Clustered Architecture Patterns: Delivering Scalability and Availability

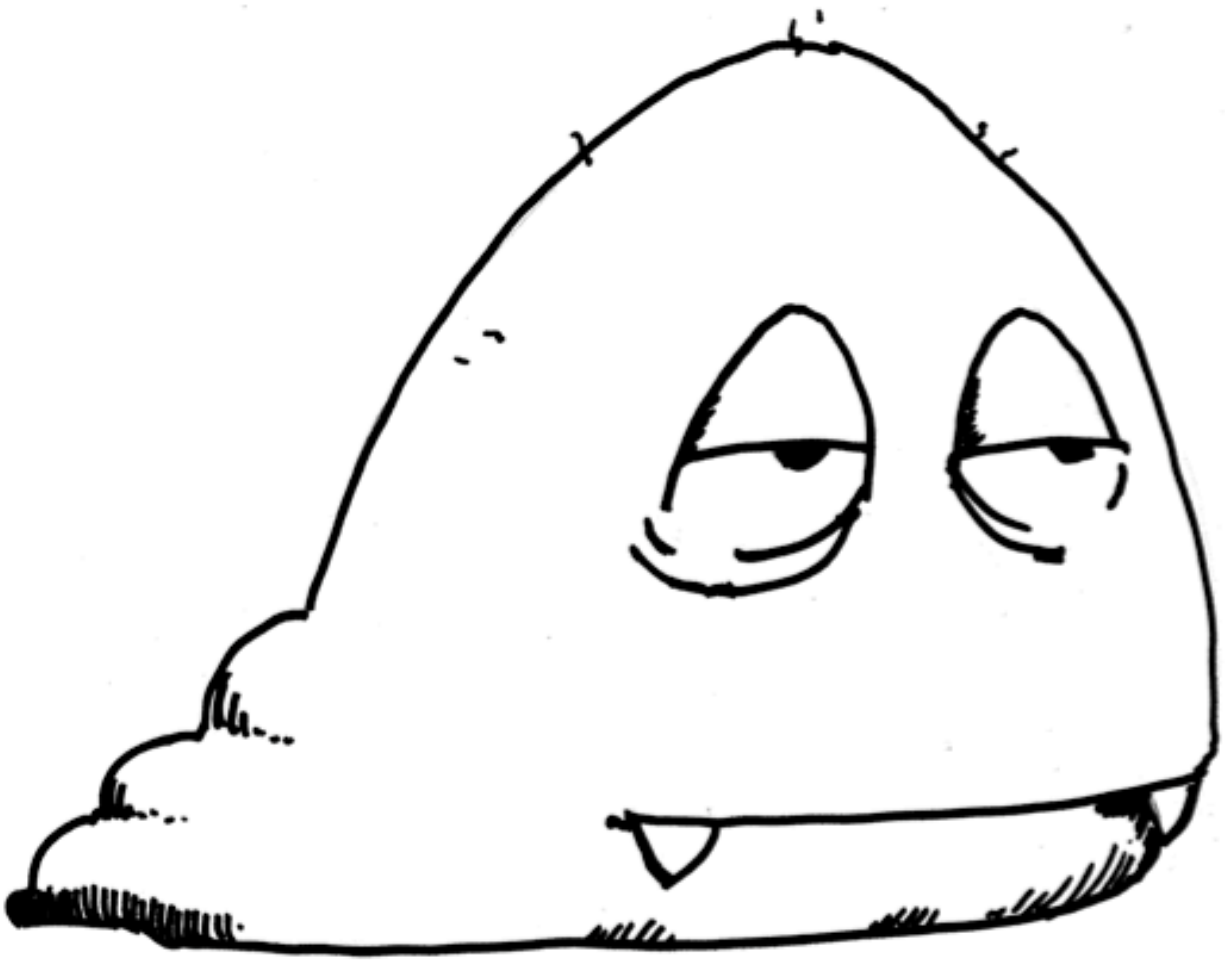
Orion Letizi – Terracotta
Co-Founder and Software Engineer

TERRACOTTA

Agenda

- Patterns from Years of Tier-Based Computing
- Network Attached Memory / JVM-level clustering
- Clustered Architecture Patterns
- When To Use Terracotta: Will It Work With Your Application?
- When To Use Terracotta: Will It Work At Your Scale?
- Examples and Case Studies

The State Monster



The State Monster

- At Walmart.com we started like everyone else: stateless + load-balanced + Oracle (24 cpus, 24GB)
- Grew up through distributed caching + partitioning + write behind
- We realized that “ilities” conflict
  - Scalability: avoid bottlenecks
  - Availability: write to disk (and I/O bottleneck)
  - Simplicity: No copy-on-read / copy-on-write semantics (relentless tuning, bug fixing)
- And yet we needed a stateless runtime for safe operation
  - Start / stop any node regardless of workload
  - Cluster-wide reboot needed to be quick; could not wait for caches to warm
- The “ilities” clearly get affected by architecture direction and the stateless model leads us down a precarious path

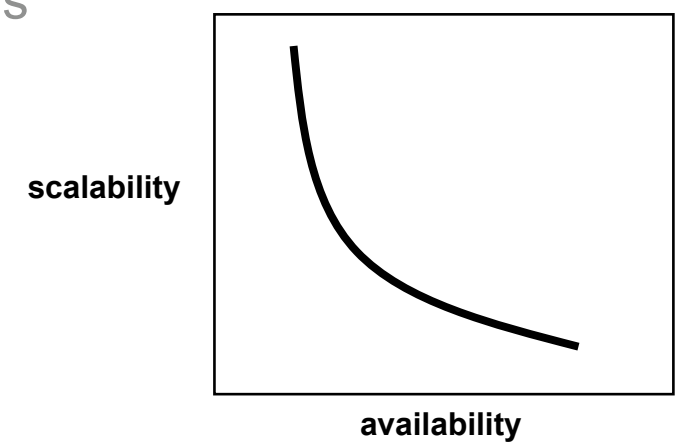
The Precarious Path: Our tools lead us astray

