Using AOP to Cluster the JVM

Jonas Bonér
Terracotta, Inc.
jonas@jonasboner.com

Open Terracotta

JVM-level Clustering for:

- POJOs
- Sessions
- Spring
- 3rd Party Frameworks

openTERRACOTTA

Goal of This Session

- Learn how JVM-level clustering and Open Terracotta works and is implemented
- Learn how JVM-level clustering provides a simpler environment for development without hindering scale-out
- Learn how AOP can be used to effectively modularize clustering and distributed computing in general by turning it into a runtime infrastructure service

Let's Start With A Demo

Since a picture says more than a thousand words

Shared Figure Editor

Agenda

- Introduction to Open Terracotta and JVM-level clustering
  - Overview, architecture, features and benchmarks
- The inner workings of a clustered JVM
  - JVM-level Clustering - the killer app for AOP?
  - Challenges to address?
  - How can it be done? Illustrated using AspectJ
- Why do we need JVM-level clustering?
  - Problems with traditional clustering solutions
  - How JVM-level clustering addresses these problems

What is clustering?

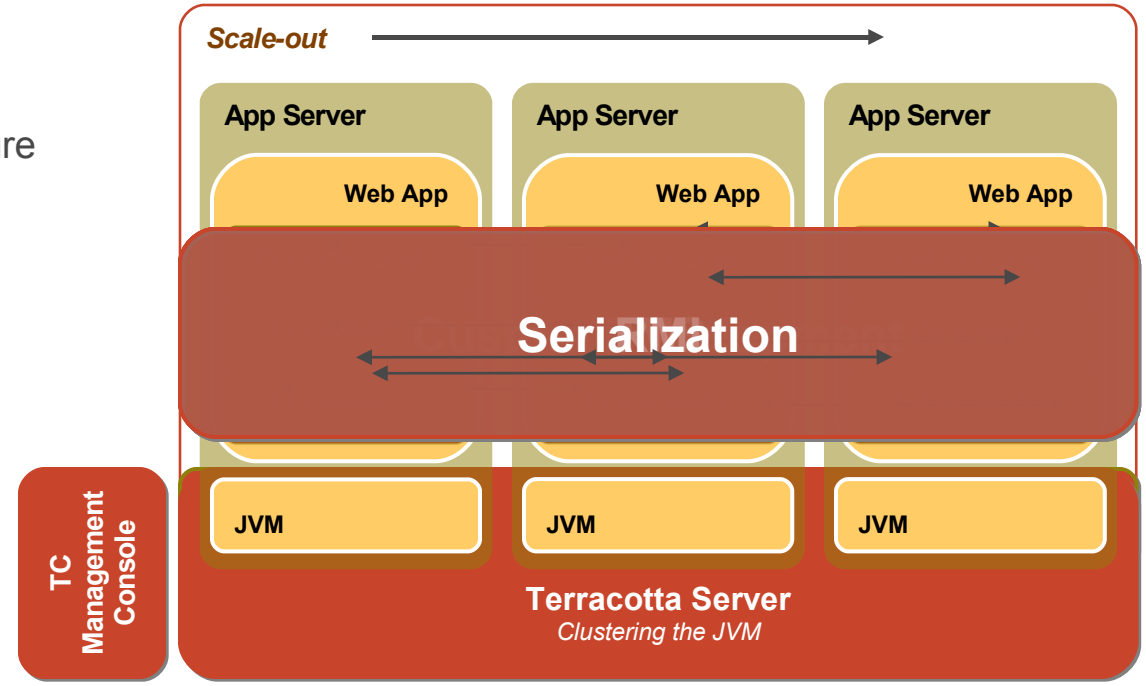
- People want clustering for…
  - High-availability / Fail-over
  - Scalability
  - Coordination
- Traditional clustering solutions are
  - Complex & expensive
  - Intrusive
  - Difficult to implement & manage
  - Solutions are inconsistent
- ..and introduces code scattering and tangling
  - Means that code is harder to write, test, understand, maintain and reuse

Open Terracotta: Open Source JVM-level Clustering

- Open Source under Mozilla-based license
- Feature overview
  - Declarative configuration with minimal or no changes to existing application code
  - No serialization - field-level replication
  - Runtime lock optimizations
  - Distributed thread coordination
  - Virtual heap

