introduced by observation and control or even by debugging, so execution time can be considered for analysis. Global observation of system state and each application instance variables greatly eases the process of detecting and solving problems the application may exhibit.

Another important aspect of the MINHA platform is that environments and software models can be replaced by simulation models, and incorporated in a standard test harness to be run automatically as code evolves. By resorting to simulated components and running the system with varying parameters, the impact of extreme environments can be assessed and reproducing testing conditions becomes automatic. Thanks to this holistic approach, when a component does not yield the expected results, the developer can quickly identify and fix any problem that may exist, and reevaluate the new component version for the same exact conditions. The ability to easily replace software components or mock them allows for iterative development and allows for component-targeted evaluation and validation, which greatly eases the development of reliable software.

Additionally, MINHA not only allows to run real applications, but it also virtualizes a significant part of a modern Java platform, thus providing unprecedented support for running existing code. In particular, the virtualization of threading and concurrency control primitives provides additional detail when simulating concurrent code, as is usually the case of middleware components. Code is run unmodified and time is accounted using the CPU time-stamp counter to closely obtain true performance characteristics.

```
World world = new World();
Entry<Main>[] e = world.createEntries(10);
for(int i=0; i<e.length; i++)
    e.queue().main("test.Main", "arg0", "arg1");
world.runAll(e);
world.close();
```

Listing 1.1. Asynchronous invocation of 10 identical MINHA entries.

MINHA's API allows the invocation of arbitrary methods, several scheduling options, asynchronous invocations, and callbacks. As presented in Listing 1.1, creating and invoking a number of identical application instances is quite straight forward. Basically, each `entry` object will represent an application instance running in its own host to which it is passed the application to run as a parameter. In this case the application is `test.Main` and ten instances of this application are run. These and other utilities that allow to create and control entries for user defined interfaces, make it very simple to simulate large scale applications without having to modify its code.

The JSR-223 Java Scripting API [7] is a framework that allows developers to run scripting language code in the JVM. Any JSR-223 compliant scripting language can be used. This way, it becomes possible to write Java applications that can be easily customizable and extendable in a scripting language of choice. Possible languages include Python (via Jython), JavaScript (via Rhino) or Lua (via LuaJ). This flexibility is also very useful for reusing code from existing protocols implemented in a scripting language. Scripting languages are convenient because they are easy to learn and use, allow complex tasks to be performed in relatively few steps, thus requiring less lines of code, and code testing can be made on the fly with handy interpreters. Mainly, their conciseness allow one to prototype ideas quickly, focusing on early proof of concept implementations close to pseudo-code.

Importantly, access between the scripting language and regular Java classes is bidirectional, meaning that the scripting language has access to regular Java and vice-versa. Therefore, algorithms programmed in our simplified API can still access other middleware components such as a group communication toolkit. As shown in Listing 1.2, one can expose an object (`File f`) as a variable to the script, that, as such, can access it and call methods on it. On the other hand, it is also possible to define a script object (`var o = new Object()`), expose it to the Java class, and invoke methods on it through the `Invocable` interface.

The flexibility in choosing any JSR-223 compliant scripting language, together with its bidirectional exposure, are determining factors for the integration of components written in different languages.

```
public class Bidirect {
 public static void main(String[] args) throws Exception {
  ScriptEngineManager manager = new ScriptEngineManager();
  ScriptEngine engine = manager.getEngineByName("JavaScript");
  // Script to Java
  File f = new File("test.txt");
  engine.put("file", f);
  engine.eval("print(file.getAbsolutePath())");
  // Java to Script
  String s = "var o=new Object(); o.hi=function(n){print('Hi
      '+n);}";
  engine.eval(s);
  Invocable inv = (Invocable) engine;
  Object o = engine.get("o");
  inv.invokeMethod(o, "hi", "Script Method" );
 }
}
```

Listing 1.2. Example of Bidirectional access between scripts and Java.