- Stateless load-balanced architecture ⇒ bottleneck on DB
- In-memory session replication ⇒ bottleneck on CPU, Memory
- Clustered DB cache ⇒ bottleneck on Memory, DB
- Memcache ⇒ bottleneck on server
- JMS-based replicated cache ⇒ bottleneck on network

- …Pushing the problem between our app tier CPU and the data tier I/O

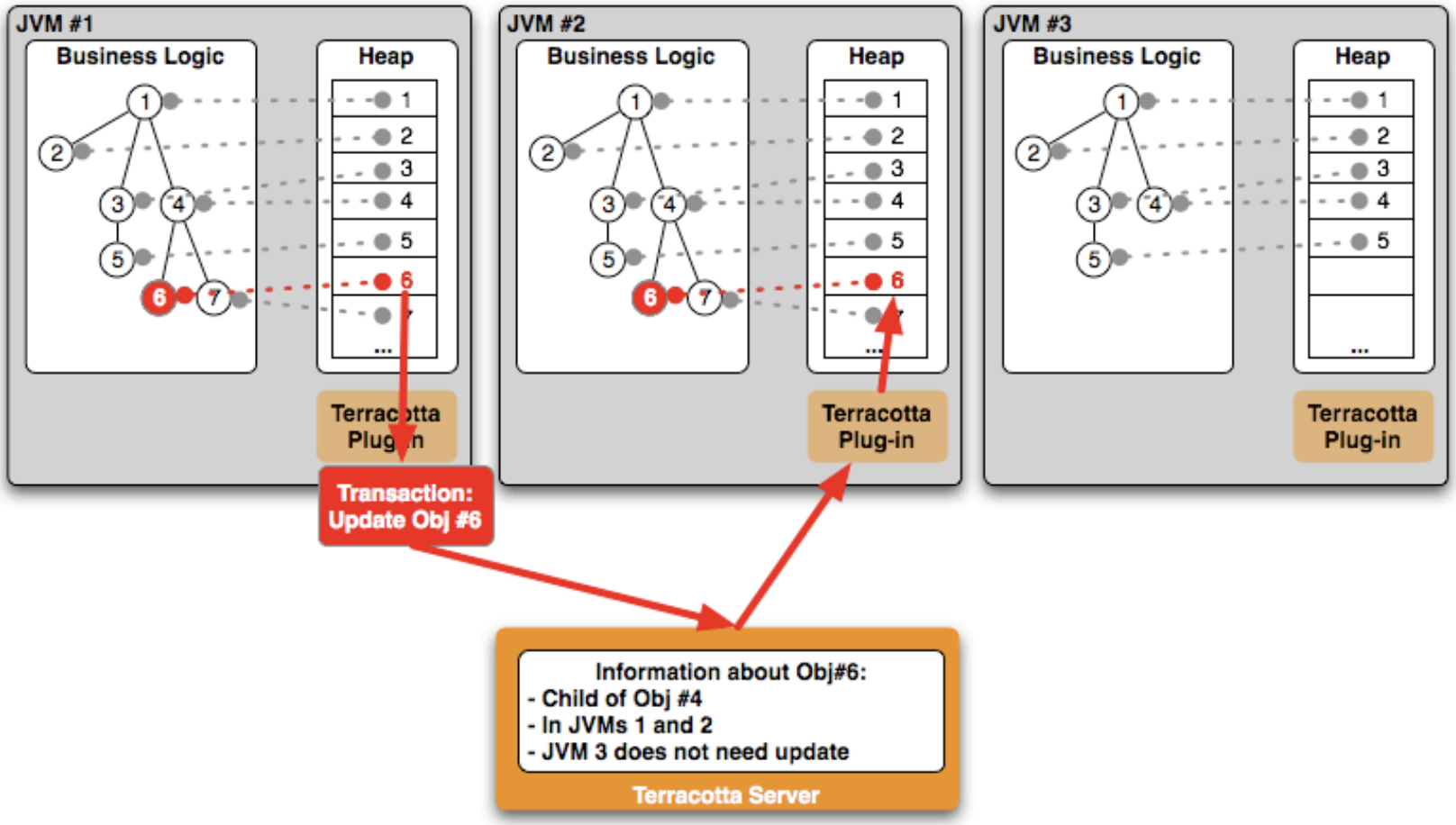
CRUD Piles Up…

- Types of clustering:
  - Load-balanced (non-partitioned) Scale Out
  - Partitioned Scale Out
- Both Trade-off Scalability **or** availability (usually by hand) in different ways



