Experimental Evaluation of Distributed Middleware with a Virtualized Java Environment

Nuno A. Carvalho, João Bordalo, Filipe Campos and José Pereira

High-Assurance Software Laboratory
Universidade do Minho
Portugal
{nuno,jbordalo,fcampos,jop}@di.uminho.pt

ABSTRACT

The correctness and performance of large scale service oriented systems depend on distributed middleware components performing various communication and coordination functions. It is, however, very difficult to experimentally assess such middleware components, as interesting behavior often arises exclusively in large scale settings, but such deployments are costly and time consuming. We address this challenge with MINHA, a system that virtualizes multiple JVM instances within a single JVM while simulating key environment components, thus reproducing the concurrency, distribution, and performance characteristics of the actual system. The usefulness of MINHA is demonstrated by applying it to the WS4D Java stack, a popular implementation of the Devices Profile for Web Services (DPWS) specification.

Categories and Subject Descriptors

C.2.4 [**Computer-communication networks**]: Distributed Systems—*Distributed applications*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging*

1. INTRODUCTION

Service oriented architectures are increasingly attractive in a wide range of application scenarios outside typical enterprise information systems. For instance, the Devices Profile for Web Services (DPWS) has been proposed as the base for systems such as large scale smart grids [7, 8] and safety critical medical devices [11]. These broadened requirements pose new challenges to service implementations themselves and to middleware components that are used to support them, as distributed services tend to exhibit complex behaviors that cannot be reproduced with simple tests. And any performance or correctness issues arising in safety critical systems can have catastrophic consequences.

Comprehensive experimental evaluation in a realistic environment is thus a key step in the validation of such systems. This has been done using a spectrum of tools ranging from the orchestration of actual systems to simulators. Obviously, the most realistic conditions can be achieved by running an actual deployment, but this option is costly and time consuming, requiring the availability of the entire system. This becomes even more difficult when dealing with large clusters or large scale systems, even when resorting to platforms such as PlanetLab [10] or virtual machines. As a consequence, toy applications and small benchmarks are frequently used. Observing and registering the results of the test deployment may itself disturb the results, hiding interesting system properties.

On the other hand, building a simulation model frees testing from the availability of the target platform for deployment. By reproducing key aspects of the system in a single addressing space in the context of a high level model of environment, it allows the analysis and global modification of the system's state with no interference. Simulators such as PeerSim [9] have been shown to scale to very large systems. Moreover, a simulation model is useful also while the whole system isn't available, as frequently happens during the development phase. Unfortunately, a simulation model can only be used to validate design. It does not validate the implementation of middleware or services themselves, which often have complex concurrent software components. Moreover, even if some frameworks such as Neko [14] or JiST [3] allow reusing code between the simulation and the implementation, they require the use of a custom API.

An interesting tradeoff is the manipulation of simulation time taking into account the real time measured when executing real code segments, while providing simulation models of concurrency and I/O primitives through native APIs [2, 1]. This paper proposes MINHA, a system that virtualizes multiple JVM instances within a single JVM while simulating key environment components, reproducing the concurrency, distribution, and performance characteristics of the actual distributed system. MINHA is available as open source on `http://gitorious.org/minha`. In comparison with previous work, it makes the following contributions:

- Virtualizes a significant portion of modern Java, allowing off-the-shelf code to run unchanged, including threading, concurrency control, and networking. In fact, it provides something akin to a Java hosted hypervisor, transparently running multiple "virtual" JVMs within a single host JVM.
- Provides simulation models of networking primitives and an automatic calibrator, that adjusts model parameters such that it mimics an existing hardware/operating system combination. This allows performance results obtained with MINHA to be compared with a real system.

In this paper, we provide first a brief introduction to MINHA in Sections 2 to 5, including the simulation kernel, JVM virtualization, and environment models, then in Section 6 we demonstrate automatic calibration and general usage. Section 7 illustrates appli-

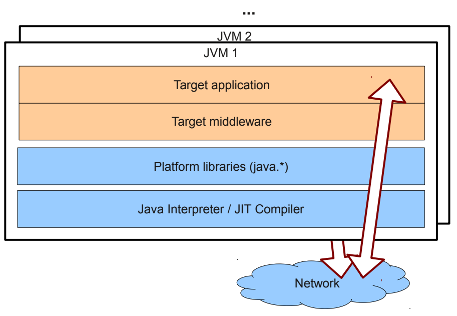


Figure 1: Distributed Java application.