3 Framework Design

As described previously, the MINHA framework allows to run unchanged Java applications in a simulated environment. It allows us to instantiate an arbitrary large number of *peers*, each with its own IP address, as if they were running in their own machines. To each peer it is possible to assign a Java application or service to run. In the same simulation, different peers may run different applications and may interact with each other through the (simulated) network.

Naturally, it is the responsibility of the developer to implement all the communication code for the application. Moreover, as it is real Java code, the developer must deal with all the implementation details of socket management, data marshalling / unmarshalling and message dispatching. This can be a serious drawback when the application is still in early design and prototyping phase. In fact, having developers focusing on these tasks prevents them from dedicating time to the core components of the protocol.

Our framework provides a way of concisely prototyping real code distributed algorithms. These prototypes can be tested and validated in large-scale simulations leveraging MINHA. Integrating the JSR-223 scripting framework, we allow each MINHA peer to run any JSR-223 compliant scripting language code. Additionally, by providing a high level API to the developer, our framework hides lower level intricacies from the developer, such as all the boilerplate code relating to thread and socket management, data marshalling / unmarshalling, and event dispatching.

In Figure 1 we depict the framework architecture and the exposed high level API. The API consists in three system primitives and four types of methods each application must implement. The three primitives are *send*, *call* and *periodic*.

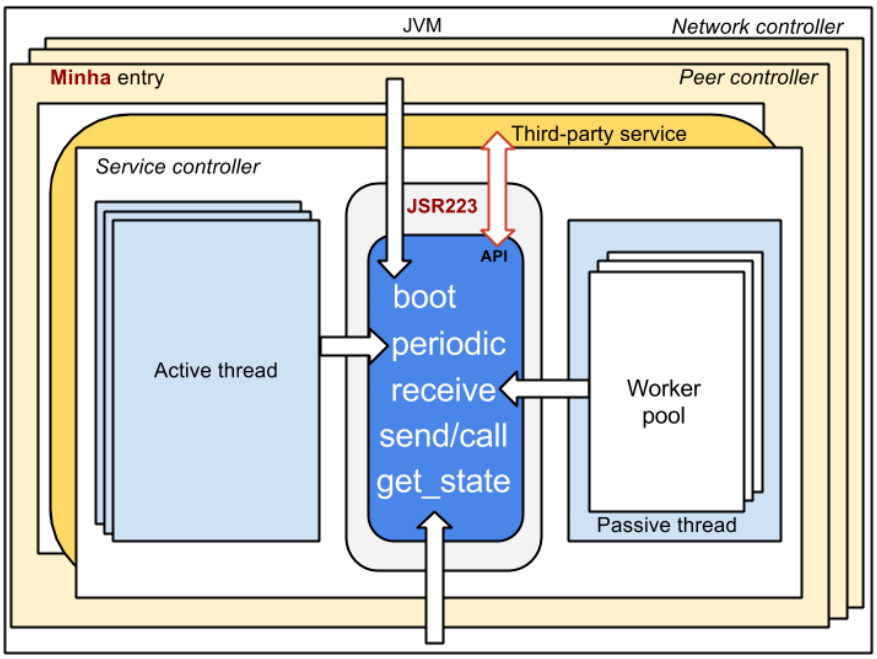

Fig. 1. Framework architecture.

Two of the primitives abstract message dispatch - *send* - and remote procedure call - *call* - without the developer needing to write any communication code. The *periodic* primitive allows the user to register, at boot time, service methods to be invoked periodically. In order to make a runnable application, besides implementing these periodic methods themselves, the developer needs to provide a *boot* method, a *get_state* method and all those events remotely invoked with *send* or *call*, locally triggered through *receive*. Method *boot* is invoked by the platform before actually running the application and should be used to bootstrap the application data structures and any required initializing procedures. The *get_state* method should return a representative state of the peer for system observation purposes. Finally, *receive* is invoked each time a message is delivered to the application and triggers the target event having the necessary logic to process such message. Besides the API, the framework is organized in three main components built on top of MINHA: network controller, peer controller and service controller.

