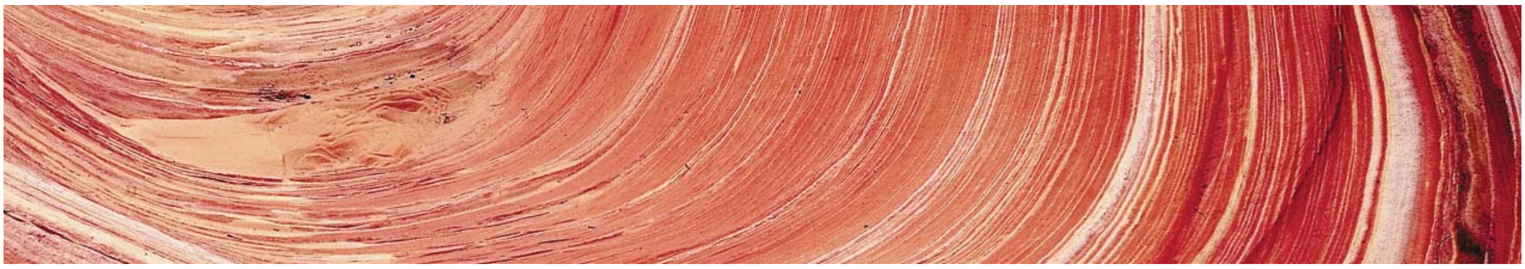
WWW.TERRACOTTATECH.COM
WWW.TERRACOTTA.ORG
INFO@TERRACOTTATECH.COM
415.738.4000

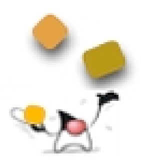


A Technical Introduction to Terracotta

OPEN SOURCE NETWORK-ATTACHED MEMORY FOR JVM HIGH-AVAILABILITY AND SCALABILITY

An Introduction to Terracotta DSO

Originally posted on InfoQ by Orion Letizi
http://www.infoq.com/articles/open-terracotta-intro



Java applications are easiest to write and test when they run in a single JVM. However, the requirements to make applications scalable and highly available have forced Java applications to run on more than one JVM. In this article, we introduce Terracotta, an enterprise-class, open-source JVM-level clustering solution.

JVM-level clustering simplifies enterprise Java by enabling applications to be deployed on multiple JVMs, yet interact with each other as if they were running on the same JVM.

In a single JVM, threads interact with each other through changes made to objects on the heap and through the language-level concurrency primitives: the 'synchronized' keyword and the Object methods wait, notify, and notifyAll. Terracotta allows threads in a cluster of JVMs to interact with each other across JVM boundaries using the same built-in JVM facilities extended to have a cluster-wide meaning. These clustering capabilities are injected into the bytecode of the application classes at runtime, so there is no need to code to a special clustering API.

Using these clustering facilities, Terracotta DSO (Distributed Shared Objects) is most commonly used in the following scenarios:

- HTTP Session Replication
- Distributed Cache
- POJO Clustering / Spring integration
- Collaboration, Coordination, and Events

A Simple Example

To illustrate what all this means in concrete terms, let's start with some sample code that you can sink your teeth into. The example domain is a retail system with a catalog of products that can be added to a customer's shopping cart. At any time, the set of active shopping carts can be viewed by, for example, an administration or reporting console.

The sample code is written using simple Java data structures. Some of the problem domain is idealized for simplicity. For example, the business data encapsulated in the in the product, catalog, customer, and order classes in a real system would likely be backed by a relational database, perhaps fronted by an object-relational system of some kind. The transient shopping cart data, though, might very well be best expressed purely as simple Java objects with no backing system of record.

The Product class contains data about a particular product: the product name, the SKU, and the price:

```java
package example;
import java.text.NumberFormat;

public class ProductImpl implements Product {
    private String name;
    private String sku;
    private double price;

    public ProductImpl(String sku, String name, double price) {
        this.sku = sku;
        this.name = name;
        this.price = price;
    }
    public String getName() {
        return this.name;
    }

    public String getSKU() {
        return this.sku;
    }

    public synchronized void increasePrice(double rate) {
        this.price += this.price * rate;
    }
}
```

Products are kept in a Catalog that maps the SKU of the product to product object. The Catalog can be used to display products and to look them up by SKU so they may be placed in a shopping cart. Here's the Catalog:

```
package example;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

public class Catalog {

    private final Map<String, Product> catalog;

    public Catalog() {
        this.catalog = new HashMap<String, Product>();
    }

    public Product getProductBySKU(String sku) {
        synchronized (this.catalog) {
            Product product = this.catalog.get(sku);
            if (product == null) {
                product = new NullProduct();
            }
            return product;
        }
    }

    public Iterator<Product> getProducts() {
        synchronized (this.catalog) {
            return new ArrayList<Product>(this.catalog.values()).iterator();
        }
    }

    public int getProductCount() {
        synchronized (this.catalog) {
            return this.catalog.size();
        }
    }

    public void putProduct(Product product) {
        synchronized (this.catalog) {
            this.catalog.put(product.getSKU(), product);
        }
    }

}
```