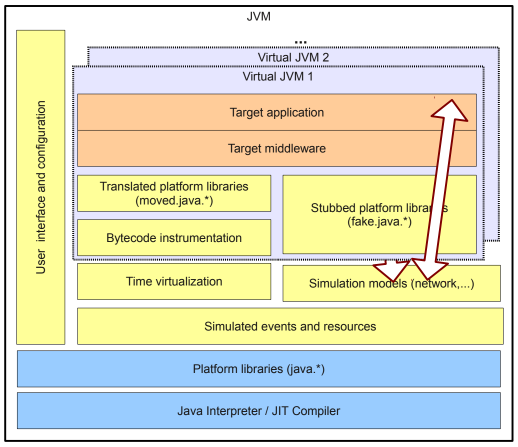

JVM
...
Virtual JVM 2
Virtual JVM 1
User interface and configuration
Target application
Target middleware
Translated platform libraries (moved.java.*)
Stubbed platform libraries (fake.java.*)
Bytecode instrumentation
Time virtualization
Simulation models (network,...)
Simulated events and resources
Platform libraries (java.*)
Java Interpreter / JIT Compiler

Figure 2: Simulation of a distributed Java application.

cability of MINHA by running a distributed application built with the WS4D JMEDS (Java Multi Edition Stack), an implementation of the Devices Profile for Web Services (DPWS) specification. Finally, Section 8 compares MINHA with previous proposals and Section 9 concludes the paper.

2. OVERVIEW

The experimental evaluation of some middleware component usually requires the architecture outlined in Figure 1. Briefly, multiple instances of an application that makes use of the target middleware component are deployed in multiple JVMs. Distributed interactions are then initiated by the application using the middleware, that makes use of platform's libraries and of the underlying Java bytecode execution mechanism. Ideally, these JVMs are scattered across multiple physical hosts, to accurately reproduce the impact of distribution and avoid mutual interference. In fact, the amount of hardware resources to run a real system, or even a set of virtual machines on a smaller number of hosts, is often prohibitive. In fact, even running multiple JVMs on the same host is quite demanding in terms of memory.

MINHA allows reproducing the same distributed run within a single JVM as shown in Figure 2. As we will show, this significantly reduces the amount of memory required even in comparison with the least demanding of the alternatives, which requires multiples JVMs. Moreover, by virtualizing time using simulation, it reduces the interference resulting from competing for shared resources.

In detail, the application and middleware classes for each instance are loaded by a custom class loader that replaces native libraries and synchronization bytecode for references to simulation models. Some of these simulation models are developed from scratch while others are produced by translating native libraries themselves. The resulting code makes use of the simulation kernel and time virtualization to run. Multiple instances are loaded under the control of a command line user interface and configuration loader, which has the following advantages.

Global observation without interference. Once the whole process is centralized, it is possible to get a global observation of all operation and system variables. As the instrumentation is at the simulator side, whenever it is necessary to debug, the introduced overhead is not taken into account opposing to the execution in the real environment. So, the execution time, even when debugging, can be considered for analysis. Moreover, global distributed execution and state can be observed consistently by any simple debugger.

Simulated components. When in a real execution, still in the development phase, it is necessary to evaluate the system for different environments (e.g. network configurations) and software components (i.e. the application layer on top of the middleware). With MINHA, such environments and software models can be replaced by simulation models, and incorporated in a standard test harness to be run automatically as code evolves.

Large scale. Large scale applications, that require a huge amount of resources to be deployed in the real environment, make this development more complicated. Testing during the development phase may require a big share of the system already in operation which requires high costs already in these phase. Some applications might even deal with sensitive information, which implies exhaustive testing to prove its correctness in order to safeguard such critical data.

Automated "What-If" analysis. By resorting to simulated components and running the system with varying parameters, the impact of extreme environments can be assessed, exploring even conditions that are not yet possible in practice. In fact, this can even be automated by having parameters to be generated automatically in order to seek for a given observable condition, e.g. the number of CPU cores to which some code scales up to.

Finally, MINHA can also be extended to perform various fault-injection operations, in space and time domains, as has been accomplished with CESIUM [2].

3. SIMULATION KERNEL

The simulation kernel of the MINHA platform is structured in two layers. The first is a simple event-based simulation kernel, offering only abstract resource management primitives. The second provides the basis for the combination of real and simulated code, by (i) measuring the time of execution and management of a simulated processor; and (ii) allowing sequential Java code to execute by eliminating the inversion of control resulting from the event simulation. The execution of a portion of real code is depicted in Figure 3. First, a `SimulationThread` (ST) is kept for each thread of control, but runs only when a simulation event corresponding to allocation of time to such thread in a CPU resource is scheduled. Concurrency is thus achieved by interleaving the execution of sections of code according to simulation time.