Terracotta Approach

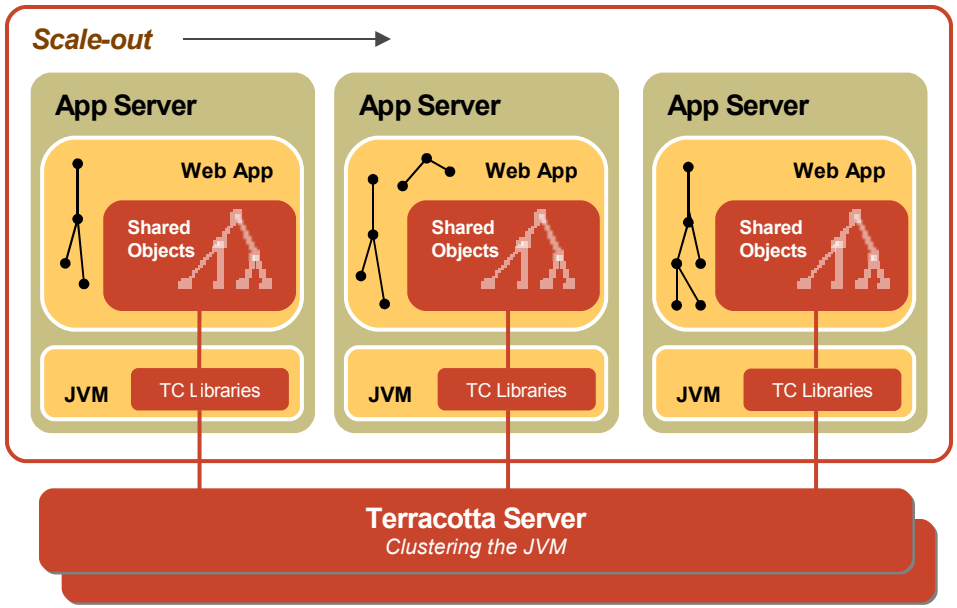
- Today's Reality
  - Scale out is complex
  - Requires custom java code
- Our approach is fundamentally different
  - Cluster at the JVM
- Development Benefits
  - Leverage Existing Infrastructure
  - Substantially less code
  - Focus on business logic
  - Consistent Solution
- Operational Benefits
  - Scale Independently
  - Consistent and Manageable
  - Provides Increased Visibility



Terracotta Use Cases

- **HTTP Session Clustering**
- **Distributed Caching**
- **POJO Clustering - Spring Beans**
- **Collaboration / Coordination / Eventing**
- **Shared Memory / Large Virtual Memory**