- …and everything we do forces the trade-offs

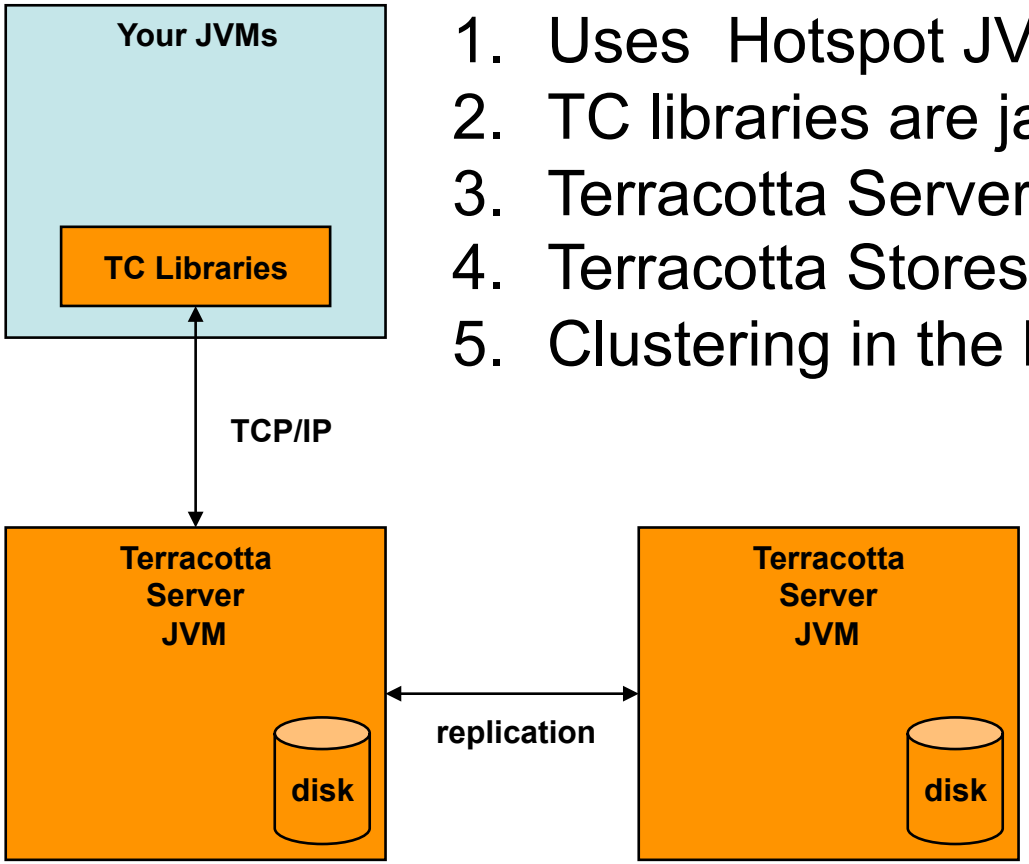
Changing the Assumptions: JVM-level Clustering



Definition

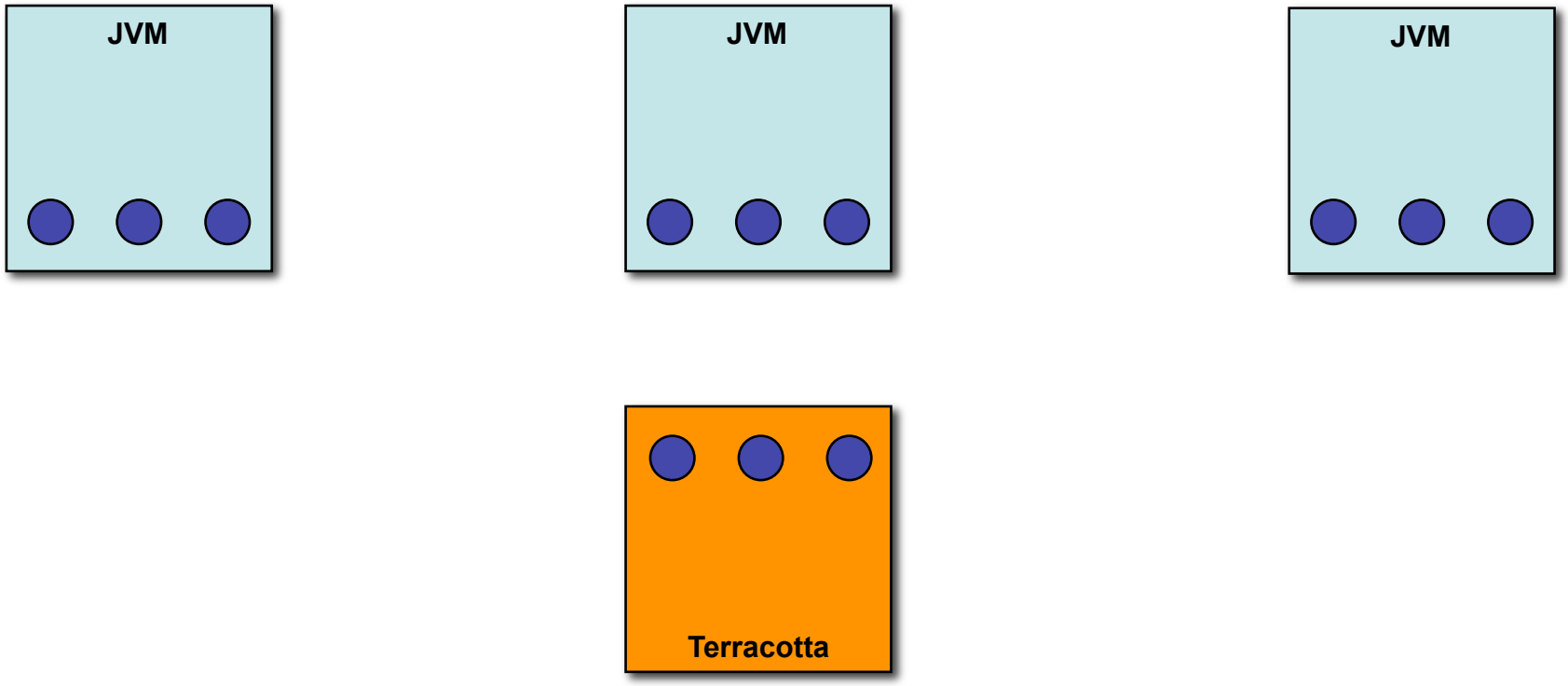
- From “The Definitive Guide…”
  - **T**erracotta is Java infrastructure software that allows you to scale your application for use on as many computers as needed, without expensive custom code or databases
- NAS but for memory (almost)
  - A server process for central storage and locking
  - A transparent client I/O driver
  - A network protocol
- Network Attached Memory (NAM)