In detail, a simulation event running in the timeline thread T corresponds to the execution of ST between two invocations of the `pause()` method, which blocks the execution of the thread. The state of the stack and program counter are implicitly part of the simulation state and thus do not need explicit management. For ST to advance, the wake-up event that calls into `wakeup()` method must be scheduled. Typically, an executing thread enters its `wakeup()` event in one or more queues or schedules it at some specific instant

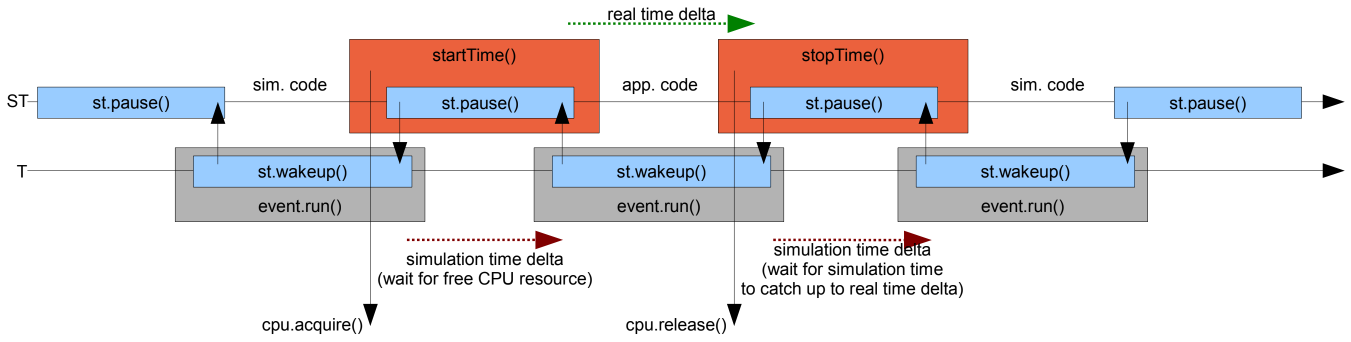


Figure 3: Avoiding inversion of control and achieving time virtualization.

in the future, before blocking in the `pause()` method.

Moreover, the actual time measured in code execution can be reflected in the usage of a resource. In particular, each `SimulationThread` is associated with a resource that represents the processor. At any given simulation instant, it is possible to execute the `startTime()` method, which postpones the execution for a simulation instant on which the processor is free. The thread execution continues until the `stopTime()` method is invoked, being the corresponding interval of time measured. The simulated processor is then marked as busy during the corresponding real time delta. Thus the simulation time advances according to the actual time spent to perform a sequence of code.

Note that the real time delta is measured with a thread-local virtualized CPU cycle timer. This avoids interference by other threads running in the same host and provides an accurate cost of instructions actually executed by the thread under control of the simulator. Finally, this works only as long as threads block only within the `pause()` method and nowhere else. The following section addresses this issue by ensuring that all Java instructions and library operations that can potentially block are virtualized.

4. VIRTUALIZED JVM

For simulation to reflect the real time of the execution of a sequence of code in the occupation of a simulated processor, blocking operations such as thread synchronization and input/output must be avoided and translated into corresponding simulation primitives. Moreover, code executing in different virtual instances cannot interfere directly through shared variables.

These goals are achieved with a custom class loader that uses ASM Java bytecode manipulation and analysis framework [4] to rewrite classes, introducing calls into the simulation kernel as appropriate. Isolation of different virtual JVMs is achieved by using a separate instance of this class loader for each virtual JVM. A subset of classes, containing the simulation kernel and environment models, are kept global by delegating their load to the system's class loader. This provides a controlled channel for virtual JVMs to interact. Although MINHA currently uses this to redirect such interactions through a simulation kernel and to simulated environment models, the same approach could be used for directly sharing real resources.

4.1 Platform libraries

The main challenge is that Java prohibits, for security reasons, the transformation of standard classes under the `java.*` package, which contains the ones that need replacement for virtualization. The solution is to rewrite all references to such classes that need to be re-implemented to stubs in a different `fake.java.*` package.

This is the case for all classes that have native methods, to prevent simulation instances from escaping their sandboxes, and again converted to simulation primitives, or in special cases, such as file system access, encapsulated in the simulation environment and invoked safely by the core. Similarly, special static methods, such as `System.nanoTime()` are intercepted and overridden to meet the semantics of the simulation environment.