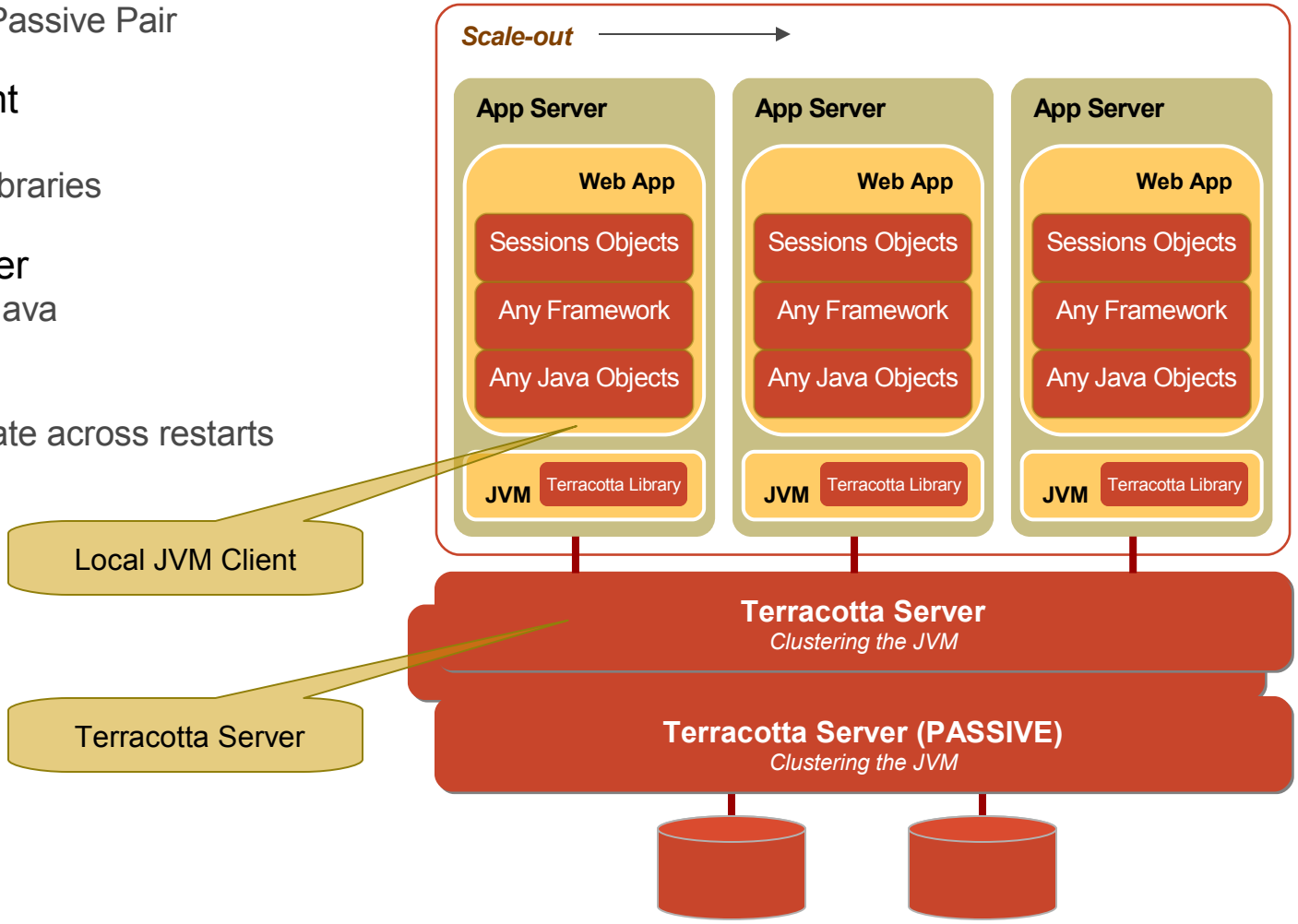
Terracotta Features



- Heap Level Replication
  - Declarative
  - No Serialization
  - Fine Grained / Field Level
    - GET_FIELD
    - PUT_FIELD
  - Only Where Resident
- JVM Coordination
  - Distributed Synchronized Block
  - Distributed Wait()/Notify()
  - Fine Grained Locking
    - MONITOR_ENTER
    - MONITOR_EXIT
- Large Virtual Heaps
  - Configurable
  - As large as available disk
  - Dynamic paging
- Management Console
  - Runtime visibility
  - Data introspection
  - Cluster monitoring

