Experimental Evaluation of Distributed Middleware
with a Virtualized Java Environment

Nuno A. Carvalho, João Bordalo, Filipe Campos and José Pereira
HASLab / INESC TEC
Universidade do Minho

MW4SOC’11
December 12, 2011
Service oriented architectures span a wide range of application scenarios

- Geographically dispersed
- Deployed outside enterprise information systems

Comprehensive evaluation requirements

- Correctness
- Performance
Current evaluation solutions

**Simulation models**: useful while the whole system isn’t available, but can only validate design and not the middleware and service implementation

**Actual deployment**: most realistic but costly and time consuming, also requires the availability of the entire system
Traditional experimental middleware evaluation
Traditional experimental middleware evaluation

- Multiple instances of an application are deployed in multiple JVMs
- JVMs are scattered across multiple physical hosts
- The amount of the required hardware resources is often prohibitive
Overview

MINHA middleware evaluation

MINHA

Virtual JVM 1

Target application

Target middleware

JVM

Virtual JVM n

Target application

Target middleware

Platform libraries (java.*)

Java Interpreter / JIT Compiler
**MINHA middleware evaluation**

- Reproduces the same distributed run within a single JVM
- Application and middleware classes for each vJVM are automatically transformed
- Some simulation models are developed from scratch, others are produced by translating native libraries
MINHA middleware evaluation advantages

- Global observation without interference
- Simulated components
- Large scale scenarios
- Automated “What-If” analysis
Agenda

- Simulation Kernel
- Virtualized JVM
- Input/Output Models
- Calibration
- Case Study
Event-based simulation kernel

Abstract resource management primitives
Simulation Kernel

Combination of real and simulated code:

- Measuring the time of execution and management of a simulated processor
- Allowing sequential Java code to execute by eliminating the inversion of control resultant from the event simulation
public class Foo {

    public static void main(...){
        int i = 0;
        while (i<100)
            i++;
    }
}

public class Foo {

    public static void main(...){
        int i = 0;
        while (i<100)
            i++;
    }
}
For simplicity, let’s assume that this segment is a thread.
For simplicity, let’s assume that this segment is a thread.
public class Foo {
    public static void main(...) {
        int i = 0;
        while (i < 100) {
            i++;
        }
    }

    For simplicity, let’s assume that this segment is a thread

public class Foo {
    public static void main(...){
        int i = 0;
        while (i<100)
            i++;
    }
}

For simplicity, let’s assume that this segment is a thread
public class Foo {
    public static void main(...){
        int i = 0;
        while (i<100)
            i++;
    }
}

For simplicity, let’s assume that this segment is a thread
Reflect real time of execution of a sequence of code in the occupation of a simulated processor

Blocking operations (thread synchronization and I/O) must be intercepted and translated into corresponding simulation primitives

Code executing in different virtual instances cannot interfere directly through shared variables
**Bytecode manipulation:** custom class loader that uses ASM Java bytecode manipulation and analysis framework to rewrite classes

**Isolation:** each virtual JVM has its own separate instance of the class loader acting like a sandbox

**Interaction:** A subset of classes, containing the simulation kernel and models, are kept global providing a controlled channel for virtual JVMs to interact
**Bytecode manipulation**: custom class loader that uses ASM Java bytecode manipulation and analysis framework to rewrite classes.

**Isolation**: each virtual JVM has its own separate instance of the class loader acting like a sandbox.

**Interaction**: A subset of classes, containing the simulation kernel and models, are kept global providing a controlled channel for virtual JVMs to interact.
**Bytecode manipulation:** custom class loader that uses ASM Java bytecode manipulation and analysis framework to rewrite classes

**Isolation:** each virtual JVM has its own separate instance of the class loader acting like a sandbox

**Interaction:** A subset of classes, containing the simulation kernel and models, are kept global providing a controlled channel for virtual JVMs to interact
**Bytecode manipulation**: custom class loader that uses ASM Java bytecode manipulation and analysis framework to rewrite classes

**Isolation**: each virtual JVM has its own separate instance of the class loader acting like a sandbox

**Interaction**: A subset of classes, containing the simulation kernel and models, are kept global providing a controlled channel for virtual JVMs to interact
**Bytecode manipulation:** Virtualized JVM uses a custom class loader that utilizes ASM (Apache Simple API for Manipulation) for bytecode manipulation and analysis, allowing classes to be rewritten.