Terracotta Deployment Model

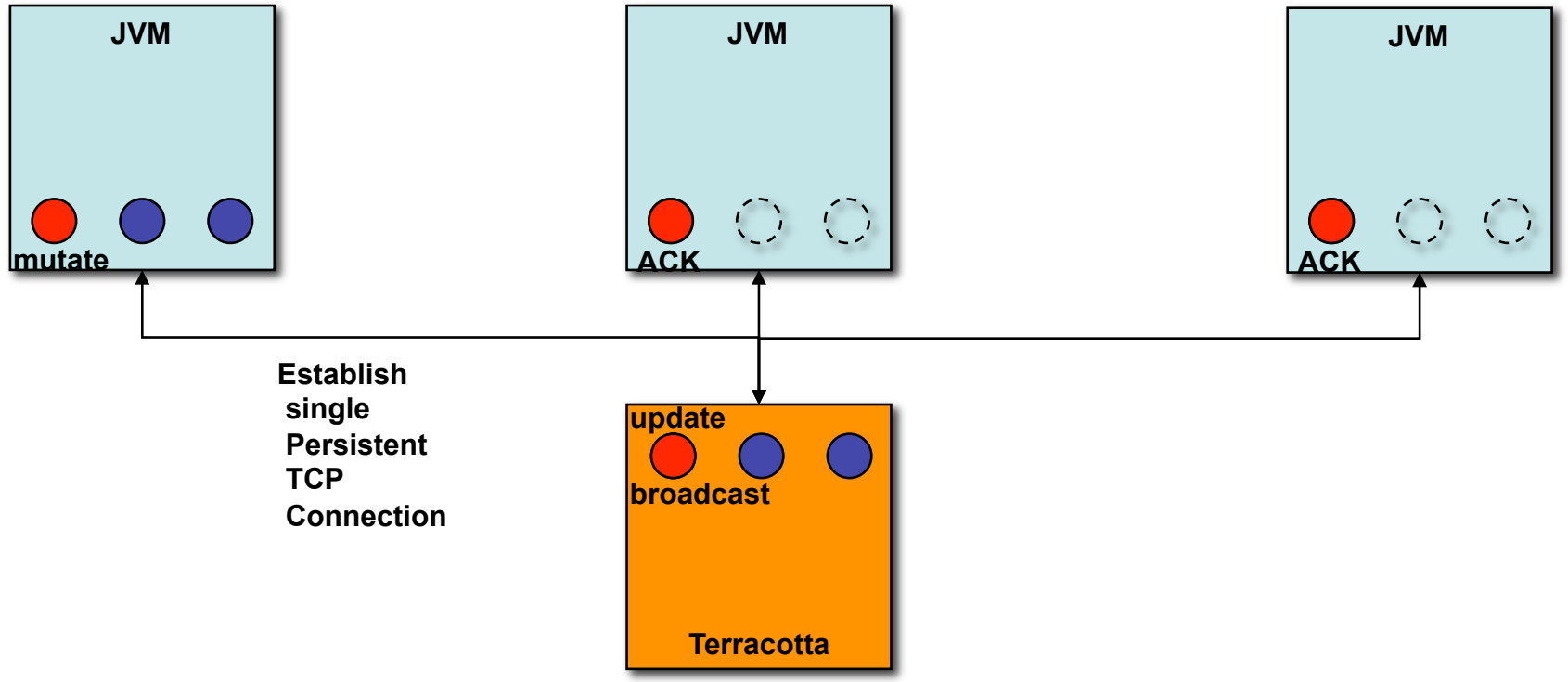


1. Uses Hotspot JVM
2. TC libraries are jars and bootclassloader
3. Terracotta Server is pure Java
4. Terracotta Stores all data to disk
5. Clustering in the box