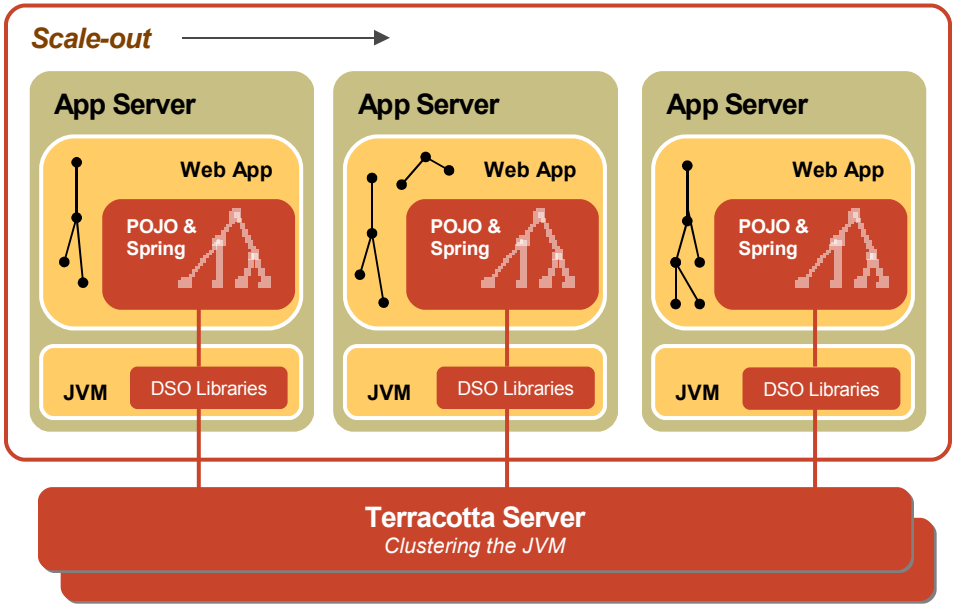
Terracotta Architecture

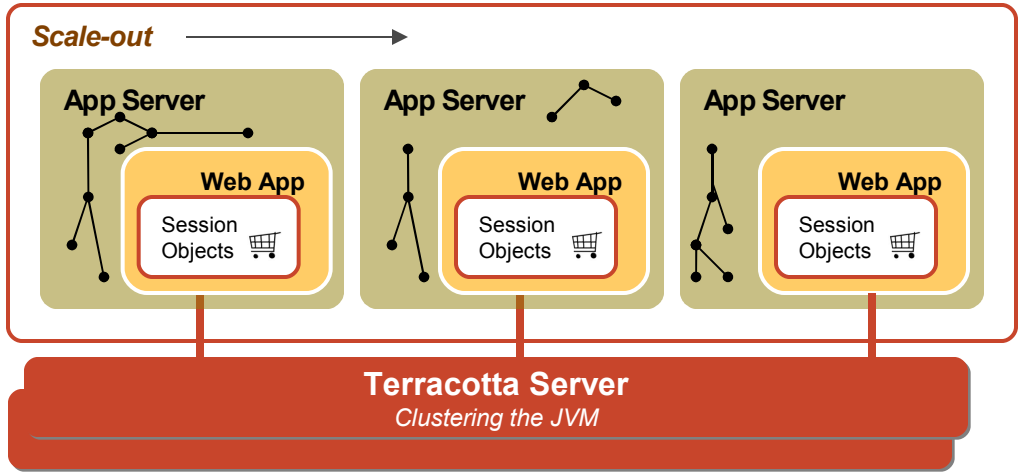
- Hub & Spoke Architecture
  - HA Active / Passive Pair
- Local JVM Client
  - Transparent
  - Pure Java Libraries
- Terracotta Server
  - 100% Pure Java
- Central Storage
  - Maintains state across restarts



Terracotta Products

- Terracotta DSO - POJO's
  - Tomcat Edition (GA)
  - Geronimo Edition (Beta)
- Terracotta Sessions
  - Tomcat Edition (GA)
  - Geronimo Edition (Beta)
- Terracotta For Spring
  - Tomcat Edition
  - Spring Beans
  - Dist App Context Events
  - JMX State
  - Spring Web Flows
    - Session Objects
    - Session Scoped Beans
    - Continuations
- Upcoming
  - JBoss, Jetty, Rife, JRuby, Wicket, Lucene, GlassFish, GlassBox etc.





Zero Impact and still Scalable?

- Hub and Spoke is a SPOF?
- Field-level changes too chatty?
- Locking?
- Networking overhead to clustering?
- Out of memory if we exceed local heap size?

Zero Impact with Scale

- Hub and Spoke ⇒ scale the hub
- Field-level changes ⇒ fine-grained, batched
- Locking ⇒ object level locks - runtime optimized
- Network overhead ⇒ runtime optimized
- Out of memory ⇒ virtual memory - runtime optimize
- So, we can have our cake and eat it too…
- Let's look at some PROOF

TPCW Benchmark – Terracotta with WebLogic

Idle Threads: 0

The number of idle threads assigned to the queue.

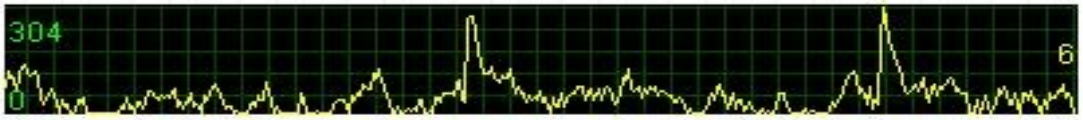
Oldest Pending Request: Fri Sep 15 14:35:23 PDT 2006

The date and time that the longest waiting request was placed in the queue.

Throughput:

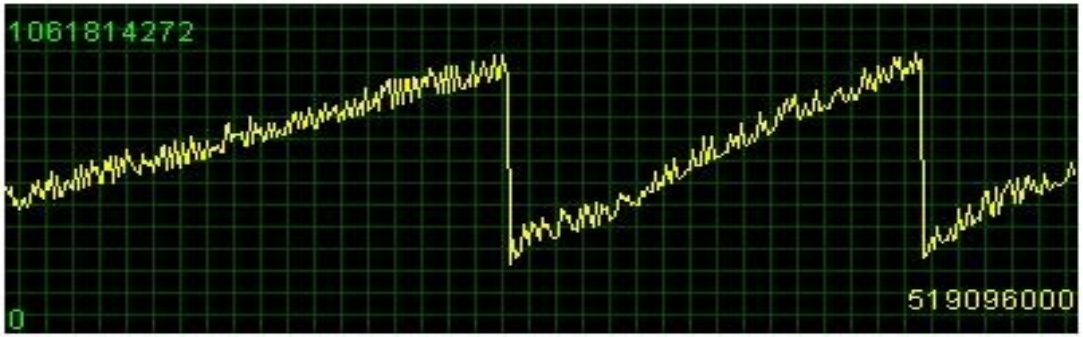
The number of requests that have been processed by the queue.

Queue Length:



The number of waiting requests in the queue.

Memory Usage:



The current amount of memory (in bytes) that is available in the JVM heap.