Some classes cannot be changed at all, e.g. `java.lang.String`. Fortunately, this is not a problem since they do not contain any static members that would leak information between simulation instances or operations that must be intercepted: synchronization primitives and native methods.

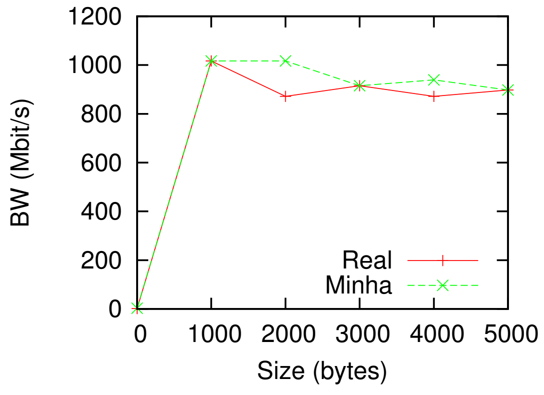
The majority of classes in platform libraries is however 100% Java code that can be translated in the same manner as user code in middleware and applications, as happens with data structures in `java.util.*`. These are analyzed and processed automatically, more specifically, all synchronization primitives are converted into simulation synchronization primitives, becoming part of the `moved` package , e.g. `java.util.Hashtable` becomes `moved.java.util.Hashtable` automatically. Again, all references to these classes are updated accordingly.

4.2 Synchronization

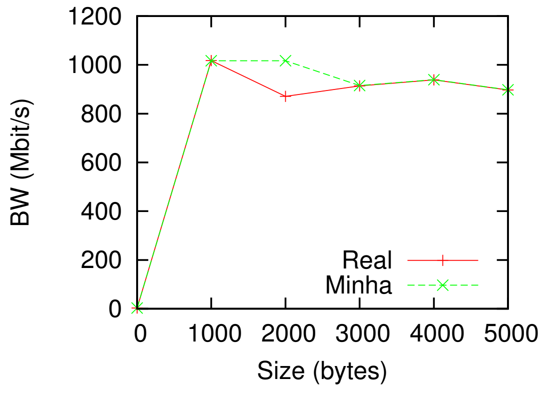
Thread synchronization done with primitives in `java.util.concurrent.*` are easily dealt with by redirecting to their simulated counterparts in `fake.java.util.concurrent.*` as described previously. The remaining challenge is to intercept and translate usage of native Java monitor bytecode operations and implicit mutex/condition variable pair in each object. This is achieved by injecting `fake.java.lang.Object` as an ancestor of all translated classes and then rewriting monitor operations to invocations of methods in this class. Such methods then use simulation primitives for synchronization, namely, for locking/unlocking, waiting, and notifying. `static synchronized` methods are translated by adding an additional field to their class, which contains a singleton instance of an object used for synchronization.

This would however be very costly, as it requires creating two additional objects for each application object and executing two simulation events for each synchronization operation. This overhead is avoided by lazily creating locks and condition variables, as these are not used in the vast majority of application objects, and creating a fast path in synchronization operations that avoids simulation events unless there is contention.

Finally, this approach would still be unfeasible in two cases. The first is explicit synchronization operations on objects that cannot be translated, e.g. `Object[]`, which is solved by keeping a hash table of shadow objects that are used when the default path fails. The



(a) Writing



(b) Reading

Figure 4: Validation of the network configuration (bandwidth).

second is implicit synchronization on class construction, needed for proper implementation of singletons, that cannot be overridden. The solution here is to pre-load classes that take advantage of such synchronization.

5. INPUT/OUTPUT MODELS

Simulation models required by platform libraries encapsulate the expected behavior of significant environment components. Depending on the goal of each run, different simulated components or models with different levels of detail can be used.

5.1 Network

Mainly, for distributed middleware and applications, networking APIs in the Java platform must be mapped to new implementations based on event simulation. Currently, this is done with a high level simulation of a network that trades detail for better simulation performance. The mapping of the IP addresses to the existing instances, i.e., how to know what addresses are allocated to which instance, as well as which addresses are active in MINHA platform, is also the responsibility of the network model.

In detail, the network is modeled as a resource shared by all communication channels, with a finite capacity, that when exhausted prevents access to the overflowing packets. What happens to such packets depends on the transport protocol being used. UDP packets are dropped whereas TCP ones are delayed (but never reordered) until there is enough free bandwidth capacity. The network access control is performed by the *leaky bucket* algorithm in which the internal transmission order of different streams is chosen by the *round robin* algorithm in order to ensure fairness between flows.