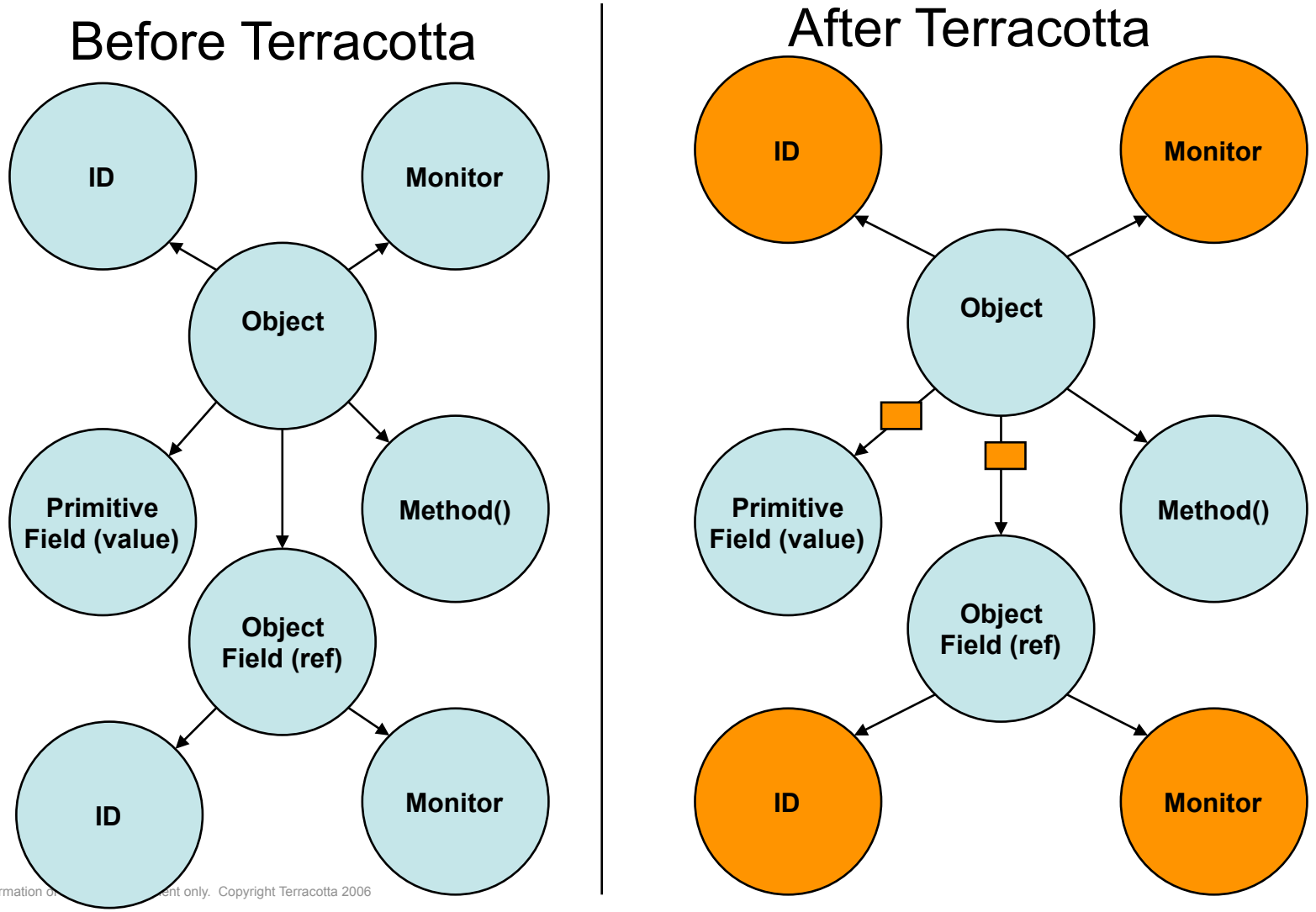
Changes In 1 JVM Visible in Others



…Like This



Oh Yeah: Did I Mention It is JVM-Level?



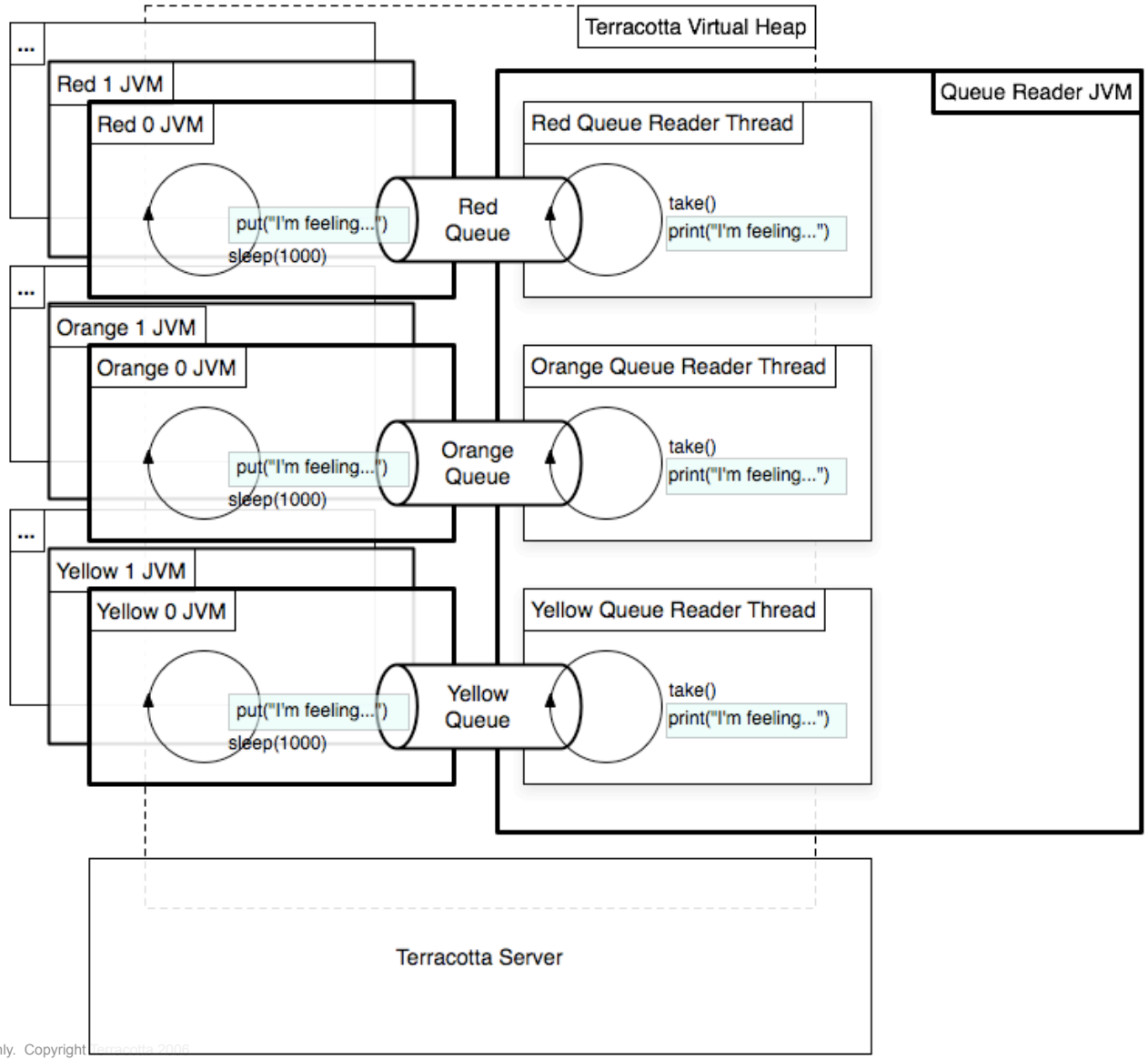
Performance + Reliability + Simplicity

- 10X throughput over conventional APIs
  - All Reads from Cache (implicit locality)
  - All Writes are Deltas-only
  - Statistics and Heuristics (greedy locks)
- Scale out the Terracotta Server
  - Simple form of Active / active available today
  - V1.0 GA this year
- Looks like Java to me (code like your mom used to make)
  - Normal access patterns: no check-out before view and check-in on commit
  - Code and test at a unit level without infrastructure intruding on app logic
  - Threads on multiple JVMs look like threads on the same JVM