The ShoppingCart class contains a list of Products that a shopper has browsed and tentatively wants to purchase:

```
package example;
import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;
```

```
public class ShoppingCartImpl implements ShoppingCart {

    private List<Product> products = new LinkedList<Product>();

    public void addProduct(final Product product) {
        synchronized (products) {
            this.products.add(product);
        }
    }
}
```

There is a companion class to the ShoppingCart called ActiveShoppingCarts that does some bookkeeping about active shopping carts:

```
package example;
import java.util.LinkedList;
import java.util.List;

public class ActiveShoppingCarts {

    private final List<ShoppingCart> activeShoppingCarts = new LinkedList<ShoppingCart>();

    public void addShoppingCart(ShoppingCart cart) {
        synchronized (activeShoppingCarts) {
            this.activeShoppingCarts.add(cart);
        }
    }

    public List getActiveShoppingCarts() {
        synchronized (this.activeShoppingCarts) {
            List<ShoppingCart> carts
                = new LinkedList<ShoppingCart>(this.activeShoppingCarts);
            return carts;
        }
    }
}
```

For the purposes of encapsulation, there is a class called Roots that holds references to the roots of the clustered object graphs that are used by this application. It isn't a requirement to put the root fields in a special class, but it's convenient for this example:

```
package example;

import java.util.concurrent.CyclicBarrier;

public class Roots {
    private final CyclicBarrier barrier;
    private final Catalog catalog;
    private final ActiveShoppingCarts activeShoppingCarts;

    public Roots(CyclicBarrier barrier, Catalog catalog,
                 ActiveShoppingCarts activeShoppingCarts) {
        this.barrier = barrier;
        this.catalog = catalog;
        this.activeShoppingCarts = activeShoppingCarts;
    }

    public ActiveShoppingCarts getActiveShoppingCarts() {
        return activeShoppingCarts;
    }

    public CyclicBarrier getBarrier() {
        return barrier;
    }

    public Catalog getCatalog() {
        return catalog;
    }
}
```

The following code shows how these classes might be used in a multi-threaded environment. It assumes two threads enter the run() method and at various points they use a CyclicBarrier to coordinate with each other. This example usage pattern is utterly contrived, but you can imagine how this code might work in a real application:

```
package example;
import java.util.Iterator;
import java.util.concurrent.CyclicBarrier;

public class Main implements Runnable {
    private final CyclicBarrier barrier;
    private final int participants;
    private int arrival = -1;
    private Catalog catalog;
    private ShoppingCartFactory shoppingCartFactory;
    private ActiveShoppingCarts activeCarts;

    public Main(int participants, CyclicBarrier barrier, Catalog catalog,
                ActiveShoppingCarts activeCarts, ShoppingCartFactory shoppingCartFactory) {
        this.barrier = barrier;
        this.participants = participants;
        this.catalog = catalog;
        this.activeCarts = activeCarts;
        this.shoppingCartFactory = shoppingCartFactory;
    }

    public void run() {
        try {
            display("Step 1: Waiting for everyone to arrive.  I'm expecting " +
                    (participants - 1) + other thread(s)...");
            this.arrival = barrier.await();
            display("We're all here!");

            String skuToPurchase;
            String firstname, lastname;

            display();
            display("Step 2: Set Up");
            boolean firstThread = arrival == (participants - 1);

            if (firstThread) {
                display("I'm the first thread, so I'm going to populate the catalog...");
                Product razor = new ProductImpl("123", "14 blade super razor", 12);
                catalog.putProduct(razor);

                Product shavingCream = new ProductImpl("456", "Super-smooth shaving cream", 5);
                catalog.putProduct(shavingCream);

                // I'm going to be John Doe and I'm going to buy the razor
                skuToPurchase = "123";
                firstname = "John";
                lastname = "Doe";
            } else {
                // I'm going to be Jane Doe and I'm going to buy the shaving cream...
                skuToPurchase = "456";
                firstname = "Jane";
                lastname = "Doe";
            }

            // wait for all threads.
            barrier.await();

            display();
            display("Step 3: Let's do a little shopping...");
            ShoppingCart cart = shoppingCartFactory.newShoppingCart();

            Product product = catalog.getProductBySKU(skuToPurchase);
            display("I'm adding \"" + product + "\" to my cart...");
            cart.addProduct(product);
            barrier.await();
            display();
            display("Step 4: Let's look at all shopping carts in all JVMs...");
            displayShoppingCarts();

            display();
            if (firstThread) {
                display("Step 5: Let's make a 10% price increase...");
                for (Iterator<Product> i = catalog.getProducts(); i.hasNext();) {
                        Product p = i.next();
                        p.increasePrice(0.1d);
                }
            } else {
                display("Step 5: Let's wait for the other JVM to make a price change...");
            }
            barrier.await();

            display();
            display("Step 6: Let's look at the shopping carts with the new prices...");
            displayShoppingCarts();

        } catch (Exception e) {
            // You wouldn't really do this here.
            throw new RuntimeException(e);
        }
    }

    // ... setup and convenience code omitted

    public static void main(String[] args) throws Exception {
        int participants = 2;
        if (args.length > 0) {
            participants = Integer.parseInt(args[0]);
        }

        Roots roots = new Roots(new CyclicBarrier(participants), new Catalog(),
                                new ActiveShoppingCarts());

        if (args.length > 1 && "run-locally".equals(args[1])) {
            // Run 'participants' number of local threads. This is the non-clustered
            // case.
            for (int i = 0; i < participants; i++) {
                new Thread(new Main(participants, roots.getBarrier(), roots.getCatalog(),
                                    roots.getActiveShoppingCarts(),
                                    new ShoppingCartFactory(roots.getActiveShoppingCarts()))).start();
            }
        } else {
            // Run a single local thread. This is the clustered case. It is assumed that
            // main() will be called participants - 1 times in other JVMs
            new Main(participants, roots.getBarrier(), roots.getCatalog(),
                     roots.getActiveShoppingCarts(),
                     new ShoppingCartFactory(roots.getActiveShoppingCarts())).run();
        }
    }
}
```

