A Battle Cry for a System-level JVM in Debian

Pablo Duboue[1][2]

DebConf10, NYC

---

[1] pablo.duboue@gmail.com

[2] DrDub on #debian-java

Outline

Outline

What Are Multi-Application JVMs?

- A JVM that supports **isolates** is a VM which allows running multiple applications (processes, tasks)
  - Multiple programs with different classpaths and different `public static void main(String[])` entry points.
- These different applications should not interfere with each other.
  - Running them in the same JVM should produce the same results as in separate JVMs.

Isolates JSR

- The API bits of a Multi-Apps JVM are defined in JSR-121

Krzysztof Palacz and others, JSR-000121 Application Isolation API Specification (2006)

- `javax.isolate.Isolate`
  - `http://jcp.org/aboutJava/communityprocess/nal/jsr121/`

javax.isolate.Isolate

```
// The creating isolate
Isolate i = new Isolate("org.example.App", "test");
i.start();

// The newly created isolate
package org.example;
public class App {
  public static void main(String... args) {
    for(int i = 0; i < args.length; i++)
      System.out.println(args[i]);
  }
}
```

Outline

Why Do We Want Multi-Apps JVMs?

- As Java desktop applications become more popular...
  - Imagine a chat client written in Java
  - Plus a mail client written in Java
  - Plus an office suite, also written in Java
- Not only just “Java” but also **eclipse-based**!
  - And top it off by running on a netbook.
- But it does not need to stop there...
  - You can be hosting a few debian DVDs torrents using azureus (p2p)
  - Having your desktop being indexed with a lucene-based desktop search
  - Doing a voice conversation using SIP-communicator

Lots of Applications, the User Should Expect Trouble.

- Per the MS Windows disclaimer:
  - “running multiple applications will slow down your system”
- Problem is, this is much worse than running machine-compiled code.
  - First, the code has to be recompiled multiple times for each of the different copies
    - Wasted time recompiling the same code over and over again
  - And all these multiple compiled copies have to be kept in RAM
    - Which occupies much more space than the original jars
    - As research shows compilation results in a 6-8 increase in machine code size vs. bytecode (Cramer et al. 1997)

DLLs vs. Java .class

- In a sense, while each .class is the machine code equivalent of a dynamic-load libray, after dynamic (JIT) compilation a copy of each library is duplicated across JVMs
  - Imagine each machine code program you are running has its own, private copy of the glibc loaded in RAM
    - Yes, Java is **that** bad!

Outline

Grzegorz Czajkowski and Laurent Daynès.

- The beauty of working on Multi-Apps JVMs is that there has been plenty of work at research institutions
  - Many of the hard problems have been ironed out
  - And with OpenJDK released, there is a real JVM to work with
- Sun Research Labs, project Barcelona:
  - `http://research.sun.com/projects/barcelona/`
- Three papers worth reading:
  1. Grzegorz Czajkowski, Application isolation in the Java virtual machine (2000)
  2. Grzegorz Czajkowski and Laurent Daynès, Multitasking without Compromise: a Virtual Machine Evolution (2001)
  3. Grzegorz Czajkowski et al., Incommunicado: Efficient Communication for Isolates (2002)

Outline

Some Approaches.

1. Approach-0: Custom Class-loaders.
    - Throw everything into a vanilla JVM.
2. Approach-1: Bytecode Interposition.
    - Throw everything into a vanilla JVM **but** change static fields on-the-fly.
3. Approach-2: JVM Modification.
    - Change the implementation of static fields in the JVM plus sandboxed JNI and shared heaps.