Demo Time

- User session clustering (conversation state)
- Queueing

CoordinationWithQueues (from the *Definitive Guide*)



QueueReader

```
private static final class QueueReader implements Runnable {
  private final NamedQueue myQueue;

  public QueueReader(NamedQueue queue) {
    this.myQueue = queue;
  }

  public void run() {
    while (true) {
    try {
      System.out.println(new Date() + ": message from the " + myQueue.getName()
                    + " queue: " + myQueue.take());
    } catch (InterruptedException e) {
    // A real application would do something interesting with this
    // exception.
    e.printStackTrace();
    }
  }
}
```

Queue Writer

```
//...
private void run() {
  while (true) {
    try {
      String mood = MOODS[random.nextInt(MOODS.length)];
      myQueue.add(myColor + " says, \"I'm feeling " + mood + ".\"");
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      // A real application would do something with this exception
      e.printStackTrace();
    }
  }
}
// ...
```

CoordinationWithQueues Config File

```
...
    <dso>
      <instrumented-classes>
        <include>
          <class-expression>
            org.terracotta.book.coordination.queues.CoordinationWithQueues
          </class-expression>
        </include>
        <include>
          <class-expression>
            org.terracotta.book.coordination.queues.CoordinationWithQueues$NamedQueue
          </class-expression>
        </include>
      </instrumented-classes>
      <roots>
        <root>
          <field-name>org.terracotta.book.coordination.queues.CoordinationWithQueues.QUEUES</field-name>
        </root>
        <root>
          <field-name>
            org.terracotta.book.coordination.queues.CoordinationWithQueues.sharedCounter</field-name>
        </root>
      </roots>
      <locks>
        <autolock>
          <method-expression>
            void org.terracotta.book.coordination.queues.CoordinationWithQueues.main(java.lang.String[])
          </method-expression>
          <lock-level>write</lock-level>
        </autolock>
      </locks>
    </dso>
...
```

Batching and Ordering In Accordance With JMM

- Durable
- Transactional
- Clustered
- Shared Memory
- Use synchronized or util.concurrent or Terracotta Integration Module to lock

What TC Does for You

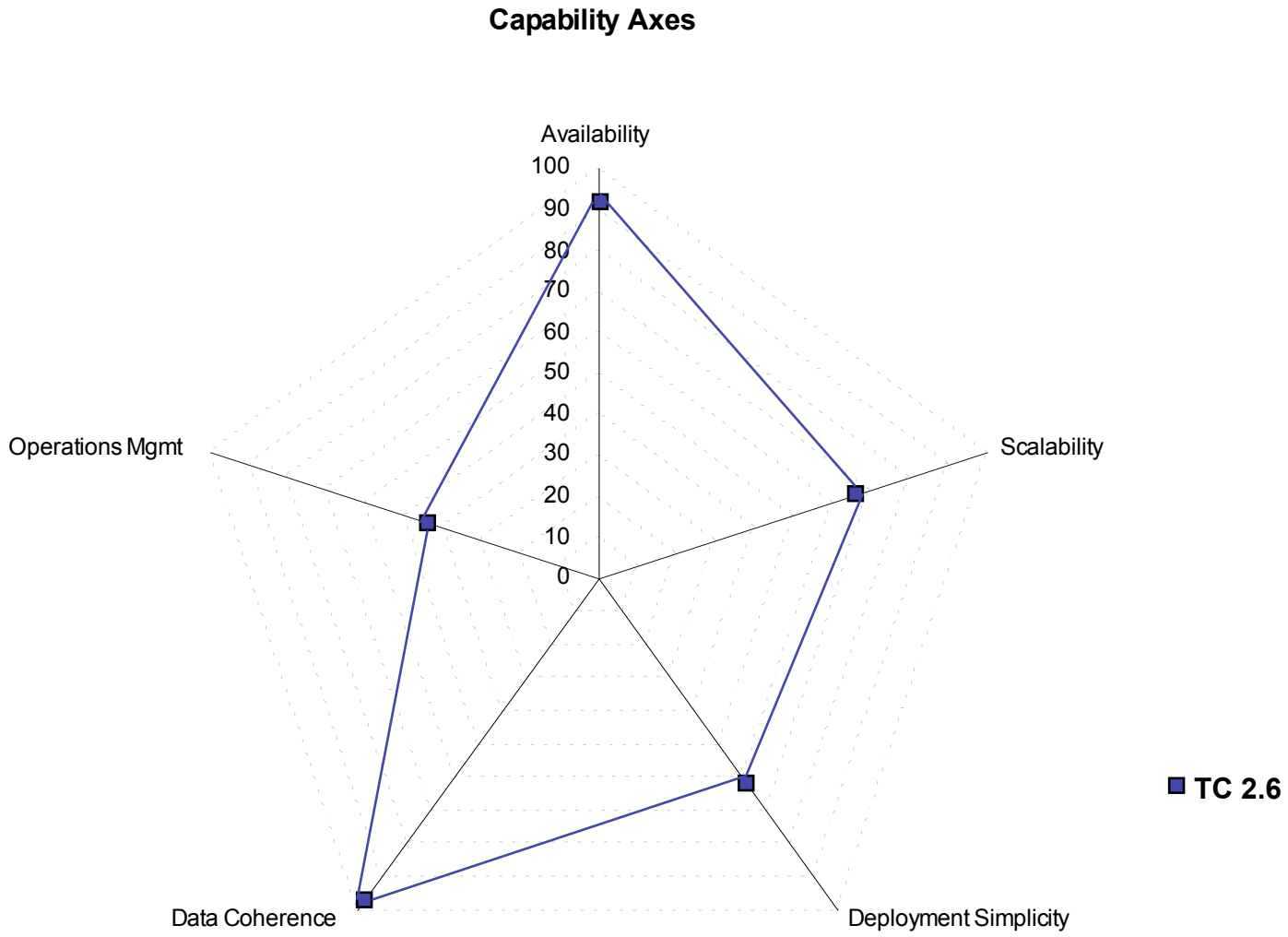
- Pure Java
  - Shares your objects and framework internals
  - Shares thread coordination
- Efficient Scale-out through heuristics and statistics (what is resident where)
- Availability in the box

Reduces Clustering to Tuning

- Always coherent
- Always on disk at in-memory speed (streaming, not seeking)
- Always Pure Java (java.*, synchronized, and util.concurrent)

- Changes needed usually for tuning

Scalability, Availability, and Simplicity Can Be Friends



When to use Terracotta: Will It Work With My App?