So far, this code works fine in the context of a single JVM. Multiple threads interact with Catalog, Product, ShoppingCart, Customer, and Order objects as simple POJOs and can coordinate with each other, if need be, using standard Java library util.concurrent classes, namely CyclicBarrier.

If this were more than just a sample application, however, we would want to deploy it on at least two physical servers for high availability with the option to add additional servers for scalability as usage increases over time. Adding servers causes a number of requirements to emerge that don't exist in the single JVM deployment scenario:

- all active shopping carts should be available in all JVMs so a browsing customer's requests can be sent to any of the servers without losing the items in that customer's shopping cart.
- a view on all the active carts will require access to all active carts in every JVM.
- thread interaction expressed in the example code by using CyclicBarrier must be extended to threads in multiple JVMs.

- if the Catalog data becomes large enough, it might not fit comfortably in RAM. It could be retrieved as needed from the product database, but the database will be a bottleneck. If caching is used to alleviate the database bottleneck, each JVM will need access to that cache. To avoid critical spikes in database usage, the cache should be loaded once from the database and shared amongst the JVMs rather than loaded separately by each JVM.

All of the requirements introduced by deploying the application in a cluster can be met by using Terracotta with a small amount of configuration and no code changes. Let's take a quick look at what the configuration looks like to make this happen.

Configuration for the Example

The first configuration step is to determine which objects in the application should be shared by the cluster. These shared object graphs are specified by declaring specific variables to be "roots." Every object reachable by reference from a root object becomes a shared object available to all JVMs, cluster-wide. In our example so far, we have three roots, all of them declared in the Roots class. This is specified in the Terracotta config like so:

```
<roots>
  <root>
    <field-name>example.Roots.barrier</field-name>
  </root>
  <root>
    <field-name>example.Roots.catalog</field-name>
  </root>
  <root>
    <field-name>example.Roots.activeShoppingCarts</field-name>
  </root>
</roots>
```

The next configuration step is to determine which classes should have their bytecode instrumented at load-time. The class of any object that is to become part of a shared object graph must have its bytecode instrumented by Terracotta when the class is loaded. This instrumentation process is how the transparent clustering capabilities of Terracotta are injected into the application. For this example, all we have to do is include everything in the example.* package. The CyclicBarrier class is automatically included by Terracotta because it is part of a core set of Java library classes that are required to be instrumented. The instrumented-classes configuration section looks like this:

```
<instrumented-classes>
  <include>
    <!--include all classes in the example package for bytecode
        instrumentation-->
    <class-expression>example..*</class-expression>
  </include>
</instrumented-classes>
```

The final configuration step is to determine which methods should have cluster-aware concurrency semantics injected into them. For the purposes of this example, we will use a regular expression that encompasses all methods in all included classes for "autolocking:"

```
<locks>
  <autolock>
    <method-expression>void *..*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
</locks>
```

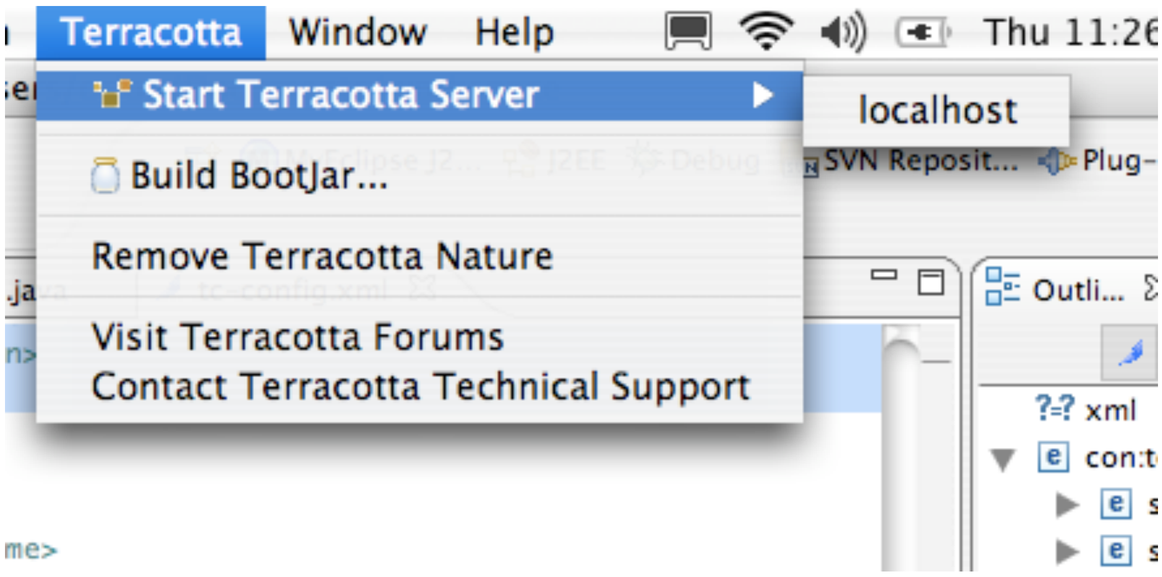
This configuration will instruct Terracotta to find all synchronized methods and blocks and all calls to wait() and notify() in the methods of every class that is instrumented and augment them to have a cluster-wide meaning.

