

Distributed Computing Made Easy: How to Build a POJO-based Data Grid

Jonas Bonér
Terracotta, Inc.

jonas@terracottatech.com

Open Terracotta
Open Source JVM-level
Clustering

Goal of This Session

- Learn how JVM-level clustering and Open Terracotta works at a high level
- Learn how clustering at runtime provides a simpler environment for development without hindering scale-out
- Learn how to scale-out POJO-based applications using Master/Worker and Locality of Reference

Let's Start With A Demo

Since a picture says more than a thousand words

Shared JTable (spreadsheet)

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);
    }
}
```

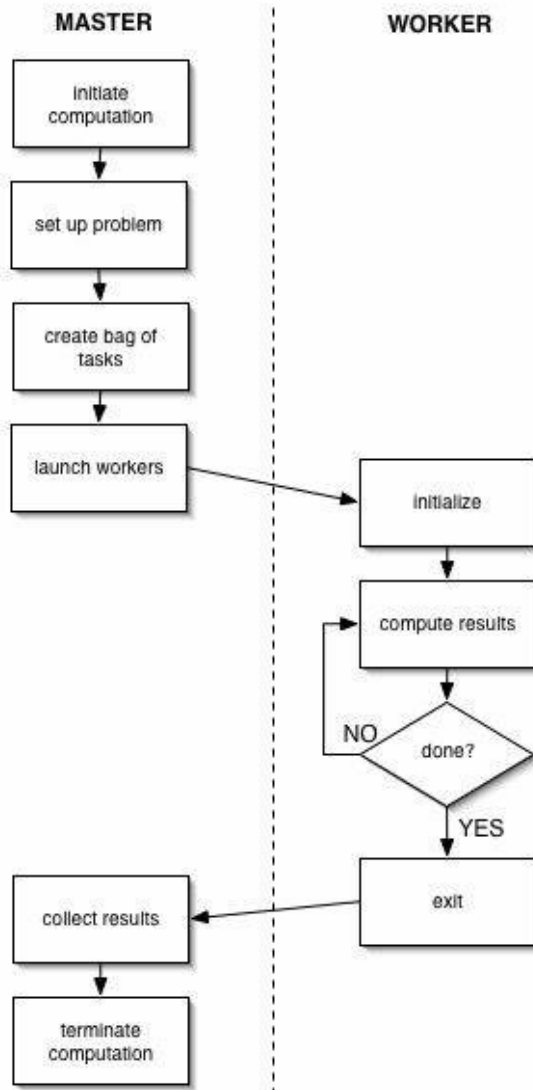
Magic is in Config File