TPCW Benchmark – WebLogic Clustering

Idle Threads: 50

The number of idle threads assigned to the queue.

Oldest Pending Request: Fri Sep 15 15:51:55 PDT 2006

The date and time that the longest waiting request was placed in the queue.

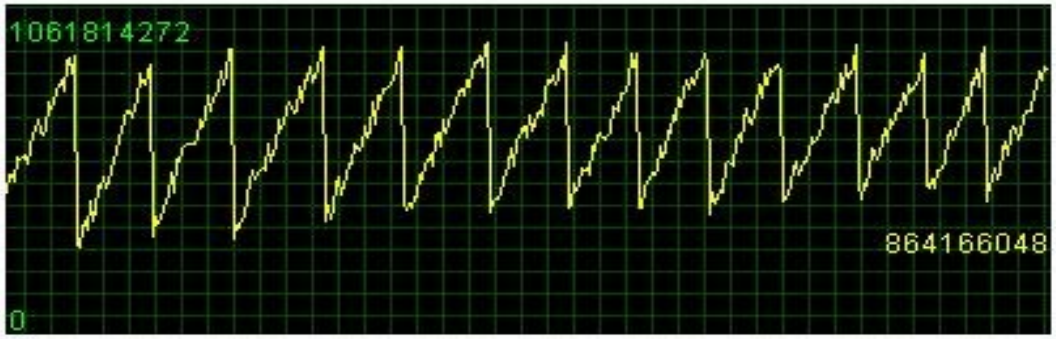
Throughput:

The number of requests that have been processed by the queue.

Queue Length:

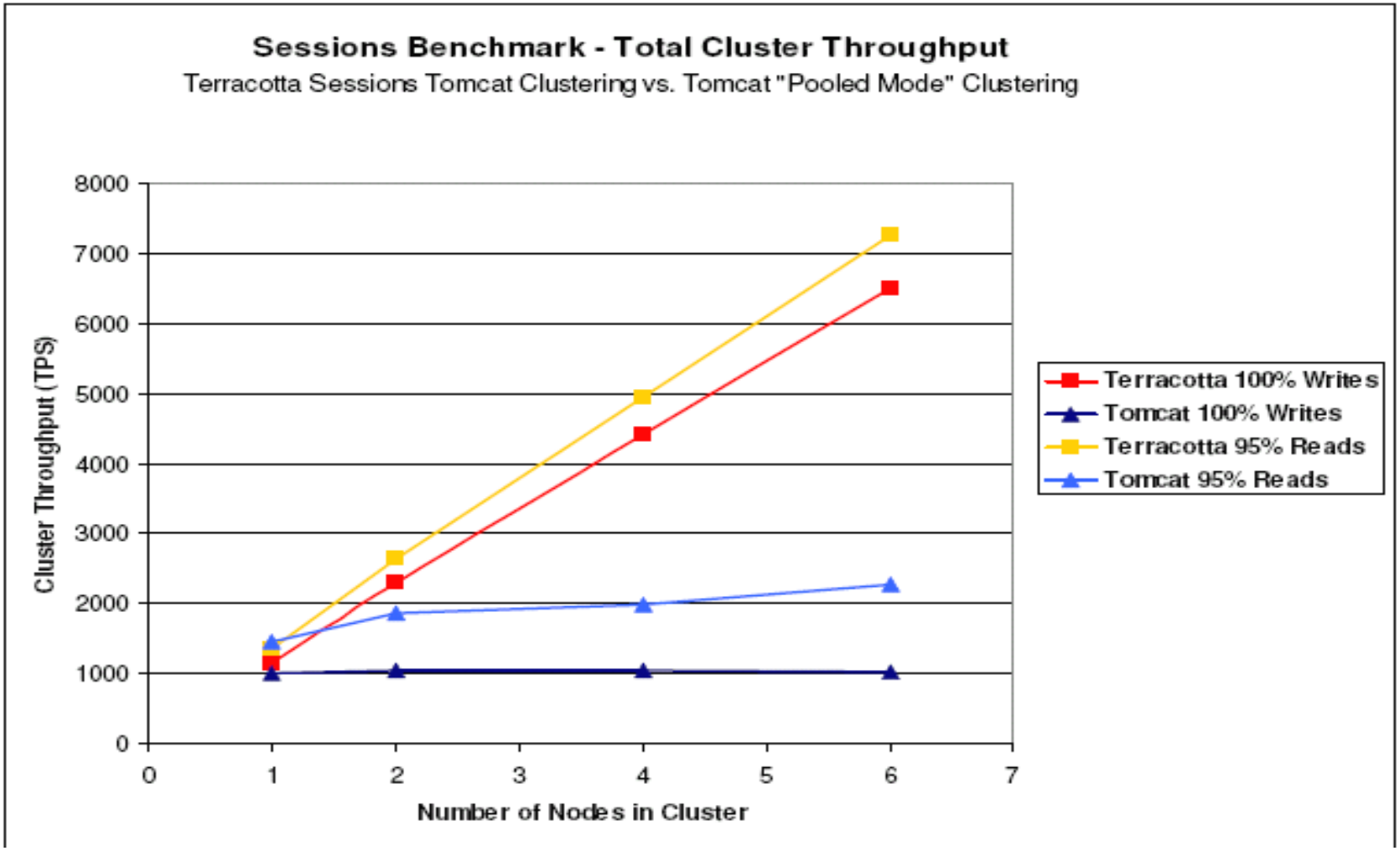
The number of waiting requests in the queue.