Running the Example

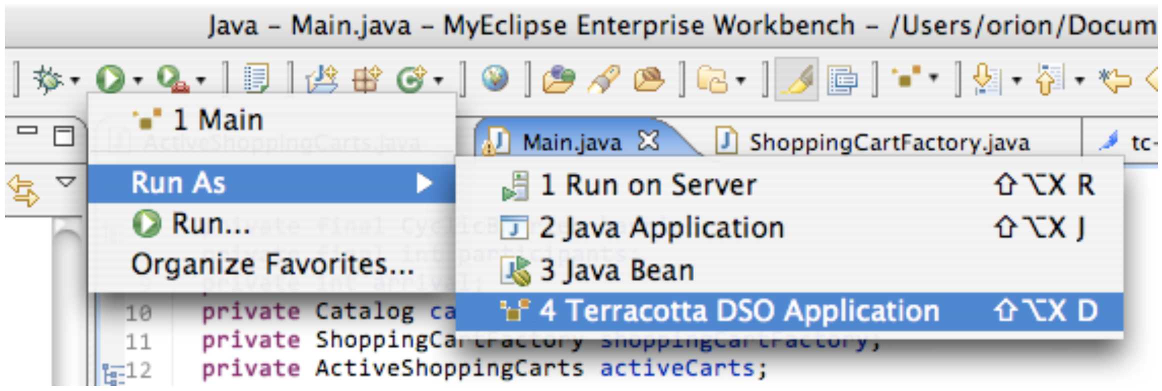
You can get the source code in this example at:

http://wiki.terracotta.org/confluence/display/labs/CatalogExample

Once the configuration is complete, running the application in a cluster can be done from the command line or from the Eclipse plugin. In this article, we'll use Eclipse, but there are multiple examples of starting Terracotta from the command line in the online tutorials and in the download kit. The first step is to start the Terracotta server. In Eclipse, this can be done from the Terracotta menu.



Once the server is started, the instances of your application can be started. For this example, we must run the Main class twice:



When the first application instance is started, you should see something like this in the output console:

```
2007-01-18 15:49:42,204 INFO - Terracotta, version 2.2 as of 20061201-071248.
2007-01-18 15:49:42,811 INFO - Configuration loaded from the file at
'/Users/orion/Documents/workspace/TerracottaExample/tc-config.xml'.
2007-01-18 15:49:42,837 INFO - Log file:
'/Users/orion/Documents/workspace/TerracottaExample/terracotta/client-logs/terracotta-client.log'.
Waiting for everyone to arrive.  I'm expecting 1 other thread(s)...
```

This indicates that the main thread in the first instance is blocking at barrier.await(). Since the barrier is a shared object, the main thread will block until another thread in either this JVM or another JVM in this Terracotta cluster calls barrier.await(). Then, both threads will be allowed to proceed. This will happen if we start up another application instance. Once the second application instance is started, you should see something like the following output in the consoles (you might see slightly different output depending on the order in which the threads reach the various barrier points):

```
Step 1: Waiting for everyone to arrive.  I'm expecting 1 other thread(s)...
We're all here!

Step 2: Set Up
I'm the first thread, so I'm going to populate the catalog...

Step 3: Let's do a little shopping...
I'm adding "Price: $12.00; Name: 14 blade super razor" to my cart...

Step 4: Let's look at all shopping carts in all JVMs...
==========================
Shopping Cart
    item 1: Price: $12.00; Name: 14 blade super razor
==========================
Shopping Cart
    item 1: Price: $5.00; Name: Super-smooth shaving cream


Step 5: Let's make a 10% price increase...

Step 6: Let's look at the shopping carts with the new prices...
==========================
Shopping Cart
    item 1: Price: $13.20; Name: 14 blade super razor
==========================
Shopping Cart
    item 1: Price: $5.50; Name: Super-smooth shaving cream
```