(The Terracotta API, so to speak)

TERRACOTTA

Users and Use Cases

- Web Apps
  - Portals
  - Social networking
  - Gaming, betting
- Financials
  - Trading apps
  - Trading bus
- Customer Service
  - CRM apps
  - Rewards / affinity programs
- Telco / IT
  - VOIP
  - Server mgmt

Terracotta is a Platform (not an app server)

- How do you share objects
  - HashMap, CHM, EHCache, HashTable, TreeMap
- How do you coordinate access
  - Wait / Notify, synchronized, util.concurrent
- Where do you store the data
  - In Terracotta as objects, write-behind to DB, write-thru to DB, etc.
- How do you process async workloads
  - Queue of changes, eventing pattern, etc.

API: Using Patterns That Are Clusterable

- Conversation clustering / persistence
- Window onto large dataset
- Data as a service
- Cluster membership and management
- Change Notification
- Shared Metadata

Pattern: Conversation Clustering (impl coming soon)

- Use EHCache
- Set a conversation key (cookie)
- Value should be object graph
  - not primitives like string or byte array
  - Value should not be too deep
  - Values should never reference each other
- Load Balancer required
  - Sticky load balancing (layer 7 preferred, or Apache mod_jk)
- No DB required

Pattern: Window Onto Large Dataset

- Isolate the JVMs from each other carefully
  - ConcurrentHashMap of ConcurrentHashMaps
  - One map per L1 JVM
- If you must lock while updating, lock the value graph itself (fine-grained) using ReadWriteLock
- Enough Heap in each JVM to hold most data it needs
  - High pre-fetch rate
- You must come up with a way to load balance or partition or this will thrash the network

Pattern: Data As A Service (impl coming December)

- Service Oriented
- Write a stateless get() router with ActiveMQ
  - Send all messages to the get() router
  - He hashes and sends the get() to appropriate stripe
  - Transparent to service consumer
- Fairly flat key / value pairs
  - String or Integer key
  - Value == shallow object graph (mostly primitives)
- Key must hash evenly. Don't hash spatially even though it is tempting. (e.g., "New York customers" is worse than "A")
- Use MySQL to index keyspace and execute queries that return vectors of key/value pairs to load
- Update MySQL write-behind (another pattern)

Pattern: Cluster Membership

- Want to know when L1 JVMs come and go
  - Listen for Terracotta JMX events
  - Catch L1 disconnect event and deterministically take over work (next worker ID after failed worker)
- Want to know when L2 JVMs fail over
  - Listen for Terracotta JMX events
  - If you don't reconnect after certain time, system.exit() and use initd to restart yourself
  - Avoid disconnected mode
- Always use a surviving L1 JVM to address a departing one. Do not use System exit hooks! Clean up on restart if last JVM in the cluster
- Don't requeue work, steal queue references instead

Pattern: Change Notification (impl coming soon)

- Use this one for write-behind
- Linked Blocking Queue in each L1 JVM
- Share the LBQ via DSO in a CHM of queues
- Have your changeable objects implement an iChangeable interface. Call flush() method on change. You will have to implement this method
- Daemon thread takes batches of object references and calls flush on each.
- Idempotency is req'd to avoid transactions / JTA. Then peek instead of take and take after flush().
- Use Cluster membership (another pattern) to recover from L1 JVM failure

Pattern: Shared Metadata

- Composite Pattern…

- Use EHCache + Linked Blocking Queue
- Conversation Clustering pattern
- Change Notification pattern

When To Use Terracotta: Will It Work At My Scale?

TERRACOTTA

Scaling Thru Controlling Scope of Object Deltas

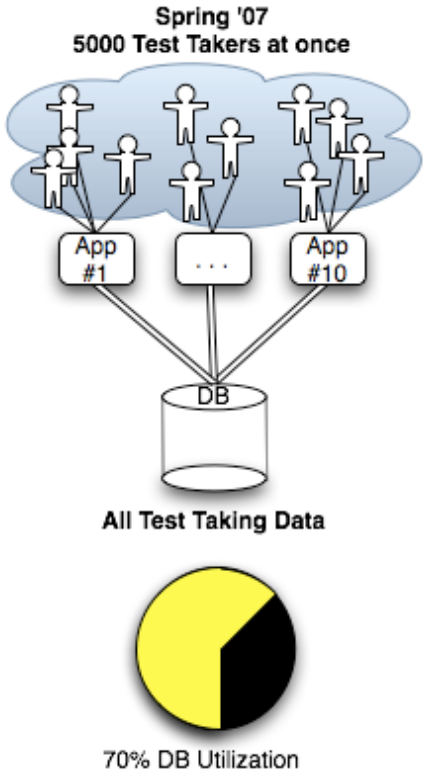
- Locality of Reference
- Object graph complexity
- Garbage creation rate
- Lock granularity and contention
- Server sizing (GC @ 64bit vs. headroom)