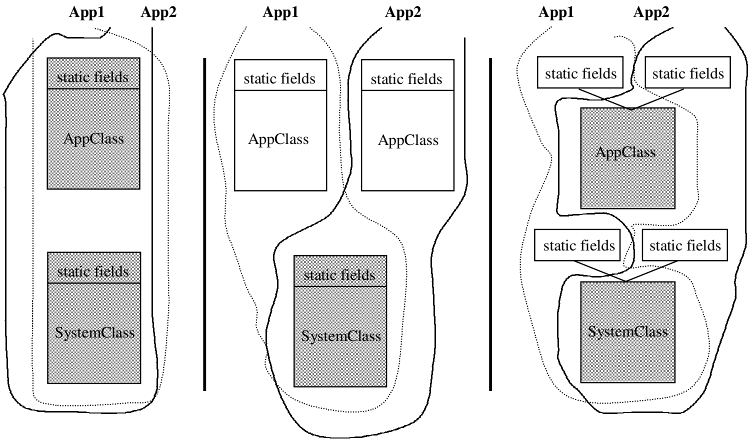
Approach-0: Custom Class-loaders

- Java has a means to let users map from fully qualified class names to the in memory class or sequence of bytecodes implementing the class.
- The different mains are loaded into the JVM and their shared classes are cross-referenced.
- This clearly keeps one version of each class across applications
  - But it produces an unacceptable amount of interference across them.
    - Think `System.setOut(...)`

Approach-0: Custom Class-loaders

- While the custom class-loaders approach seem laughable at first, it is in wide-spread use (!)
  - An application server is just that, in a sense (think tomcat)
- The JVM strict semantics are perfect for application isolation
  - To make it work, a very strict java security manager is in place to protect the system library classes that produce interference
- You don’t get any benefit if you are using the same non-system library in multiple web applications deployed in the same application server.

Approach-1: Bytecode Interposition.



(from Czajkowski '00)

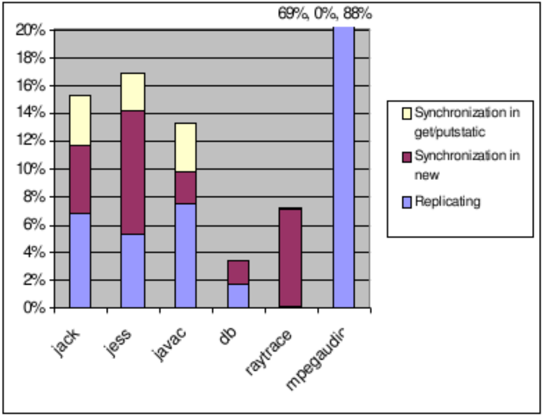
Approach-1: Bytecode Interposition.

```
class Counter {
  static int cnt;
  static { cnt = 7; }
  static void add(int val) {
    cnt = cnt + val;
  }
}
```

Approach-1: Bytecode Interposition.

```
class Counter$sFields { int cnt; }
class Counter$aMethods {
  static Counter$sFields[] sfArr =
    new Counter$sFields[MAX_APPS];
  static Counter$sFields getSFields(){
  int id = Thread.currentAppId();
  Counter$sFields sFields;
  synchronized (Counter$aMethods.class) {
    sFields = sfArr[id];
    if (sFields == null) {
      sFields = new Counter$sFields();
      sfArr[id] = sFields;
      Counter.hidden$initializer();
    }
  }
  return sFields;
}
```