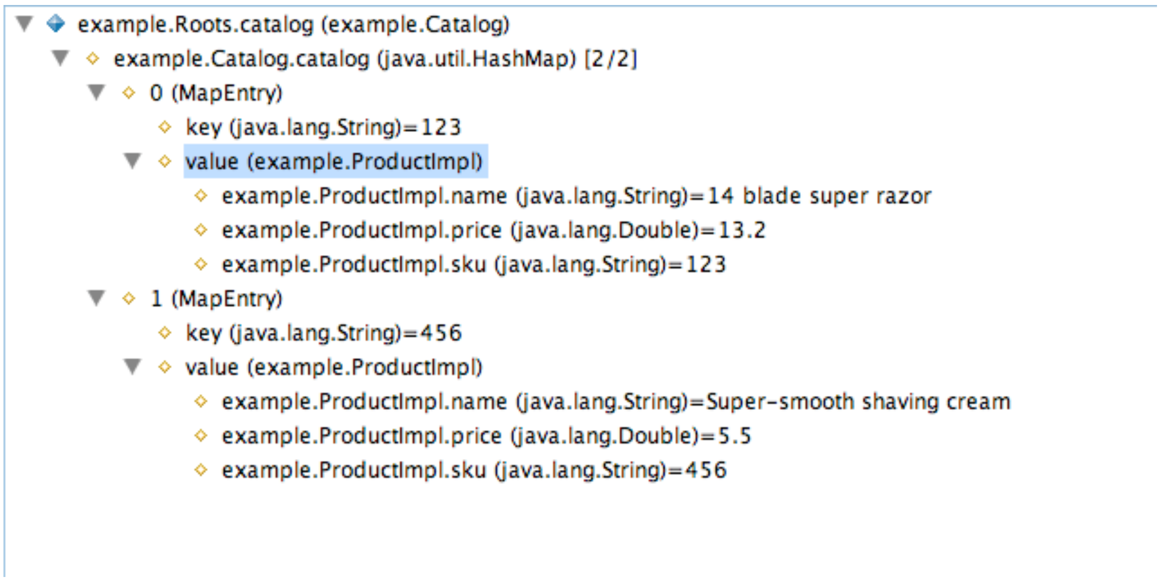
In the console output of the other application instance, you should see something like this:

```
Step 1: Waiting for everyone to arrive.  I'm expecting 1 other thread(s)...
We're all here!

Step 2: Set Up

Step 3: Let's do a little shopping...
I'm adding "Price: $5.00; Name: Super-smooth shaving cream" to my cart...

Step 4: Let's look at all shopping carts in all JVMs...
==========================
Shopping Cart
    item 1: Price: $12.00; Name: 14 blade super razor
==========================
Shopping Cart
    item 1: Price: $5.00; Name: Super-smooth shaving cream


Step 5: Let's wait for the other JVM to make a price change...

Step 6: Let's look at the shopping carts with the new prices...
==========================
Shopping Cart
    item 1: Price: $13.20; Name: 14 blade super razor
==========================
Shopping Cart
    item 1: Price: $5.50; Name: Super-smooth shaving cream
```

In Step 1, the two threads in the different application instances were waiting for each other to start and arrive at the same rendezvous point. In Step 2, the first thread created Product objects and added them to the clustered Catalog. In Step 3, each thread pulled a Product object out of the clustered Catalog by SKU—notice that the products added by the first thread in one JVM are automatically available in the

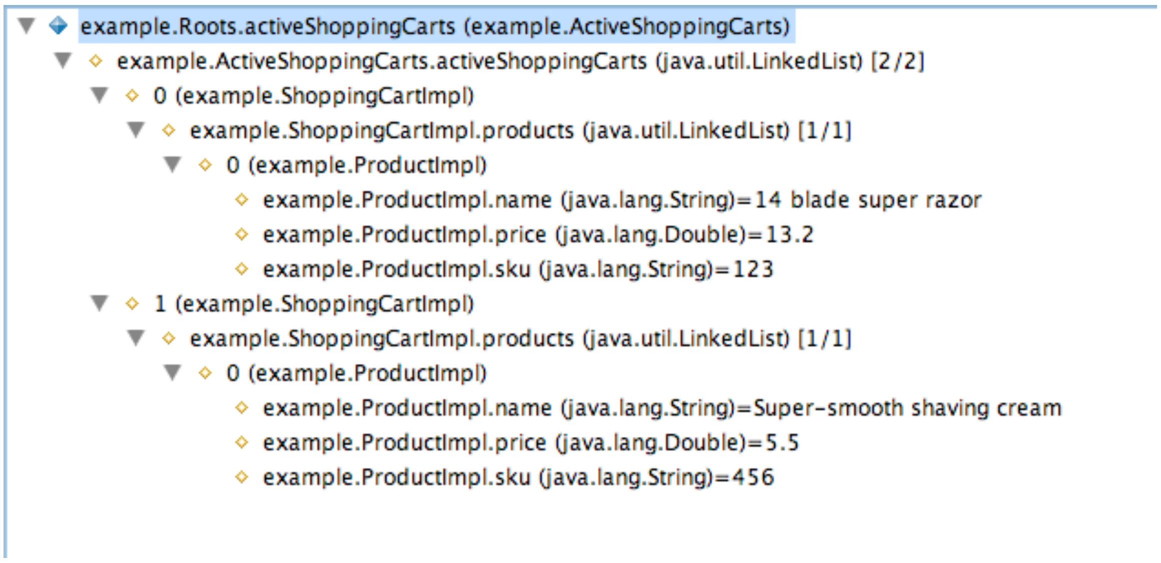
Catalog in the other JVM. In Step 4, both threads iterate over all of the active shopping carts in all JVMs and print the contents. In Step 5, the first thread made a 10% price increase by iterating over all of the Products in the catalog while the second thread blocks. In Step 6, both threads display all of the active shopping carts—notice that the price increase was done in one thread by manipulating the Catalog and the new prices were automatically reflected in all of the Shopping Carts in both JVMs.