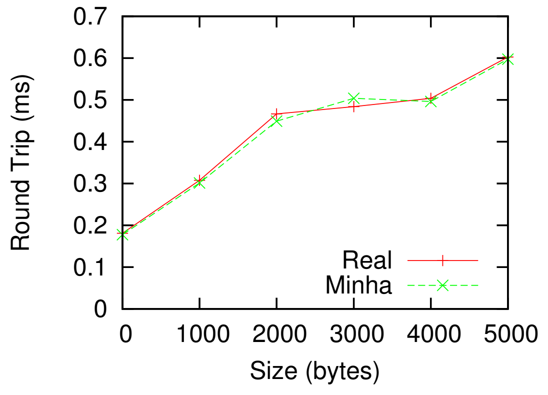


Figure 5: Validation of the network configuration (round-trip time).

The initial version of MINHA supports TCP, UDP, Multicast and IP network protocols through the `java.net` API. The same procedure can be repeated for other APIs (e.g., `java.nio`) and protocols.

5.2 File system

In terms of file system access, MINHA currently intercepts reads and writes in order to avoid direct invocation of native I/O methods by real code, thus providing the illusion of separate file systems to different instances even if running in the same JVM.

Therefore, this does not currently faithfully reproduce the performance characteristics of separate disks and should not be used to asses software components whose performance depends heavily on storage. To achieve that, simulation models of storage components must be build and calibrated, much as is described here for networks.

6. DEPLOYMENT

Using MINHA to assess some distributed application or middleware component requires configuring a number of virtual JVM instances, by providing their command line arguments, and calibrating I/O models to mimic the target environment, with parameters computed from a set of micro-benchmarks.

6.1 General usage

The deployment process of a distributed application is straightforward. We simply start MINHA with the application's main classes as arguments. The following shows the command line that starts a simple echo client/server pair:

```
$ java minha.Run 10.0.0.1 EchoServer, \
                 10.0.0.2 EchoClient 10.0.0.1
```

In this example two virtual JVMs are started. The first runs on a simulated node with the 10.0.0.1 IP address and executes `EchoServer` without further arguments. The second runs on a node with the 10.0.0.2 IP address the `EchoClient` class, with the server's address as a command line argument.

Note that both `EchoServer` and `EchoClient` are standard Java programs making use of the standard API and run also directly on unmodified JVMs. For instance, to achieve the same effect without MINHA, one would run the following on a real host with address 10.0.0.1:

```
$ java EchoServer
```

and the the following on a second real host:

```
$ java EchoClient 10.0.0.1
```

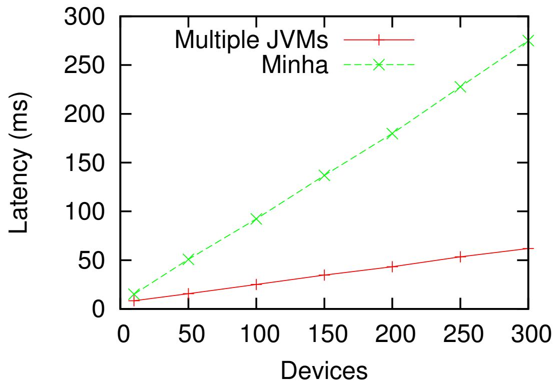
There are more configuration parameters for `minha.Run`, the loader, including automatic assignment of IP addresses, that due to lack of space will not be mentioned in this article.

6.2 Calibration and validation

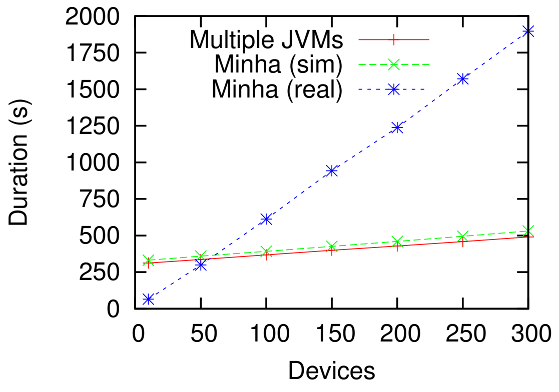
The calibration procedure is driven by the available hardware for testing, hence MINHA is configured accordingly. Since the CPU time spent on each task is dictated by the hardware, the calibration is only valid to the hardware and operating system on which it was performed. In this paper, all the tests were executed in two hosts with the following configuration each: 64-bit Ubuntu Server 8.04.4 Linux, two 12 core AMD Opteron^{TM} Processor 6172, 2.1GHz, 128 GB RAM, 64-bit Sun Microsystems Java SE 1.6.0_24 connected through a Gigabit network.