Network controller: In MINHA multiple virtual JVMs can be run in each JVM, significantly reducing the resources required by typical alternatives. This component is responsible for interfacing our framework with MINHA. The system parameters are specified in a YAML configuration file. The number of peers to run and the number of simulation rounds to perform are two of the required

configuration parameters. The network controller creates a MINHA host, which runs in a virtualized JVM, for every peer to run. Alongside this step, each one of this hosts is assigned a unique IP address by the MINHA platform. The component is also responsible for loading both the script engine for each peer and all the user-defined scripts, which implement the framework's API. After instantiating all peers and scheduling their start-up, it initiates the MINHA simulation. Because this component interfaces with the MINHA platform it can perform operations outside the simulation environment. In particular, the network controller is able to globally observe the system. In the current implementation, this is achieved by having all applications implement the *get_state* method. The *get_state* method implementation is application dependent and should return a representative state of the peer. The goal is to be able to globally observe the state of the system by being able to inspect, for the same virtual time, the inner state of every peer in the system. This observation can be done periodically according to user configuration.

Peer controller: This controller corresponds to a MINHA entry, or peer. This component will run the user application, which can actually consist of a stack of what we call services. These services can be smaller applications or protocols that are used as building blocks for a larger application. For instance, in a typical epidemic application [9], different protocols are used as they rely on each other. In our platform, each service is implemented in a JSR-223 compliant language and implementing the exposed API. Alternatively, a service can also be a third-party off-the-shelf application whose specific interface is exposed to the other services leveraging the bidirectional characteristic of the JSR-223 framework. With all the services ready, the user then declares, in the configuration file, which services to run on each peer and, for each service, specific configuration parameters it may require. At runtime, the peer controller loads the list of services to run and their respective configurations, such as protocol specific parameters, port, and times for periodic behavior. It then instantiates each service by invoking the correspondent *boot* method, exposing in the scripting side all the necessary Java objects such as loggers and communication end-points. Only then it starts the service controller for each service. Third-party services are also instantiated and started through their specific interfaces. All services have access to the list of other services available, which enables integration.

Service controller: The service controller is responsible for the mechanisms necessary to offer the API abstraction of the framework. It schedules the necessary threads to allow periodic invocation of protocol methods, according to configuration. It handles message passing by managing the necessary sockets for inter-peer communication as well as data marshalling and unmarshalling mechanisms. Since distributed protocols can have multiple periodic procedures executing concurrently, one active thread is started for each cyclic method registered through the *periodic* primitive. Each thread will then periodically invoke, through JSR-223, its respective procedure. A passive thread listens for messages continuously and assigns a worker for processing each incoming message through *receive*, which

in turn inspects the message type and invokes the corresponding event, whose implementation the user must provide.

4 Use Case

In this section we present a concise implementation of the well-known Chord distributed hash table [13], as a use case that shows the benefits of using our framework. Chord maps keys to nodes in a peer-to-peer structured infrastructure. When joining the network, a node receives a unique identifier that determines its position in a ring. Every node is responsible for the keys that fall between itself and its predecessor, keeping track of the latter and maintaining a `finger` table whose entries point to nodes at an exponentially increasing distance, the first one corresponding to its successor. For completeness, we provide the specification found in the Chord paper in Listing 1.3.

The corresponding implementation in our system using Python is presented on Listing 1.4. The `boot` initializes the protocol. Specifically note the registration of all the cyclic procedures using the `periodic` primitive. These are stabilization tasks which have to be performed at regular intervals. Namely, function `stabilize` verifies that a node is its own successor's predecessor and notifies the successor, function `fix_fingers` iteratively refreshes fingers, and `check_pred` checks if a node's predecessor has failed. At the end of initialization, function `join` is invoked, allowing the node to join the Chord ring. Here, only its successor is set, since its predecessor will be updated as part of the stabilization mechanism.