```
<terraccotta-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>
</terraccotta-config>
```

Agenda

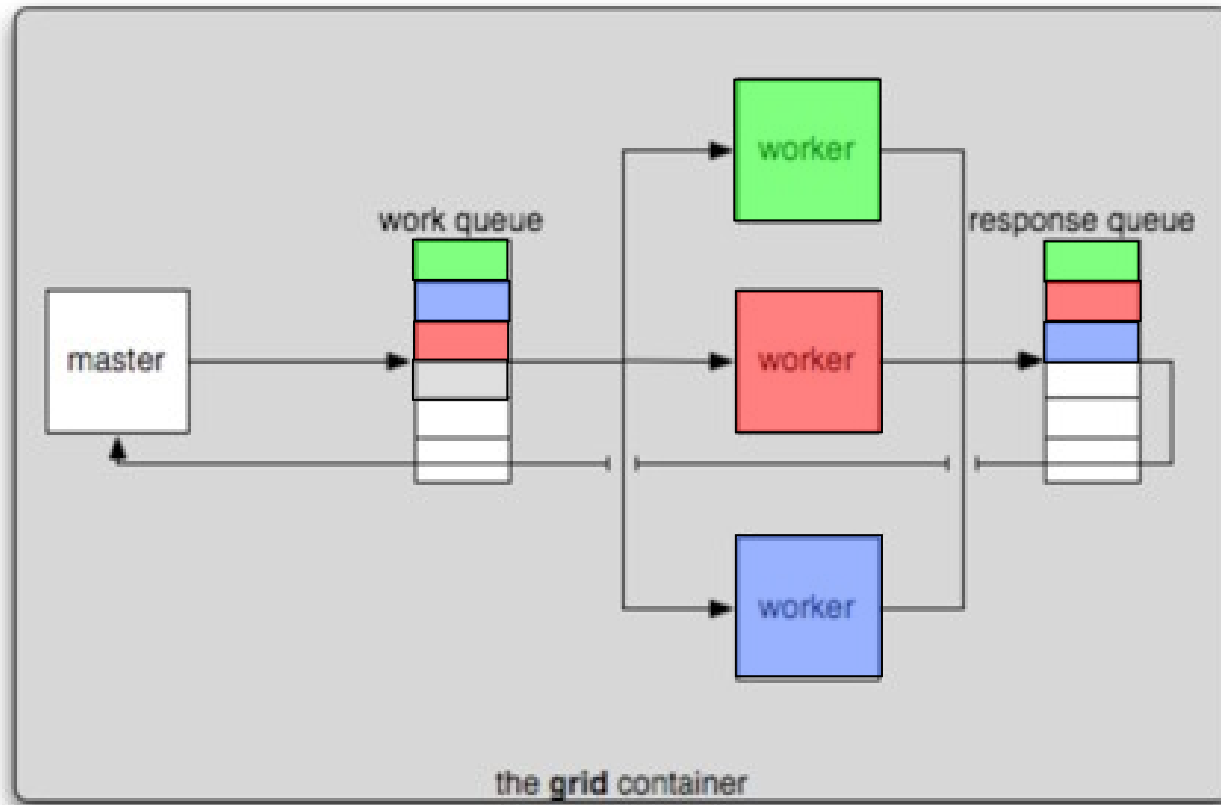
1. Master/Worker Pattern
2. Data Grids
3. JVM-level Clustering with Open Terracotta
4. Case-study – Distributed Web Spider:
 - § Master/Worker Container (POJO-based, single JVM)
 - § Web Spider Implementation
 - § Cluster It Using JVM-level Clustering
 - § Run It as a Data Grid
- § Real-World Challenges
- § Questions

Master/Worker Pattern



- 1 Master
- 1-N Workers
- 1 Shared Memory Space
- Common applications
 - Financial Risk Analysis and other Simulations
 - Searching / aggregation on large datasets
 - Sales Order pipeline processing

Simplified Architecture Overview



Possible Approaches in Java

- Threading primitives allows you to write your own implementation
 - Might be tricky to get right
 - Might be tricky to get good performance

- `java.util.concurrent.ExecutorService`
 - Since Java 1.5
 - Highly tuned
 - High-level abstractions
 - Direct support for Master/Worker pattern

- CommonJ WorkManager
 - IBM and BEA specification
 - Allows threading in JEE

Comparing the Options

- POJOs and multi-threaded programming work but do not abstract the implementation from day-to-day development
- Java 5's `ExecutorService` abstracts the implementation, but it **does not**