Network calibration is performed by running two network benchmarks: flood and round trip. The flood benchmark provides us with the maximum outgoing bandwidth of an host, as well as the sending and receiving overheads. The round trip benchmark exhibits the time needed by an host to receive a package. In both benchmarks, 500.000 messages with payloads from 1 to 5000 bytes are exchanged. Network bandwidth was configured as Gigabit, using the *leaky bucket* algorithm to control network access. The network latency is computed from a round-trip benchmark.

Benchmark results of runs with two hosts, in both the real and simulated environments, are compared to verify if they match. Figure 4(a) shows the maximum outgoing bandwidth of a TCP socket using the flood benchmark. Figure 4(b) shows the result of the same benchmark at the receiver's end. Figure 5 shows the result of the round trip benchmark. These results in real and simulated benchmarks are very similar, which validates the calibration for the used hardware.



(a) Latency.



(b) Duration.

Figure 6: Performance and resource usage.

7. CASE STUDY: WS4D

The case study used for evaluating MINHA was a publish/subscribe scenario using Web Services where a publisher notifies several hundred devices that subscribe a specific topic. This is a challenging scenario, since it is costly to obtain such large number of independent devices and very time consuming to setup and run experiments across them.

7.1 Middleware

The Devices Profile for Web Services (DPWS) OASIS standard defines a set of protocols, that resource constrained devices should implement in order to achieve seamless networking and interoperability through Web Services. DPWS-compliant stack implementations must support WS-Eventing, which will be used to implement the notification scenario for the case study. Although WS-Eventing lacks explicit support for brokered dissemination, provided in the WS-Notification family of standards, it embodies a flexible filtering mechanism in the base specification, favoring lightweight implementations and the many-to-one dissemination scenario. It has therefore been the preferred choice for connected devices, namely, within DMTF standard WS-Management [18] and DPWS.

Multiple implementations of the DPWS exist, and more and more vendors are adopting it in their products. For instance, it is included in Microsoft's most recent operating systems, Windows Vista, Windows Embedded CE, and Windows 7, thus being available in most personal computers and various devices such as set-top boxes. There are also some projects, like Service-Oriented Architecture for Devices (SOA4D), managed by Schneider Electric, or Web Services for Devices (WS4D) [17], managed by MATERNA and the universities of Dortmund and Rostock, which provide DPWS stacks implemented in different programming languages. In particular, we selected the WS4D Java Multi Edition DPWS Stack (JMEDS) as it is compatible with both the Standard (J2SE) and Micro (J2ME CLDC) Editions of the Java platform, hence supporting a wide range of devices. For this purpose, the JMEDS framework supplies several facilities of its own alternatively to those provided by the J2SE API, even in the J2SE version, such as data structures (e.g. various types of Lists, Maps, Sets and Iterators); utility classes (some mathematical functions missing on CLDC, string manipulation or logging operations); and thread pools and synchronization locks. It also makes use of a third party XML parser library.

The case study was implemented on version 2 beta 3a of the WS4D JMEDS available for J2SE, which supports DPWS Standard version 1.1, enabling the implementation of both Client and Device entities which are compliant with this specification.

7.2 Application

The evaluation consisted in running an eventing scenario where a value is propagated from a single publisher device to a number of consumer devices, through the use of WS-Eventing. Hence, the producer device provides an eventing service with a notification operation that generates a message to convey new temperature values, and all the consumer devices are configured to contact the producer to subscribe that same notification operation.

The execution procedure of each test comprised the following steps: First, the manager device is started, to control the entire run. The publisher device and the subscriber devices are then started. The manager is notified, through the WS-Discovery Hello messages from each device, that all of them have started correctly. Afterwards, it informs all the subscribers of the publisher's endpoint for them to subscribe to and signals the publisher to start disseminating events. The publisher periodically generates and disseminates new events. When the publisher has disseminated the configured amount of events, it notifies the manager which contacts all the devices to inform that the testing run is terminating and on which file they should write the acquired measurements.

7.3 Results and discussion

The tests consisted in 10 runs for each evaluated number of devices, where each run consisted in the periodic emission of 60 events with an interval of 5 seconds, using one of the hosts described in Section 5. As a baseline, first, each device is run as a separate real JVM. This run is labeled as *Multiple JVMs*. Then, each device is run as a virtual JVM under control of MINHA within a single host JVM. This run is labeled *Minha*. Results presented in Figure 6 are the arithmetic mean of all the runs for each configuration. For latency measurements, the first 10 iterations were discarded in order to minimize the effect of Java JIT compilation, although it also masks the delay of TCP connection establishment.