Primitives `call` and `send` respectively perform a non-blocking and blocking (waits for a returned result) communication with another node in the overlay. Their arguments include the destination IP, the name of the event to trigger, and its necessary parameters. The receiver node then invokes `receive` (from the super-class), which in turn applies the correspondent event. Such events include function `find_successor`, that looks for the successor of a given identifier, `get_pred` for returning the predecessor, `poke` working as a *ping*, and `notify` for telling a node that its predecessor might be incorrect. Notice that an extra parameter, `src`, is provided to each event, corresponding to the sender's IP. This is necessary because some protocols require pairwise interactions. Finally, function `closest_preceding_node` returns the highest predecessor of a given identifier found in the finger table, and `between` (from utilities) determines the inclusion of a value in a given range.

All these procedures correspond to executable code that can be readily deployed. As most of the complexity is hidden inside our infrastructure, we end up with an extremely concise specification. In fact, excluding `boot` and the single comment line, and without compromising code legibility, we have 35 LOC, an increase of only 17% over the pseudo-code from the original paper (compare with Listing 1.3), which does not contain initialization code, and a decrease of 15% over SPLAY's implementation [8] (also excluding initialization).

```
1  n.find_successor(id)
2    if(id ∈ (n,successor))
3      return successor;
4    else
5      n' = closest_preceding_node(id);
6      return n'.find_successor(id);

7  n.closest_preceding_node(id)
8    for i = m downto 1
9      if(finger[i] ∈ (n,id))
10       return finger[i];
11   return n;

12 n.join(n')
13   predecessor = nil;
14   successor = n'.find_successor(n);

15 n.stabilize()
16   x = successor.predecessor;
17   if(x ∈ (n,successor))
18     successor = x;
19   successor.notify(n);

20 n.notify(n'):
21   if(predecessor is nil or n' ∈ (predecessor,n))
22     predecessor = n'

23 n.fix_fingers()
24   next = next + 1;
25   if(next > m)
26     next = 1;
27   finger[next] = find_successor(n + 2^(next-1));

28 n.check_predecessor()
29   if(predecessor has failed)
30     predecessor = nil;
```

Listing 1.3. Chord specification as found in the original researh paper [13].

```
1  class ChordService(P2Pservices.Service):
2    def boot(self, **kwargs):
3      self.m = kwargs['m']
4      self.iD = random.randint(1, 2**self.m)
5      self.pred = None
6      self.finger = [None] * self.m
7      self.refresh = 0
8      self.periodic(stabilize,fix_fingers,check_pred)
9      self.join(kwargs['start_node'])

10   def join(self, n):
11     self.pred = None
12     self.finger[0] = self.call(n.ip, find_successor, self.iD)

13   def closest_preceding_node(self, iD):
14     for n in reversed(self.finger):
15         if n != None and between(n.iD, self.iD, iD):
16             return n
17     return self

18   def stabilize(self):
19     x = self.call(self.finger[0].ip, get_pred)
20     if x != None and between(x.iD,self.iD,self.finger[0].iD):
21         self.finger[0] = x
22     self.send(self.finger[0].ip, notify, self)

23   def fix_fingers(self):
24     self.refresh = (self.refresh % self.m) + 1
25     self.finger[self.refresh-1] = self.find_successor((self.iD
           + 2**(self.refresh-1)) % 2**self.m)

26   def check_pred(self):
27     if self.pred != None:
28       try:
29         self.call(self.pred.ip, poke)
30       except Timeout:
31         self.pred = None

32   # invoked by receive
33   def find_successor(self, src, iD):
34     if between(iD, self.iD, self.finger[0].iD):
35         return self.finger[0]
36     n = self.closest_preceding_node(iD)
37     return self.call(n.ip, find_successor, iD)

38   def get_pred(self, src):
39     return self.pred

40   def poke(self, src):
41     pass

42   def notify(self, src, n):
43     if self.pred==None or between(n.iD,self.pred.iD,self.iD):
44         self.pred = n
```

Listing 1.4. Concise implementation of Chord.

```
---
# simulation config
number_of_peers: 1000
simulation_rounds: 20
round_time: 60000
services: [ChordService]

# service config
ChordService:
  port: 32143
  periodic_interval: 5000
  m: 10 # 2^m nodes and keys, with identifiers of length m
...
```

Listing 1.5. YAML file with simulation and service parameters.