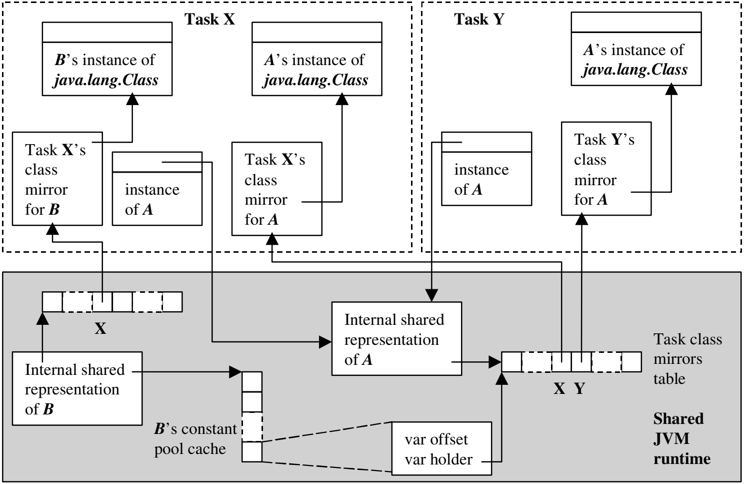
Approach-1: Overheads



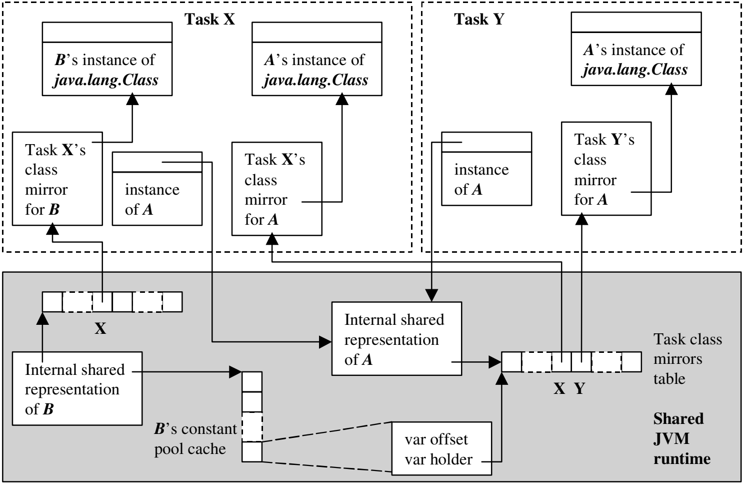
Approach-1: Other Issues.

- Need special implementations for key classes in the java library (e.g., System)
- Different bytecode interposition for architectures that allow for the double check idiom to work well without need for synchronization

Approach-2: JVM Modification.



Approach-2: JNI Sandboxing.



Approach-2: Other Issues.

- Using extra heap space in a best-effort basis
  - Application asks for 2Gb, but MVM is managing 6Gb
    - Application temporarily receives 6Gb until other applications load.
- Class Initialization and Class Resolution Barriers
  - Bits of native code that gets compiled away after the class is initialized
    - In the MVM case, it cannot be compiled away, so it adds to overhead.
- Few system classes still need to be modified as in the previous approach
- These modifications do not support custom class-loaders
  - Eclipse-based applications are still on their own.

Infrastructure Issues

- `/usr/bin/java`
  - The best way to think about it is `screen` vs. `bash`
  - Extra arguments to refer to the instance of the MVM to launch against
- System-level (init.d)
  - If we want to have a system-level started upon boot.
  - Running under which user?
  - Really necessary?

Bug-Reporting Issues

- MVM bugs
  - Can be tricky to debug (interference)
  - Might be related more to incomplete MVM implementations
- If we want to support a MVM we need to give some flexibility to accept MVM-related bug reports.
  - This is in the same line as other non-OpenJDK bug reports (although worse as it pertains to multiple applications)

Regular JVM vs. MVM

- The MVM is a different JDK and will be managed by `update-alternatives` as usual
- However, in many aspects the MVM is a focused fork of OpenJDK
    - The JNI libraries should work and most of the custom JVM arguments.
    - But application wrappers won't detect it as "the" OpenJDK.
- Different system libraries for different architectures
    - For Approach-1, to profit from sound double check idiom implementations.

Supporting Multiple Architectures / JVMs

- Nine Architectures and Four JVMs.
    - Implementing a MVM solution for Debian is not just patching OpenJDK to build a i386 MVM.
- Relationship with GCJ
    - Obviously, GCJ also cares about native code and Java.

Thinking Small

- JIT-cache
  - Maybe we can gain most of the advantages of the MVM by setting up a system global JIT-cache on disk
    - Address only the reduplication of compilation
    - Won't address the memory reduplication (until patched into an 'almost' MVM solution)
- JNI Isolates
  - This might be one of the most interesting features in the MVM
  - We can try to have this in upstream (and into Debian) as an starting point.

Summary

- Keeping multiple copies of a system library in RAM is a solved problem for machine code libraries since the advent of dynamic load libraries
  - However, Java as we have it in Debian (and OpenJDK) can’t do that.
- This problem has been studied (and solved) in the research world.
- It will take effort to get this technology implemented and integrated
  - But it is doable
- Pointers? Contacts? Volunteers?
- DrDub in #debian-java / pablo.duboue@gmail.com