 - Separate *Master* from *Worker*
 - Provide information about *Work* status (Accepted, Rejected, failure cause etc.)
- The **CommonJ WorkManager** spec, however
 - Still simple POJO based
 - Can wrap Java 5 concurrency abstractions
 - Gives us the right abstraction level
 - Allows us to add a layer of reliability

Review the Goal

- What we want to do:
 - ~ Implement a POJO-based single-JVM *WorkManager*
 - ~ Run the *WorkManager* on multiple JVMs
 - ~ Ensure application performance by minimizing data movement payload across worker contexts
- If we pull this off...



POJO-based Data Grid

What is a Data Grid?

Here is my definition:

“A *Data Grid* is a **set of servers** that together creates a mainframe class processing service where **data and operations can move seamlessly** across the grid in order to **optimize the performance and scalability** of the computing tasks submitted to the grid.”

Grids: Master/Worker In a Box?

- **Grid solutions are focused on:**
 - “moving the application to the data”
 - “moving the operations instead of data”

- **Scaling and HA**
 - Scalability via Locality of Reference
 - High-Availability by data duplication

- **Off-the-shelf Grid solutions:**
 - API-based
 - Uses Java Serialization
 - Black box Master/Worker and workload routing

Scaling Data Grids

- Make use of Locality of Reference
- Data local to a specific node stays there
- Instead of moving all data to all nodes we can:
 - Move operations around instead of data
 - Move the application to the data
- Work partitioning
 - Ultimate: Work is “Embarrassingly Parallel”
 - Acceptable: Partition the work into logical groups working on the same data set
- Benefits
 - Low latency
 - Unlimited scalability

The Ideal Solution: **Simplicity and Scale-Out**

- **Simplicity** requires ...
 - No usage of proprietary APIs
 - Preservation of Object Identity - no serialization, works with POJOs
 - Preservation of the semantics of the JLS and JMM

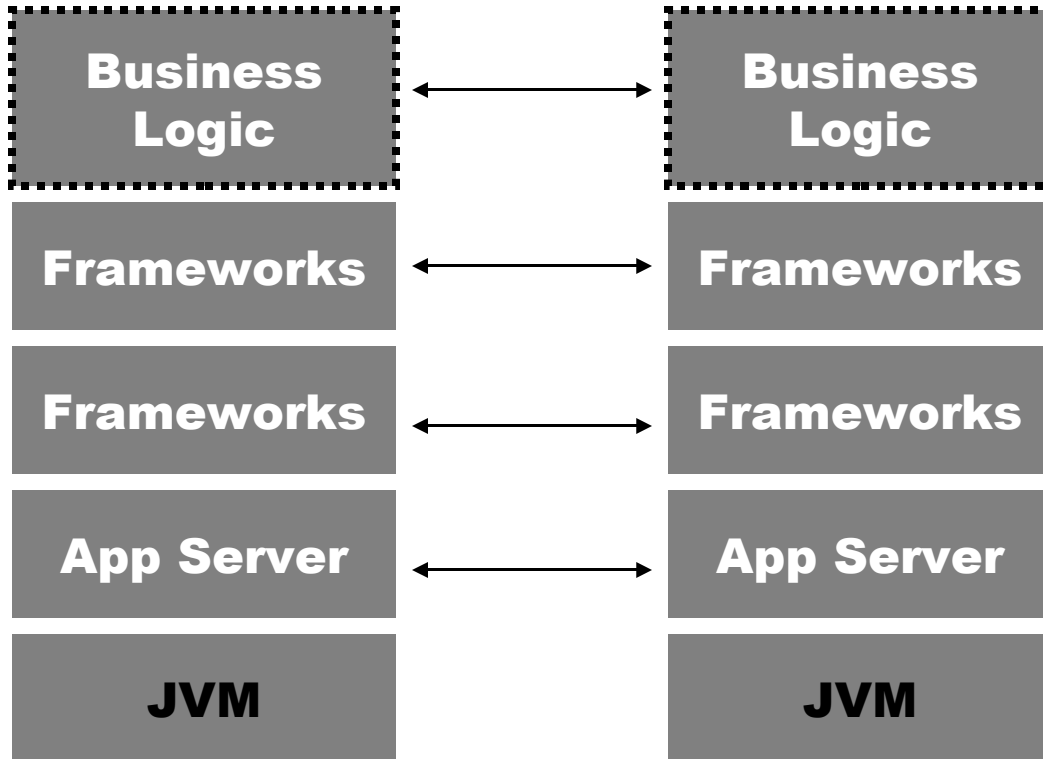
- **Scale-out** requires...
 - Fine-grained and lazy replication
 - Runtime lock optimization for clustering
 - Runtime caching for data access

The Ideal Solution: Cluster the JVM...not the app

Enter Open Terracotta

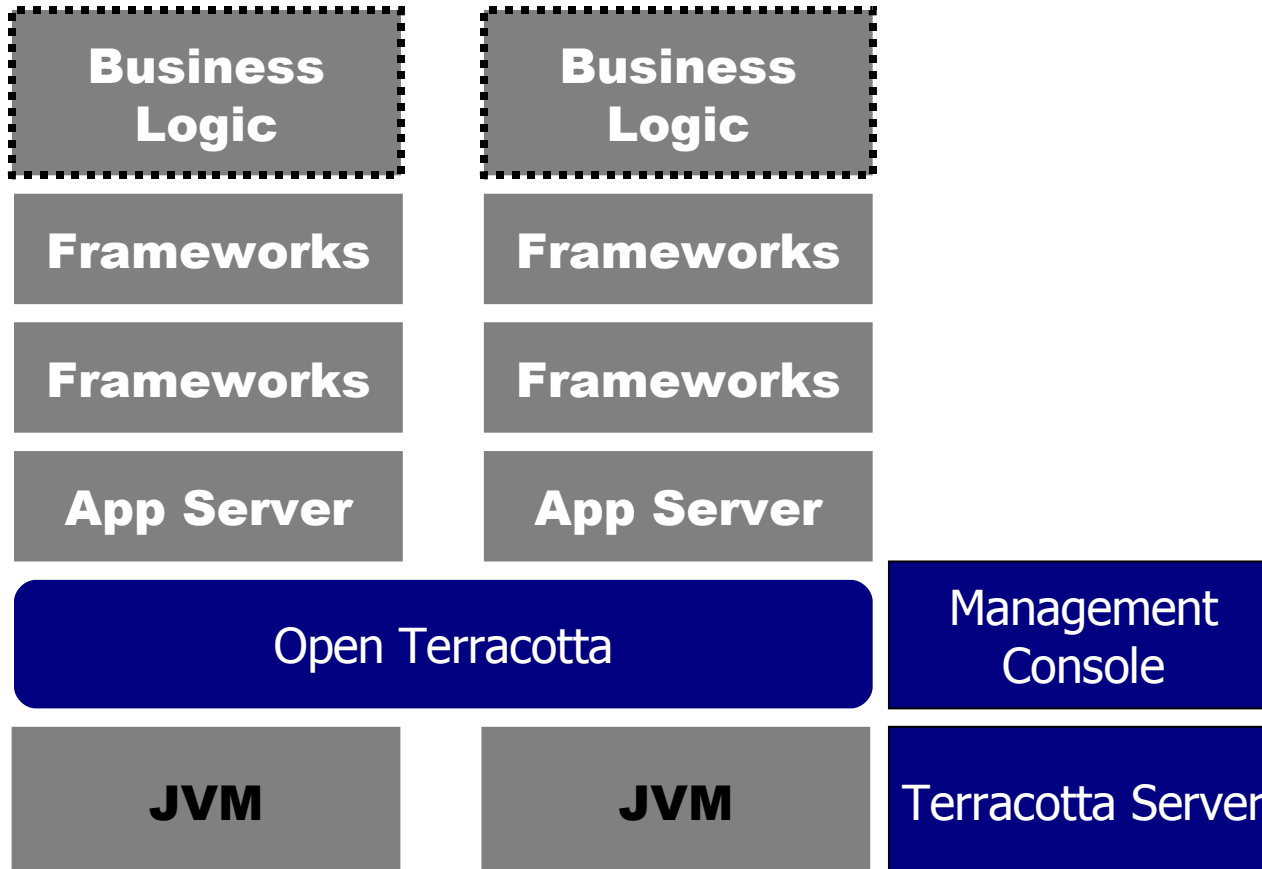
- Recognizes that state sharing is a deployment/operational artifact and delivers it as a runtime **infrastructure service**
- **Clusters the JVM** - shares any POJO and its references by:
 - **Plugging into the Java Memory Model** and automatically detects what changed in the “shared” domain model
 - Only replicating **what changed to where needed**
- Open Source under Mozilla-based license