As for the configuration, consider for instance the YAML file shown in Listing 1.5. A simulation of twenty rounds (`simulation_rounds`) is defined for a network of one thousand peers (`number_of_peers`), each round running for sixty seconds (`round_time`) of simulated time (time units are in milliseconds). After each round a global observation is performed over the entire network, therefore `simulation_rounds` specifies the number of snapshots to be taken. Each peer runs a single protocol, `ChordService`, whose parameters have to be provided also. General protocol parameters include, for instance, the `port` where the service will run (here 32143), as well as the `periodic_interval` for cyclic procedures (five seconds in this case). Configurations specific to the protocol are also defined, in this case `m` is set to ten, resulting in a ring space of 1024 positions.

5 Related Work

Considering the current approaches to large scale evaluation of distributed systems, PlanetLab [3] is a very valuable global research network for assessing large-scale distributed systems, by allowing experimentation in live networks of geographically dispersed hosts. In a more lightweight approach, network emulators such as ModelNet [15] can reproduce some of the characteristics of a networked environment, such as delays and bandwidth, allowing users to evaluate unmodified applications across various network models, each machine in the cluster hosting several end-nodes from the emulated topology. However, in these test beds, system observability and reproducibility of testing conditions on normal and faulty environments poses several challenges. Despite that existing technologies allow to partially observe the state of the system, a coherently global observation is physically impossible. The lack of knowledge about the system seriously hinders the ability to find and address problems, which is further aggravated by failures and non predictable interactions due to concurrency.

A common approach to this problem is to build a simulation model, that frees testing from the availability of the target platform for deployment and provides perfect observability. Simulators such as ns-2 [12] or PeerSim [11] have been shown to scale to very large systems. However, they can only validate the design and simulation model not the real implementation. This requires maintaining the simulation and real implementations in tandem which is error-prone and time consuming.

An interesting trade-off is achieved by JiST (Java in Simulation Time) [2], an event-driven simulation kernel that allows code to be written as Java threaded code, but avoids the overhead of a native thread by using continuations. JiST does not however virtualize Java APIs and thus cannot be used to run most existing Java code, neither does it accurately reflect the actual overhead of Java code in simulation time. Neko [14] provides the ability to use simulation models as actual code, provided its event-driven API is used instead of the standard Java classes. It also does not accurately reflect the actual cost of executing code, as it uses a simple model that allows the relative cost of the communication and computation to be adjusted. Protopeer [5] allows switching between event-driven simulation and a real deployment without modifying the application. This is achieved by abstracting time and the networking API which offers a limited set of operations. The simulated network can be subject to message delay and loss following models already available or others customized by the developer. The major drawback of Protopeer is the requirement of using a specific API thus precluding the use of off-the-shelf middleware components.

The approach of MINHA [4] is closer to CESIUM [1], which also accurately reflects the cost of executing real code in simulated resource usage. MINHA does however virtualize a significant part of a modern Java platform, thus providing unprecedented support for running off-the-shelf code. In particular, the virtualization of threading and concurrency control primitives provides additional detail when simulating concurrent code, as is usually the case of middleware components.

SPLAY [8] is an integrated system that facilitates the design, deployment and testing of large-scale distributed applications. It also allows developers to express algorithms in a concise, simple language that highly resembles pseudo-code found in research papers. However, SPLAY limits the developer to the Lua language [6] and does not offer facilities for an incremental integration with off-the-shelf existing middleware.

6 Discussion and Future Work

In this paper, we present a unified solution for practical testing and validation of large-scale applications. We achieve this by extending the MINHA simulation platform with a framework for flexibly and concisely prototyping distributed algorithms. The framework allows to effortlessly integrate prototypes with existing middleware components and test them in the large.

We believe this framework can effectively ease the development of large scale distributed systems. Not only ideas can be quickly prototyped and tested but, when developing a complex system, each component can be mocked and progressively improved while the entire system keeps working as a whole. This progressive and iterative development process definitely contributes for higher quality applications.