**Isolation:** Each virtual JVM has a separate instance of the class loader, acting like a sandbox to ensure isolation between different virtual environments.

**Interaction:** A subset of classes, containing the simulation kernel and models, are kept global, providing a controlled channel for virtual JVMs to interact with each other.

**Diagram:**
- **Virtual JVM n**
- **Virtual JVM 1**
- **Virtual JVM 2**
- **Bytecode instrumentation**
- **Time virtualization**
- **Simulation models (network,..)**
- **Simulated events and resources**
- **Platform libraries (java.*)**
- **Java Interpreter / JIT Compiler**
- **Target middleware**
- **Target application**

The diagram illustrates the flow of data and functionalities between virtual JVMs, highlighting the distinct layers and components involved in the virtualization process.
**Platform libraries**

- Java prohibits the transformation of classes under `java.*` package

- Rewrite classes that contain native methods

- Overwrite special static methods, like `System.nanoTime()`

- The remaining classes are analyzed and processed automatically
Synchronization

Primitives in java.util.concurrent.*

Rewrite to fake.*

Java monitor operations and implicit mutex/condition variables

Inject a special fake.java.lang.Object ancestor on all translated classes and rewrite monitor operations to invocations to methods on this class

static synchronized methods are solved in a similar way with a singleton object
**Filesystem**

- Reads and writes are intercepted in order to avoid direct invocation of native methods, thus providing separate filesystems to different virtual JVMs.

**Network**

- Modeled as a resource shared by all communication channels with a finite capacity.
- Access control is performed by the leaky bucket algorithm.
- TCP and UDP sockets, including Multicast, supported through the java.net API.
Network

- Bandwidth
- Sending and receiving overheads
- Latency

Performed by running two benchmarks:

- Flood
- Round-trip
Bandwidth with realistic behavior

a) Writing

b) Reading
Latency with realistic behavior

c) Latency

![Graph showing latency with realistic behavior](image)
Case Study

Devices Profile for Web Services (DPWS)

- Standard that defines a set of protocols for devices to achieve seamless networking and interoperability through Web Services
- Proposed as the base for large scale smart grids and safety critical medical devices
- Used on recent operating systems, home automation, assembly lines and car industry

Web Services for Devices (WS4D-JMEDS)

- Framework that implements DPWS standard
- Supports J2SE and J2ME
Membership notification

Manager finds peers through multicast
Manager sends producers addresses to peers
Peers register themselves on producers
Producers initiate notification rounds
Number of peers go from 10 to 300
In a normal WS4D deploy we would have

- Each peer on a different device
- Each device with only one CPU core
In a normal WS4D deploy we would have

- Each peer on a different device
- Each device with only one CPU core

Due to hardware restrictions we deployed 300 devices on multiple JVMs on a single host with 24 CPU cores

- Localhost network with minimal latency
- Producer can send up to 24 notifications in parallel (biasing the results)
MINHA eliminates false latency when all peers run on a single host

MINHA is faster than real deployments on I/O bound scenarios (up to 50 times)
Conclusion

Allows off-the-shelf code (bytecode) to run unchanged including threading, concurrency control and networking

Manages a simulated timeline which is updated using accurate measurements of time spent executing real code fragments

Provides simulation models of networking primitives and an automatic calibrator

Allows off-the-shelf middleware stack evaluation deployed on a large scale system with hundred of devices
http://gitorious.org/minha