Take Your Applications from this...

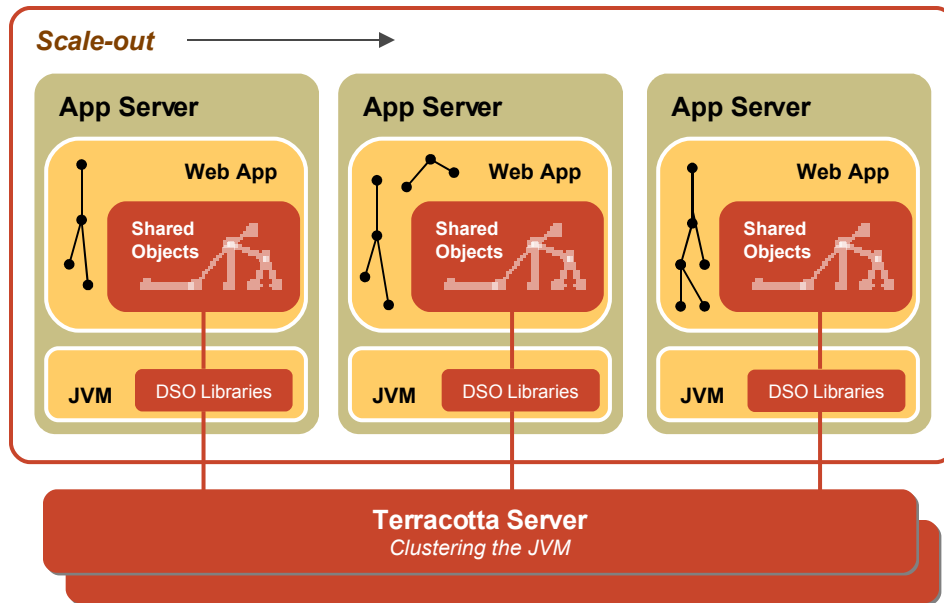


Clustered
App Servers,
JMS, RMI,
Multicast Etc.
Is Expensive

...To This



Terracotta Architecture: Cluster w/o JEE Resources



Generic Platform for
Distributed Computing

Capabilities

- **Heap Level Replication** - share almost any object graph - field-level replication
- **ACID Replication** - no new exceptions or error scenarios
- **Central storage** - keeps app state across restarts
- **Comm. Hub** - manage shared object comms w/o multicast or split brain
- **Virtual Memory** - page in objects on demand
- **Thread Coordination** - maintaining JMM (synchronized, wait, notify) across the cluster
- **Management Console** - runtime visibility, monitoring, data introspection

Cluster CommonJ WorkManager

- Clustering the JVM underneath *CommonJ WorkManager* delivers POJO-based Data Grid:
 - **Performance:**
 - » Locality of Reference + fine-grained replication
 - **Scale-Out:**
 - » Ability to scale Master and Workers independently
 - **High-Availability:**
 - » Terracotta Server + fail-over to any other node

Case Study

1. Implement a Master/Worker “container”
2. Implement a Web Crawler that uses our “container”
3. Cluster it with Terracotta
4. Run app
5. Look into how we can tackle some real-world challenges

CommonJ WorkManager Specification