All of the clustered objects can be viewed in real-time in the Terracotta administration console. Each root can be viewed as a tree of primitives and references. Here is a view onto the Catalog:



```
▼ ◆ example.Roots.catalog (example.Catalog)
    ▼ ◇ example.Catalog.catalog (java.util.HashMap) [2/2]
        ▼ ◇ 0 (MapEntry)
                ◇ key (java.lang.String)=123
            ▼ ◇ value (example.ProductImpl)
                    ◇ example.ProductImpl.name (java.lang.String)=14 blade super razor
                    ◇ example.ProductImpl.price (java.lang.Double)=13.2
                    ◇ example.ProductImpl.sku (java.lang.String)=123
        ▼ ◇ 1 (MapEntry)
                ◇ key (java.lang.String)=456
            ▼ ◇ value (example.ProductImpl)
                    ◇ example.ProductImpl.name (java.lang.String)=Super-smooth shaving cream
                    ◇ example.ProductImpl.price (java.lang.Double)=5.5
                    ◇ example.ProductImpl.sku (java.lang.String)=456
```

You can see the Product objects inside the 'catalog' HashMap. You can see those same Product objects referenced by the Shopping Carts as well:



```
▼ ◆ example.Roots.activeShoppingCarts (example.ActiveShoppingCarts)
    ▼ ◇ example.ActiveShoppingCarts.activeShoppingCarts (java.util.LinkedList) [2/2]
        ▼ ◇ 0 (example.ShoppingCartImpl)
            ▼ ◇ example.ShoppingCartImpl.products (java.util.LinkedList) [1/1]
                ▼ ◇ 0 (example.ProductImpl)
                        ◇ example.ProductImpl.name (java.lang.String)=14 blade super razor
                        ◇ example.ProductImpl.price (java.lang.Double)=13.2
                        ◇ example.ProductImpl.sku (java.lang.String)=123
        ▼ ◇ 1 (example.ShoppingCartImpl)
            ▼ ◇ example.ShoppingCartImpl.products (java.util.LinkedList) [1/1]
                ▼ ◇ 0 (example.ProductImpl)
                        ◇ example.ProductImpl.name (java.lang.String)=Super-smooth shaving cream
                        ◇ example.ProductImpl.price (java.lang.Double)=5.5
                        ◇ example.ProductImpl.sku (java.lang.String)=456
```

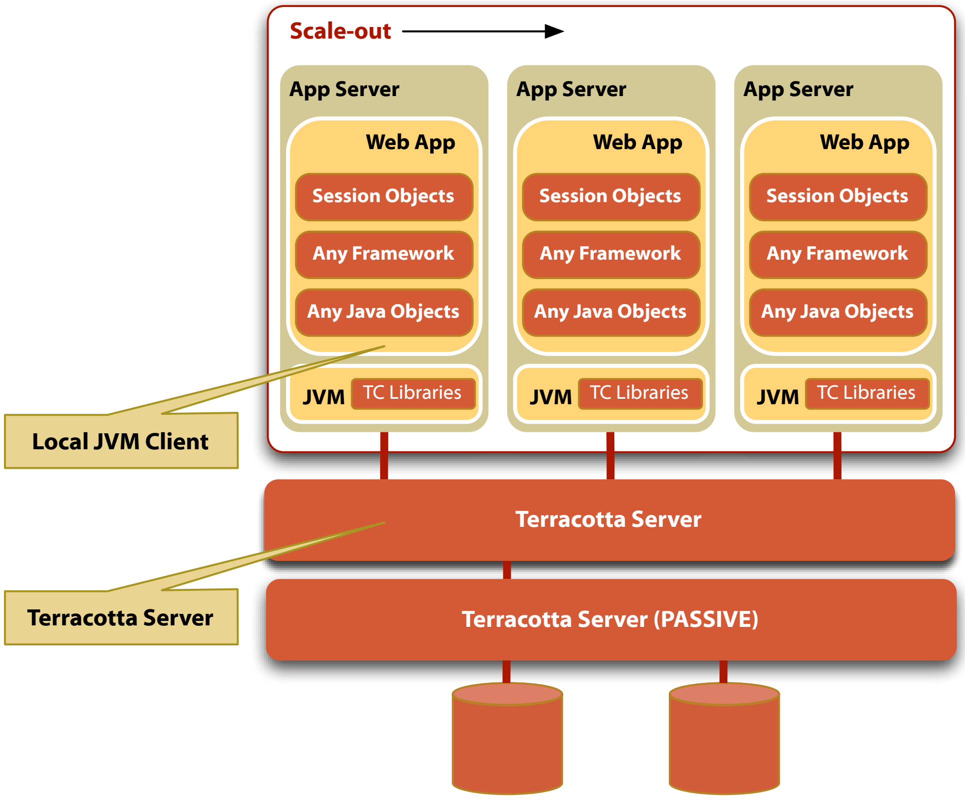
This view shows the Products referenced by the LinkedList inside the ShoppingCart.