Memory Usage:



The current amount of memory (in bytes) that is available in the JVM heap.

Benchmark – TC with Tomcat v/s Tomcat Clustering



DEMO 2

Shared JTable (spreadsheet)
+ code and config

Entire Application

```
package demo.jtable;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;

class TableDemo extends JFrame {

   // Shared object
   private DefaultTableModel model;

   private static Object[][] tableData = {
      {" 9:00", "", "", ""}, {"10:00", "", "", ""}, {"11:00", "", "", ""},
      {"12:00", "", "", ""}, {" 1:00", "", "", ""}, {" 2:00", "", "", ""},
      {" 3:00", "", "", ""}, {" 4:00", "", "", ""}, {" 5:00", "", "", ""}
   };
   TableDemo() {
      super("Table Demo");
      setSize(350, 220);
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      Object[] header = {"Time", "Room A", "Room B", "Room C"};
      model = new DefaultTableModel(tableData, header);
      JTable schedule = new JTable(model);
      getContentPane().add(new JScrollPane(schedule), java.awt.BorderLayout.CENTER);
   }
   public static void main(String[] args) {
      new TableDemo().setVisible(true);
   }
}
```

Magic is in Config File

```
<terracotta-config>
  <dso>
    <server-host>localhost</server-host>
    <server-port>9510</server-port>
    <dso-client>
      <roots>
        <root>
          <field-name>demo.jtable.TableDemo.model</field-name>
        </root>
      </roots>
      <included-classes>
        <include>
          <class-expression>demo..*</class-expression>
        </include>
      </included-classes>
    </dso-client>
  </dso>
</terracotta-config>
```

Inner workings of a clustered JVM

- Maintain the semantics of the application (e.g Java Language Specification (JLS) and the Java Memory Model (JMM)) across the cluster
- Load-time bytecode instrumentation
  - Hook in `java.lang.ClassLoader` and `-javaagent`
  - Transparent to the user -- classes are woven on-the-fly
- Aspect-Oriented Programming techniques
  - AspectWerkz aspects
  - Custom bytecode instrumentation using ASM
  - Declarative configuration (XML)
    - Simplified pointcut pattern language (based on AspectWerkz)

What do we need to maintain across the cluster?

1. State sharing
   - Share data in the Java heap for the instances reachable from a shared top-level “root” reference
   - Make sure each JVMs local heap are in sync (in regards to the clustered parts of the heap)

§ Thread coordination
   - Resource access and guarding
   - Thread signaling

Sample

instrument clusterable classes

put top level root in shared space

```
@Clustered public class Counter {
  @Root public final static Counter instance = new Counter();
  private final Object lock = new Object();
  private volatile int counter = 0;

  public void increment() {
    synchronized(lock) { this.counter++; lock.notifyAll(); }
  }
  p
    synchronized(lock) {
      while(this.counter < expected) {
        try { lock.wait(); } catch(InterruptedException ex) {}
      }
    }
  }
```