```
public interface Work extends Runnable { }

    public interface WorkManager {
        WorkItem schedule(Work work);
        WorkItem schedule(Work work, WorkListener listener);
        boolean waitForAll(Collection workItems, long timeout);
        Collection waitForAny(Collection workItems, long timeout);
    }

    public interface WorkItem {
        Work getResult();
        int getStatus();
    }

    public interface WorkListener {
        void workAccepted(WorkEvent we);
        void workRejected(WorkEvent we);
        void workStarted(WorkEvent we);
        void workCompleted(WorkEvent we);
    }

    public interface WorkEvent {
        int WORK_ACCEPTED = 1;
        int WORK_REJECTED = 2;
        int WORK_STARTED = 3;
        int WORK_COMPLETED = 4;
        public int getType();
        public WorkItem getWorkItem();
        public WorkException getException();
    }
```

2. Implementing a Web Crawler

- **What is a Web Spider?**
 1. Grabs the page from a URL
 2. Does something with it – for example indexes it using *Lucene*
 3. Parses it and find all URLs from this page
 4. Grabs these pages
 5. Parses them and...so on...you get the idea

- **How to slice the problem?**
 1. Create new *Work* for a URL to a page to parse
 2. When executed, the *Work* grabs the page, parses it and gathers all its URLs
 3. For each new URL: GOTO 1.

- **Use the Master/Worker “container” to parallelize the work**

- **Let’s look at the code...**

3. Cluster with Terracotta

- Do not change the application
- Declaratively select which objects should be shared across the grid
- E.g. which part(s) of the Java heap that should be always **up-to-date and visible** to **all parts** of the application that needs it – in the whole grid

Terracotta Configuration