In detail, Figure 6(a) presents the interval between the emission of a message by the publisher and its reception by a subscriber is then measured in nanoseconds. The sampling of the emission time instant is done right before the emission of a notification by the publisher. The reception time measurement is the first operation in the event treatment performed by the subscriber's designated event sink which deals with new messages. It is interesting that using multiple JVMs underestimates the impact in latency of a growing number of devices, as the large number of CPU cores required to run the test as whole turns out to be used also for parallel dissemination, which would not happen in a real setting with actual devices.

Figure 6(b) shows the total amount of time for each run. Unsurprisingly, without MINHA the test runs in real time and takes approximately 300 s, plus some setup time for devices to start up and connect that grows with scale. With MINHA, the time observed within the simulation closely reproduces the real time necessary without MINHA. However, since MINHA has a simulated timeline (i.e. it does not wait, but instead advances the clock to the next significant instance) it is able to finish the run in less time than for up to 50 devices. After that, the overhead of the simulation means that the simulation takes a significant amount of additional time to run. This is expected to improve significantly in the future as parallel simulation is used in MINHA instead of the simple single-thread kernel currently implemented.

Moreover, this is not a significant challenge to the scalability of MINHA, since the upper bound on the size of a system that can be simulated is in general imposed by available memory. In this case, MINHA has an advantage, since it uses much less memory for a run with 300 devices (almost 5 times less), as portrayed in table 1. This table shows the sum of resident memory used by all the 302 processes involved with multiple JVMs, and the memory used by the single process that runs MINHA.

Table 1: Average memory usage for 300 devices.

Execution mode	RAM (GB)
Multiple JVMs	25.4
MINHA	5.7

8. RELATED WORK

A number of simulation tools have been targeted at distributed computer systems, providing simulation primitives as well as models of computer networks and other components. For instance, OMNeT++ [16] is targeted mainly at networks but its large model library includes also disks, file-systems, and other significant components. It does not however provide a direct route to assessing existing middleware components as possible in MINHA. It should however be possible, and very interesting, to interface MINHA to OMNeT++ to reuse its mature simulation kernel and vast model library.

Simulation kernels such as RacewaySSF [5] can also take advantage of multiple CPU cores to parallelize simulation code and execute large models faster. It should be possible to use such a kernel instead of MINHA's, thus reaping the same benefits. Note however that in MINHA, most of the time is dedicated to running real code, and thus it should be possible to take advantage of multiple cores even if no parallel event-driven simulation is done. Interestingly, the SSF standard uses a dedicated thread for each process marked as not simple, much as MINHA runs Java threads.

An interesting trade-off is achieved by JiST (Java in Simulation Time) [3], 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. JiST does not however virtualize Java APIs and thus cannot be used to run most of existing Java code. It also does not accurately reflect the actual overhead of Java code in simulation time.

Neko [14] provides the ability to use simulation models as actual code, provided that its event-driven API is used instead of the standard Java classes. Moreover, it does not accurately reflect the actual cost of executing code, using a simple model that allows the relative cost of the communication and computation operations to be adjusted.

Combining real and simulated components has also been used previously to evaluate software. For instance, ModelNet [15] simulates a WAN in a LAN to evaluate large scale distributed programs. It requires however that sufficient computer nodes are available to accurately reproduce the behavior of systems. Another example is FAUmachine [12], a virtualization system that can simulate various hardware components. It is however targeted mostly at operating system components and not at distributed systems.

The approach of MINHA is closer to CESIUM [2], 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 simulation concurrent code, as is usually the case of middleware components. Moreover, CESIUM uses `currentTimeMicros()` for timing code execution, which is too coarse and susceptible to interference by background tasks.

A similar approach is also provided by UMLsim [1], a Linux-based hypervisor that virtualizes Linux while providing a simulated timeline and network. This does however require that a separate JVM is run within each virtual machine, which incurs in significant overhead and limits the scale of tests that can be performed. Moreover, the code has not been maintained and is restricted to a now obsolete version of Linux, making it a poor choice nowadays.