lock and start tx

intercept notifyAll

commit tx and unlock

reconcile pending changes

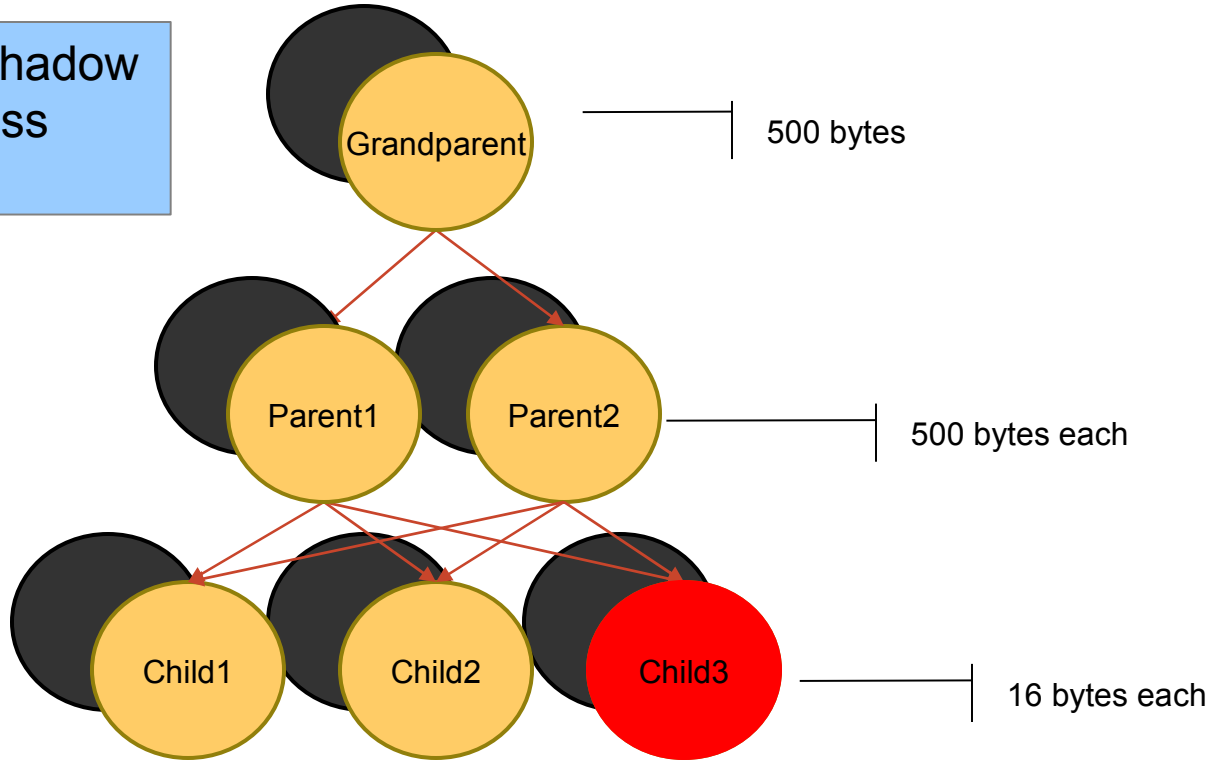
record field change

State sharing - field-level replication and management

- We need to break down the shared object graph into “literals”
  - int, long, float, String etc.
- Need to be able to detect changes to the object graph
  - New object reference is added/removed
  - Literals changes value
- Every reference has a “shadow” that is “managing” the reference
  - Knows if the reference has been changed
  - Can lazily page in and out the actual value
  - Page in: Reconcile changes done in another node
  - Page out: Null out the reference

Field-level replication

each instance has a shadow that is managing access and modification



Total object graph size: 1548 bytes

Terracotta replicates: 16 bytes

Add a “mixin” to all “clusterable” classes

- Using AspectJ’s Inter-Type Declarations

```
declare parents: (@Clustered *) implements TransparentAccess;


public void TransparentAccess.setFields(Map values) {
  ClusterManager.setFields(this, values);
}
public Map TransparentAccess.getFields() {
  return ClusterManager.getFields(this);
}
```

- But Terracotta is using strongly typed direct access

Manage field access and modification

- Intercept the PUTFIELD and GETFIELD bytecode instructions
- Create two advice for managing get and set shared field

```
before(TransparentAccess o, Object value) :
    set(* *.*) && isClusteredTarget(o) && args(value) {
  … // register changed field value in shadow
}

before(TransparentAccess o) :
    get(* *.*) && isClusteredTarget(o) {
  … // reconcile any pending changes
}
```

Thread coordination

- We need to maintain the semantics of the Java Memory Model (JMM) across multiple JVMs
- Maintain the semantics of:
  - synchronized(object) {..}
  - wait()/notify()/notifyAll()
  - Flushing of changes
- Locking optimizations
  - User defined
  - Runtime optimized

Manage synchronized blocks

- Intercept MONITORENTER and MONITOREXIT bytecode instructions

```
before(Object o) :
    isClustered() && lock() && args(o) {
  ClusterManager.lock(o); // lock o in cluster
}
after(Object o) : // after finally
    isClustered() && unlock() && args(o) {
  ClusterManager.unlock(o); //unlock o in cluster + flush changes
}
```

Manage thread coordination (wait/notify)

- Intercept the wait(), notify() and notifyAll() methods in java.lang.Object

```
void around(Object o) :
    isClusteredTarget(o) && call(void Object.wait()) {
  ClusterManager.objectWait(o);
}
void around(Object o) :
    isClusteredTarget(o) && call(void Object.notify()) {
  ClusterManager.objectNotify(o);
}
... // remaining methods omitted
```

Why do we need JVM-level clustering?