We plan to assess the platform by implementing a broad number of protocols. With this effort we intend not only to show the usefulness of the framework but also to build a library of useful services that can be used in subsequent applications. Implementing different kinds of protocols will enable us to ensure a considerable expressiveness level for the framework, rather than taking the risk of making it biased towards a particular type of distributed applications. An automated and simplified mechanism for deploying these applications on real environments is also in the scope of our short-term work, this in order to take full advantage of supporting real code. Last but not least, we plan to evaluate the performance of the framework itself. Rather than evaluating the distributed applications themselves, we will assess the simulation overhead and scalability, taking our library of protocols as an increasingly rich benchmark.

Acknowledgment

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant LeanBigData, agreement no. 619606.

References

1. Alvarez, G.A., Cristian, F.: Applying simulation to the design and performance evaluation of fault-tolerant systems. In: Reliable Distributed Systems, 1997. Proceedings., The Sixteenth Symposium on. pp. 35–42. IEEE (1997)
2. Barr, R., Haas, Z.J., van Renesse, R.: Jist: An efficient approach to simulation using virtual machines. Software: Practice and Experience 35(6), 539–576 (2005)
3. Bavier, A.C., Bowman, M., Chun, B.N., Culler, D.E., Karlin, S., Muir, S., Peterson, L.L., Roscoe, T., Spalink, T., Wawrzoniak, M.: Operating systems support for planetary-scale network services. In: NSDI. vol. 4, pp. 19–19 (2004)
4. Carvalho, N.A., Bordalo, J., Campos, F., Pereira, J.: Experimental evaluation of distributed middleware with a virtualized java environment. In: Proceedings of the 6th Workshop on Middleware for Service Oriented Computing. p. 3. ACM (2011)
5. Galuba, W., Aberer, K., Despotovic, Z., Kellerer, W.: Protopeer: From simulation to live deployment in one step. In: Peer-to-Peer Computing, 2008. P2P'08. Eighth International Conference on. pp. 191–192. IEEE (2008)
6. Ierusalimschy, R., De Figueiredo, L.H., Celes Filho, W.: The implementation of lua 5.0. J. UCS 11(7), 1159–1176 (2005)
7. JCP - Java Community Process: JSR-223 Java Scripting API. `https://www.jcp.org/en/jsr/detail?id=223` (2006)

8. Leonini, L., Rivière, É., Felber, P.: Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In: NSDI. vol. 9, pp. 185–198 (2009)
9. Maia, F., Matos, M., Vilaça, R., Pereira, J., Oliveira, R., Riviere, E.: Dataflasks: epidemic store for massive scale systems. In: 2014 IEEE 33rd International Symposium on Reliable Distributed Systems (SRDS). pp. 79–88. IEEE (2014)
10. Matos, M., Felber, P., Oliveira, R., Pereira, J.O., Riviere, E.: Scaling up publish/-subscribe overlays using interest correlation for link sharing. IEEE Transactions on Parallel & Distributed Systems 24(12), 2462–2471 (2013)
11. Montresor, A., Jelasity, M.: PeerSim: A scalable P2P simulator. In: Proc. of the 9th Int. Conference on Peer-to-Peer (P2P'09). pp. 99–100. Seattle, WA (Sep 2009)
12. The Network Simulator NS-2. `http://www.isi.edu/nsnam/ns/`
13. Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F., Balakrishnan, H.: Chord: a scalable peer-to-peer lookup protocol for internet applications. Networking, IEEE/ACM Transactions on 11(1), 17–32 (2003)
14. Urban, P., Défago, X., Schiper, A.: Neko: A single environment to simulate and prototype distributed algorithms. In: Information Networking, 2001. Proceedings. 15th International Conference on. pp. 503–511. IEEE (2001)
15. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., Becker, D.: Scalability and accuracy in a large-scale network emulator. ACM SIGOPS Operating Systems Review 36(SI), 271–284 (2002)