```
<dso>
```

```
<roots>
```

```
<root>
```

```
<field-name>
```

```
org.terracotta.commonj.workmanager.SingleWorkQueue.m_workQueue
```

```
</field-name>
```

```
</root>
```

```
</roots>
```

```
<instrumented-classes>
```

```
<include>
```

```
<class-expression>org.terracotta.commonj.*</class-expression>
```

```
</include>
```

```
<include>
```

```
<class-expression>org.terracotta.spider.*</class-expression>
```

```
</include>
```

```
</instrumented-classes>
```

```
</dso>
```

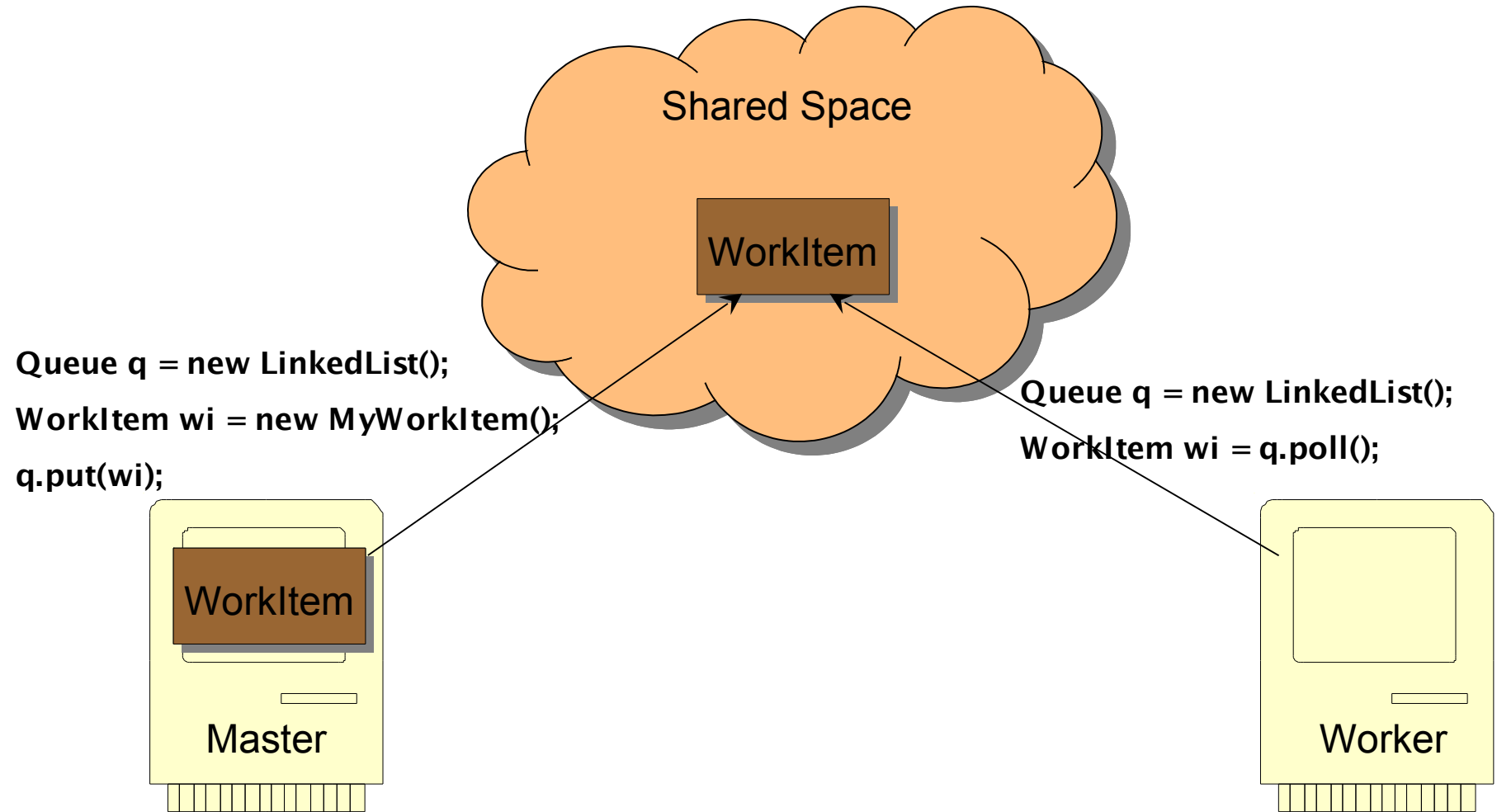
define *roots*



define *includes*



Master and Worker are operating on
the exact same but **still local** WorkItem instance



4. Run App

5. Challenges

- Very high volumes of data?
- How to handle work failure?
- Routing?
- Ordering matters?
- Worker failure?

Very High Volumes of Data?

- **Problem:** Bottlenecks on the single *Queue*
- **Solution:** Create one *Queue* per *Worker*
 - Load-balancing in the *Master*
 - Locality of Reference
 - Minimizes contention

Change::

```
public WorkerItem schedule(final Work work) {
    m_queue = new ConcurrentQueue<WorkerItem>(new DefaultQueue<WorkerItem>());
    return m_router.route(work),
}
m_queue.put(workItem);
return workItem;
}
```

Routing

- Keep state in the *Work* – no state in *Worker*
- Route *Work* that are working on the same data to the same node
- Work can repost itself or new work onto the *Queue* and is guaranteed to be routed to the same node

```
public class RoutableWorkItem<ID> extends  
    DefaultWorkItem implements Routable<ID> {  
    protected ID m_routingID;  
  
    ...  
}
```

Routing

```
public interface Router<ID> {  
    RutableWorkItem<ID> route(Work work);  
    RutableWorkItem<ID> route(Work work, WorkListener listener);  
    RutableWorkItem<ID> route(RutableWorkItem<ID> workItem);  
}
```

- Can use different load-balancing algorithms
 - Round-robin
 - Work load sensitive balancing (*Router* looks at *Queue* depth)
 - Data affinity - “Sticky routing”
 - Your own...

Retry

- Retry on failure
- Event-based failure reporting
- Use the `WorkListener`

```
public void WorkListener#workRejected(WorkEvent we);  
public void workRejected(WorkEvent we) {  
    Exception cause = we.getException();  
    WorkItem wi = we.getWorkItem();  
    Work work = wi.getResult();  
    ... // reroute the work onto queue X  
}
```

Ordering Matters?

- Use a `PriorityBlockingQueue<T>` (instead of a `LinkedBlockingQueue<T>`)

- Let your `Work` implement `Comparable`

- Create a custom `Comparator<T>`:

```
Comparator c = new Comparator<RoutableWorkItem<ID>>() {  
    public int compare(  
        RoutableWorkItem<ID> workItem1,  
        RoutableWorkItem<ID> workItem2) {  
        Comparable work1 =  
            (Comparable)workItem1.getResult();  
        Comparable work2 =  
            (Comparable)workItem2.getResult();  
        return work1.compareTo(work2);  
    }  
};
```

- Pass it into the constructor of the `PriorityBlockingQueue<T>`

Worker Failure Detection: Approaches

- Heartbeat mechanism
- Work timestamp – Master checks for timeout
- Worker holds an “is-alive-lock” that Master tries to take
- Notification from Terracotta Server (in Feb)

- If detected: reroute all non-completed work

Wrap Up: Developer Benefits

- Work with plain POJOs
- Event-driven development - does not require explicit threading and guarding
- Test on a single JVM, deploy on multiple JVMs
- White box impl: Freedom to design Master, Worker, routing algorithms, fail-over schemes etc. the way you need

Resources

- Checkout the source for the Open Data Grid:
 - <http://svn.terracotta.org/svn/forge/projects/labs/opendatagrid/>

- Download Open Terracotta today:
 - <http://terracotta.org>

- Articles:
 - <http://www.theserverside.com/tt/articles/article.tss?l=DistCompute>
 - <http://jonasboner.com/2006/09/14/how-to-implement-a-distributed-commonj-workmanager/>
 - <http://javathink.blogspot.com/2006/09/lets-go-distributed-how-to-build.html>

- Documentation and blogs:
 - <http://terracotta.org>
 - <http://blog.terracottatech.com/>
 - <http://jonasboner.com>

Questions?

Thank You

<http://terracotta.org>



Not Used

Historical Background

- *Tuple Space (TS)*
 - Implementation of *Associative Memory*
- *Linda* shared data model
 - Tuple: `t = ("tag", values ...)`
 - Template: `s = ("tag", values ..., formals ...)`
 - `in(s)`: blocking receive from *TS*
 - `out(s)`: non-blocking send to *TS*
 - `eval(s)`: forks a new process that evaluates `s`
- **Blackboard Systems**
 - *Shared Memory* with *Agents* accessing or modifying the Memory
- **Essentially Master/Worker**
 - Master spawns Workers (agents/processes)
 - Workers grabs Work from a Memory Space
 - Executes Work and then puts it back into the Memory Space

Examples of Traditional Solutions

- **Tuple Space implementations**
 - JavaSpaces (Jini) etc.

- **Native Language Support**
 - Orca, Occam, Erlang, Sisal, Oz, E, Parallel Haskell etc.

- **Messaging**
 - JMS, Tibco, MQSeries etc.

- **Language extensions**
 - OpenMP, MPI etc

Metaphor: Network Attached Memory

- Terracotta is *Network-Attached Memory (NAM)*
 - Similar to *Network-Attached Storage (NAS)*
- JVM-level replication makes *NAM's* presence transparent
 - Just like *NAS* can be transparent behind a file I/O API
- Getting *NAM* to perform is similar to any I/O platform
 - Read-ahead buffering
 - Read / write locking optimizations
 - Etc.

How To Distribute State and Operations?

- Sockets
 - Low level, complex, hard to maintain, not reusable etc.
- RMI (Remote Method Invocation)
 - Stub/skeleton compilation, intrusive, using Serializable, hard to scale
- Messaging
 - Verbose, intrusive, asynchronous, using Serializable, hard to scale
- Database
 - JDBC/ORM, intrusive, hard to scale
- Clustering and Data Grid APIs
 - Intrusive, using Serializable