How Does Terracotta Work?

Now that we've seen Terracotta's transparent clustering in action, let's discuss the particulars of how it works.

Architecture

Terracotta uses a client/server architecture where the JVMs running the clustered application connect to a central Terracotta server at startup. The Terracotta server stores object data and coordinates thread concurrency between JVMs. The Terracotta DSO libraries inside the application JVMs handle the bytecode instrumentation at class load time and transfers object data and the lock and unlock requests at synchronization boundaries, as well as the wait() and notify() requests between the application JVM and the Terracotta server at runtime.



Cluster Injection and Bytecode Instrumentation

Clustering behavior is injected into an application by instrumenting the bytecode of the application classes as they are loaded into the JVM. Terracotta uses bytecode injection techniques similar to those found in many Aspect-Oriented Programming frameworks such as AspectJ and AspectWerkz. The bytecode of a class is intercepted at load time and examined by the Terracotta transparency libraries. The bytecode is then modified according to the configuration before being passed on to the JVM to be blessed as a class.

To maintain object changes, the PUTFIELD and GETFIELD bytecode instructions are overloaded. PUTFIELD instructions are trapped to record changes to a clustered object's fields. GETFIELD bytecode instructions are trapped to retrieve object data from the server as needed, if it hasn't already retrieved the object referenced by the field in question from the server and instantiated it on the heap.

To manage thread coordination, the MONITORENTER and MONITOREXIT bytecode instructions are overloaded as well as the INVOKEVIRTUAL instructions for the various Object.wait() and Object.notify() methods. MONITORENTER signifies the request of a thread for an object's monitor. A thread will block at this instruction until it is granted the lock on that object. Once the lock is granted, that thread holds an exclusive lock on that object until the MONITOREXIT instruction is executed for that object. If the object in question happens to be a clustered object, Terracotta ensures that, in addition to requesting the local lock on that object, the thread also blocks until it receives the exclusive cluster-wide lock on that object. When the thread releases the local lock, it releases the corresponding cluster-wide lock.

In the example code, all of the synchronized methods and synchronized blocks in the example.* package are configured for "autolocking," meaning that the MONITORENTER and MONITOREXIT instructions are augmented. It is also possible to declare a method as a locked method in the Terracotta configuration in order to add clustering to application code that doesn't already have explicit synchronization.

The call sites of the wait() and notify() methods are also instrumented. When a wait() method is called on a shared object, the Terracotta server adds the calling thread to the set of threads in the entire cluster waiting on that object. When one of the notify() methods is called, the server ensures that the appropriate number of threads across the cluster are notified. When notify() is called on one JVM, the server chooses a waiting thread—if any--and causes that thread to be notified. When notifyAll() is called, the server causes all threads waiting on that object on all JVMs to be notified.

Roots and Clustered Object Graphs

Clustered objects start at the root of a shared object graph. The root is identified by a particular set of one or more fields and declared in the Terracotta configuration with a unique name. In the example, all of the roots are declared in the Roots class. This was a matter of convenience—any field may be declared a root.

When a root is first instantiated, the root object and all of the objects reachable by reference from the top-level root object become clustered objects. Their field data is shipped down to and stored by the Terracotta server. Once a root has been created in any JVM, all other assignments to that field are ignored and the value of the clustered root object is assigned to that field instead. This happens in the example when the second application instance creates the Roots object. The assignments to the root fields made in the Roots constructor are ignored because those roots have already been created by the first application instance. Instead of assigning the root fields the value of the arguments passed in on the constructor as the source code says, the Terracotta transparency libraries retrieve the root from the server, instantiate it on the local heap, and assign a reference to it to the root variable in question. This represents the only major change to the application semantics effected by Terracotta's transparency mechanism.

When a non-clustered object becomes referenceable from a clustered object, the new object and the entire graph of objects reachable by reference by the new object becomes clustered. Once an object becomes clustered, it is assigned a cluster-wide unique object id and remains clustered for the remainder of its lifecycle. When an object becomes unreachable by any root graph and there are no instances of it on any clustered JVMs, then it is eligible for removal by the cluster garbage collector in the server.

Synchronization, Clustered Locks, and Object Changes

Synchronized methods, synchronized blocks, and methods declared to be locked in the Terracotta configuration present the boundaries of a Terracotta transaction. The notion of a Terracotta transaction is somewhat different than a JTA transaction. It is much more similar to the transactions used in the Java memory model.