- Let's take a step back…

Java™ Language Specification (JLS) is GOOD

- Java has a very strict and valuable set of semantics and rules that developers trust
  - Object identity and pass-by-reference:

    ```
    map.put("ID", obj1);
    Object obj2 = map.get("ID");
    (obj1 == obj2) => true
    ```
  - Coordination between threads:
    - synchronized(..) and wait()/notify()
    - java.util.concurrent.*
- These natural rules of Java should not be broken
  - Breaking these rules open up many problems

Stateless Scale-out Complicates Things

- Replication infrastructure is not up to the task
  - The database is a single point of failure (SPoF)
  - Message queuing or multicast bottlenecks on the network
  - Buddy systems cannot react to cascading failure
- Serializing objects is not fun
  - Breaks your object graph and domain model
  - Leads to coarse-grained replication regardless of delta
- Tuning is never-ending
- What is needed is a JRE service that handles these issues transparently…at runtime

Managed Runtimes Relieve Developers

- Example: Memory Management
  - Remember malloc() and free(), heap vs. stack
  - The JVM introduced most developers to garbage collection, but compile-time used to win
  - Today, the JVM is faster; it decides what do do at runtime instead of at compile time
    - » More information available at runtime
    - » *"…only 12 times faster than C means you haven't started optimizing"* – Doug Lea (on Java GC)
- Other Java features that make Developers' lives easier:
  - Platform-independent thread coordination / locking
  - Fat / thin locks in JRockit JVM decided at runtime
  - Platform specific optimizations

Impact of Development-Time Solutions

- Most existing caching/clustering solutions are API based
- Roughly cache.get() and cache.put()
  - This is a simplified view, of course transactions, replication schemes, fault tolerance, etc. are also included
- These APIs affect **simplicity**
- These APIs affect **scalability**

How API-based Clustering Impacts Simplicity

- Scale-out solutions rely on Java Serialization
- This breaks object identity
  - Data put into the cache and then read back will fail:
  - $(obj == obj) \Rightarrow false$
- Perturbs the Domain Model
  - Management of object references using primary keys
- Adds new coding rules
  - Need to put() changes back - easy to forget
  - Can't trust callers outside the caching class to put a top-level object back in the cache if they edited it
- This is not as simple as the Java language can be

Example of Impact on Simplicity

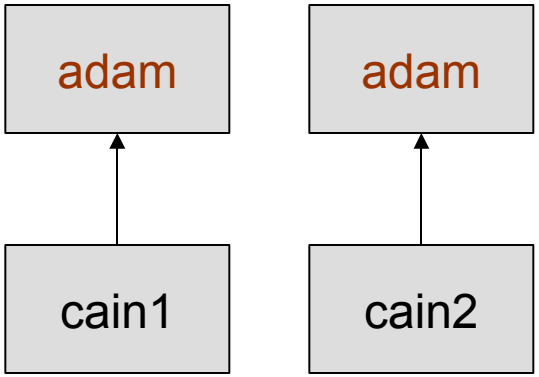
```
// create Cain and put him in the distributed map
Person cain1 = new Person("Cain", adam);
distMap.put("Cain", cain1);


...
// later in time we want to modify Cain
// then we have to get Cain out of the map
Person cain2 = (Person)distMap.get("Cain");
cain2.addBrother(abel);
// then we need to put him back into the map
distMap.put("Cain", cain2);
```

Impact Continued

- Why is it needed to get the object out of the map?
  - Don't we already have a reference to it?
- Why is it needed to put the object back into the map?
- Is it not there already, under the correct key?

Object Identity
Is NOT preserved

You end up with
two distinct
object graphs



How API-based Clustering Impacts Scalability

- Java Serialization is not scalable
- There is a high cost to serialization
  - **Field updates** ⇒ push object graph ⇒ too much data
  - **Coarse-grained locks** ⇒ locking a top-level object in cache, regardless of scope of change ⇒ premature lock contention
- There is a high cost to scale-out
  - DB sees too many clients
  - Clustering takes immeasurable JVM resources (not “too much” just “not factored”)

JVM-level clustering gives you Simplicity & Scale-out

- Java language as programming model
- State distribution and resource coordination is a runtime and deployment artifact - ala garbage collection
- AOP can help you untangle your application code

Download - Use It - Get Involved

http://jonasboner.com

http://terracotta.org

http://www.terracottatech.com

info@terracottatech.com

+1-415-738-4000