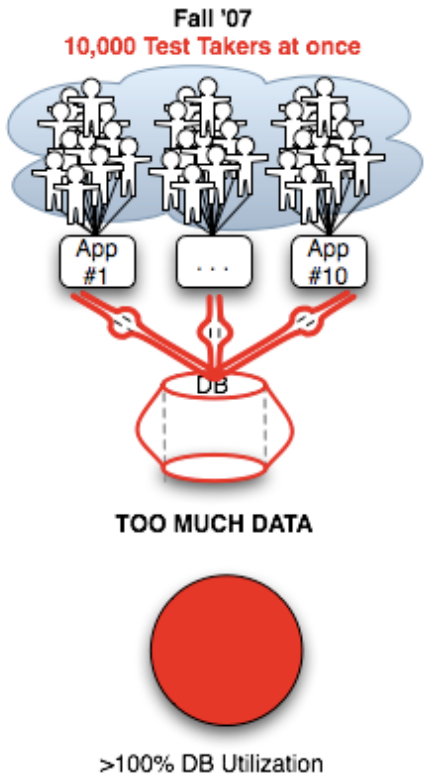
- With TC, our goal is to make this all visual (story time)

Case Study: Conversation Clustering

TERRACOTTA

Large Publisher Gets Caught Down the Path with Oracle

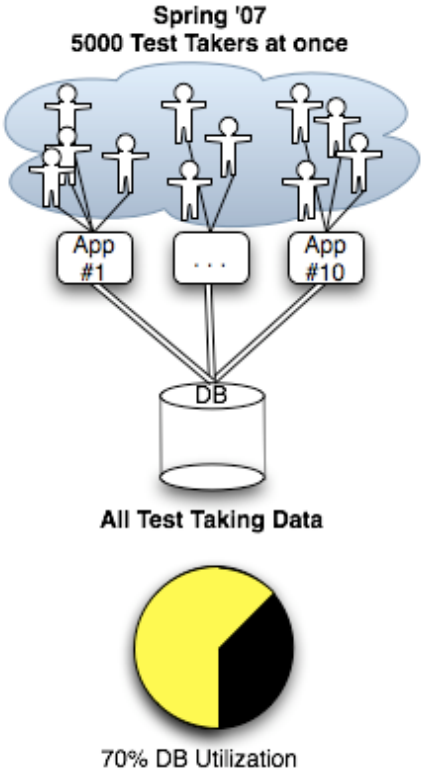


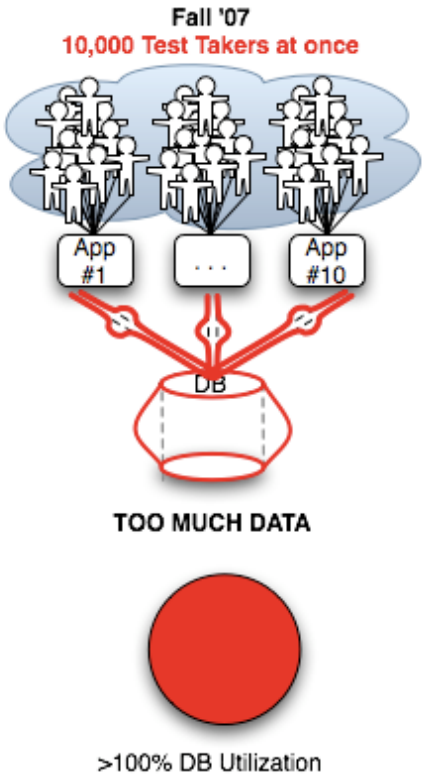


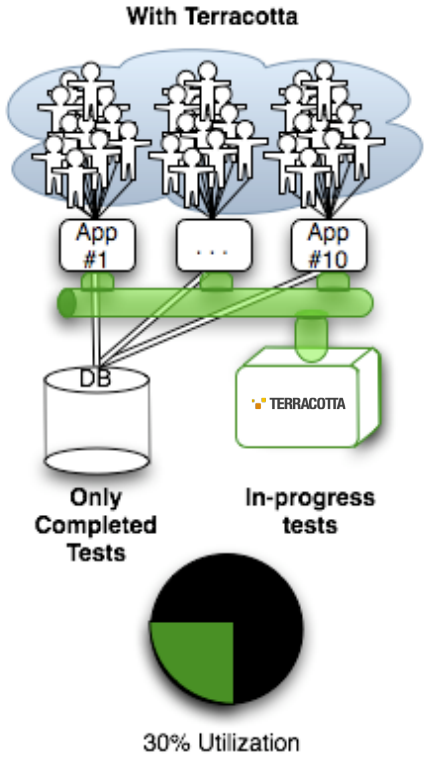
Scaling Out or Up?

Breaking the Pattern without Leaving "Load-Balanced" World

- $1.5 Million DB & HW savings
- Doubled business
- More than halved database load







Results

- Database was still the SoR which kept reporting and backup simple
- Scalability was increased by over 10X
- Availability was not compromised since test data was still on disk, but in memory-resident format instead of relational

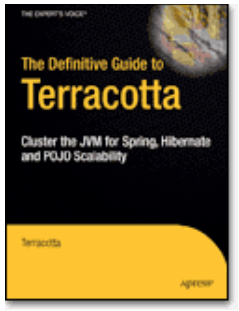
- …simple scalability + availability

Summary

- Simplicity, scalability, and availability can be friends
- Write normal Java code that works across JVMs
- Use clustered architecture patterns
  - Conversation clustering/persistence
  - Window onto large dataset
  - Change notification
  - Shared metadata
  - Clustered membership and data management
- Leave business data in the database, use durable NAM for application state data
- Centralized operational control: manage your application cluster like you do your database
- Terracotta is open source
  - Free to use through production
  - Enterprise subscription, training, and services available

Resources

- Open Source (MPL-based) JVM-level clustering: http://www.terracotta.org
- Apress / Amazon.com: “Definitive Guide to Terracotta”
  - By Alex Miller, Ari Zilka, Geert Bevin, Jonas Bonér, Orion Letizi, Taylor Gautier



- Forums: http://forums.terracotta.org/
- Enterprise Offerings: http://www.terracottatech.com/