In fact, many researchers and developers must resort to actual distributed systems to evaluate their distributed software. Namely, EmuLab [6] provided a set of dedicated computer nodes and networking hardware that could be reconfigured to mimic different large scale systems. By using a decentralized approach, PlanetLab [10] provides a much larger platform to run any off-the-shelf code, with added realism. Although setting up and running experiments in PlanetLab can in itself be challenging, systems such as Splay [13] make it much easier to setup, run, and observe experiments at the expense of limiting the developer to a specific framework and the Lua language.

9. CONCLUSION

This paper introduces MINHA, a simulation platform that allows multiple virtual JVM instances that run off-the-shelf Java middleware components and distributed applications within a single host JVM, much as an hypervisor enables multiple virtual machines within a single server. In contrast with common hypervisors, MINHA manages a simulated timeline which is updated using accurate measurements of the time spent executing real code fragments, hence reproducing the performance of multiple physically independent hosts. Finally, it includes simulation models of networking components for realistically mimicking a distributed execution.

The usefulness of MINHA is demonstrated by showing first how an automatic calibration mechanism ensures that performance measurements accurately reproduce those obtained in a real system. Then, how the WS4D Java stack, an off-the-shelf middleware component, is evaluated on a large scale system with hundreds of simulated devices in single host using $5\times$ less RAM than needed for separate JVMs. In fact, MINHA avoids the error introduced by running all devices within a single host, competing for the same CPU resources, and provides a more truthful approximation of a distributed system.

10. ACKNOWLEDGMENTS

This work was partially funded by PT Inovação and by FCT through project SFRH/BD/66242/2009.

11. REFERENCES

[1] W. Almesberger. umlsim - A UML-based simulator. In *Linux.Conf.Au*, 2004.

[2] G. Alvarez and F. Cristian. Applying simulation to the design and performance evaluation of fault-tolerant systems. In *16th Symposium on Reliable Distributed Systems (SRDS'97)*, 1997.

[3] R. Barr, Z. Haas, and R. van Renesse. JiST: An efficient approach to simulation using virtual machines. *Software–Practice and Experience*, 35(6), May 2005. Reprinted in Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks, Jie Wu, editor.

[4] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[5] J. H. Cowie, D. M. Nicol, and A. T. Ogielski. Modeling 100,000 nodes and beyond: Self-validating design. *Computing in Science and Engineering*, 1999.

[6] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the Emulab network testbed. In R. Isaacs and Y. Zhou, editors, *USENIX Annual Technical Conference*, pages 113–128. USENIX Association, 2008.

[7] S. Karnouskos and T. de Holanda. Simulation of a smart grid city with software agents. *Computer Modeling and Simulation, 2009. EMS '09. Third UKSim European Symposium on*, pages 424 – 429, 2009.

[8] S. Karnouskos and A. Izmaylova. Simulation of web service enabled smart meters in an event-based infrastructure. *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 125 – 130, 2009.

[9] A. Montresor and M. Jelasity. PeerSim: A scalable P2P simulator. In H. Schulzrinne, K. Aberer, and A. Datta, editors, *Peer-to-Peer Computing*, pages 99–100. IEEE, 2009.

[10] L. Peterson and T. Roscoe. The design principles of PlanetLab. *SIGOPS Oper. Syst. Rev.*, 40:11–16, January 2006.

[11] S. Pöhlsen, S. Schlichting, M. Strähle, F. Franz, and C. Werner. A dpws-based architecture for medical device interoperability. In O. Dössel, W. C. Schlegel, and R. Magjarevic, editors, *World Congress on Medical Physics and Biomedical Engineering, September 7 - 12, 2009, Munich, Germany*, volume 25/5 of *IFMBE Proceedings*, pages 82–85. Springer Berlin Heidelberg, 2009.

[12] S. Potyra, V. Sieh, and M. D. Cin. Evaluating fault-tolerant system designs using FAUmachine. In *Proceedings of the 2007 workshop on Engineering fault tolerant systems*, EFTS '07, New York, NY, USA, 2007. ACM.

[13] SPLAY Project. http://www.splay-project.org/.

[14] P. Urban, X. Defago, and A. Schiper. Neko: a single environment to simulate and prototype distributed algorithms. In *Information Networking, 2001. Proceedings. 15th International Conference on*, pages 503 –511, 2001.

[15] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36:271–284, December 2002.

[16] A. Varga. OMNeT++. In K. Wehrle, M. Günes, and J. Gross, editors, *Modeling and Tools for Network Simulation*, pages 35–59. Springer, 2010.

[17] Web Services for Devices (WS4D) Project. http://www.ws4d.org/.

[18] Web Services Management (WS-Management) Standard. http://www.dmtf.org/standards/wsman.