As discussed previously, a MONITORENTER instruction on a shared object is augmented to also be a clustered lock request such that the calling thread will block until it is granted both the local lock and the clustered lock for that object. All object changes made between a MONITORENTER and a corresponding MONITOREXIT are collected by Terracotta in a local transaction record. Terracotta guarantees that all changes made in all transactions associated with a particular object's lock in all clustered JVMs will be applied locally before a thread is allowed to proceed past the MONITORENTER instruction. Transactions may contain changes to any object, not just the object whose lock is associated with that transaction.

Fine-Grained Change Replication

The transactions that contain changes to objects contain only the data of the fields that have changed. These transactions are sent to the server and to the other clustered JVMs to keep the cluster consistent. The server only sends the transaction to the other JVMs that have objects instantiated on the heap that are represented in the transaction. Likewise, it only sends the portion of the transaction to those JVMs that must be applied. For example, if a thread makes changes to field 'p' in object 'a' and field 'q' in object 'b', then only the field data for a.p and the field data for b.p are put into the transaction and sent to the server. The server determines which other JVMs have instantiations of a or b. If another JVM has an instance of object a but not object b, then it receives the field data for a.p, but not for b.q.

In our example code, when the price was updated on the Product objects, only the price field was shipped to the cluster, not the name field or the SKU field which didn't change.

Object Identity and Serialization

Because object changes are tracked at the field-level and transactions contain object fragments rather than whole object graphs, Terracotta doesn't use Java serialization to replicate object changes. In the example, when we increased the prices on the Product objects, the only thing we had to ship around the cluster was the object ids of the objects that changed, an identifier of the field that had changed on that object, and the bytes containing the price field. The rest of the Product object is ignored. Using serialization would have serialized every field of the Product object and, had the Product object been a deep graph with lots of references to other objects which in turn had references to other objects, Java serialization would have serialized the entire deep graph—all for a change to a double value.

This approach is much more efficient than replication by serialization because it only moves the data that has changed across the cluster instead of entire serialized object graphs. But, aside from efficiency, there is a critical architectural benefit to using object fields as the units of change: preservation of object identity.

If Java serialization were used to move changes around the cluster, a changed object would be deserialized in the JVM of a clustered application and would somehow have to replace the existing object instance. This is why many other clustering and caching technologies require a GET/PUT API where a clustered object must be retrieved from the cluster using some sort of "GET" call and, when the changes made to that object, it must be put back into the cluster by means of some "PUT" call.

Terracotta has no such restrictions. A clustered object lives on the heap just like every other object. When changes are made locally to that object, they are made to the object on the heap. When changes are made remotely to that object, the transaction is received by the local JVM and applied directly to the existing object that's already on the heap. This means that there is one and only one instance of a clustered object on the heap at a given time (this picture gets slightly more complicated when multiple classloaders come into play, but that's beyond the scope of this article).

With Terracotta, you don't have to remember to get a fresh copy of an object and you don't have to remember to put it back when you're done with it. And, because there are no copies—a clustered object is just a plain object on the heap—clustered object behave just like any other object: any changes you make to a clustered object are available to every other object that has a reference to the changed object. Likewise, for a reference 'foo' to object 'bar' and a reference 'baz' to the same object 'bar', then foo == baz is true, not just foo.equals(baz).

In our example, you can see the preservation of object identity at work by the fact that, when changing the prices to the Product objects found in the catalog, the prices to the Product objects found in the shopping cart were also changed. That is because the Product objects in the Catalog are the same objects on the heap as the Products in the shopping cart.

This preservation of object identity makes clustered, multi-JVM applications behave much more like regular, single-JVM applications. This simplicity and the power of object identity preservation across the cluster allows for the problems of clustering to be made orthogonal to the problems of designing and implementing an application. The clustering behavior is pushed down into the Terracotta layer at the JVM-level, melting away into the infrastructure. Much like garbage collection allowed memory management to disappear from application code, Terracotta allows clustering and distributed computing behaviors to disappear as well.

Virtual Heap/Network-Attached Memory

Besides the ability to share objects and signal threads between JVMs, Terracotta also allows for efficient use of the local JVM heap for very large object graphs. As a shared object graph grows, it may not fit comfortably in the heap of a single JVM. Terracotta handles this by pruning the local instance of a shared object graph according to usage patterns on that instance. Terracotta keeps a configurable window on the clustered object graph such that the pieces that don't fit within a certain percentage of heap will be flushed out according to a cache policy. As those missing pieces are needed, they are automatically faulted into the JVM from the server. You can think of the Terracotta cluster as an arbitrarily large virtual heap or as network-attached memory.

This feature allows arbitrarily large object graphs to fit into standard heap sizes. It also allows for flexible, run-time data partitioning. In our example code, you can imagine what would happen if the Catalog became very large with hundreds of thousands of products and, perhaps, gigabytes of product data. To populate such a huge catalog might take minutes or hours. Without Terracotta, getting it all to fit in the heap of a single JVMs might require a 64-bit OS with more than 4GB of RAM. And, to get high-availability, you'll need at least two such application server machines. To get scalability, you'll need to add many such application server machines. Each application server machine added without Terracotta will require the Catalog to be loaded independently.

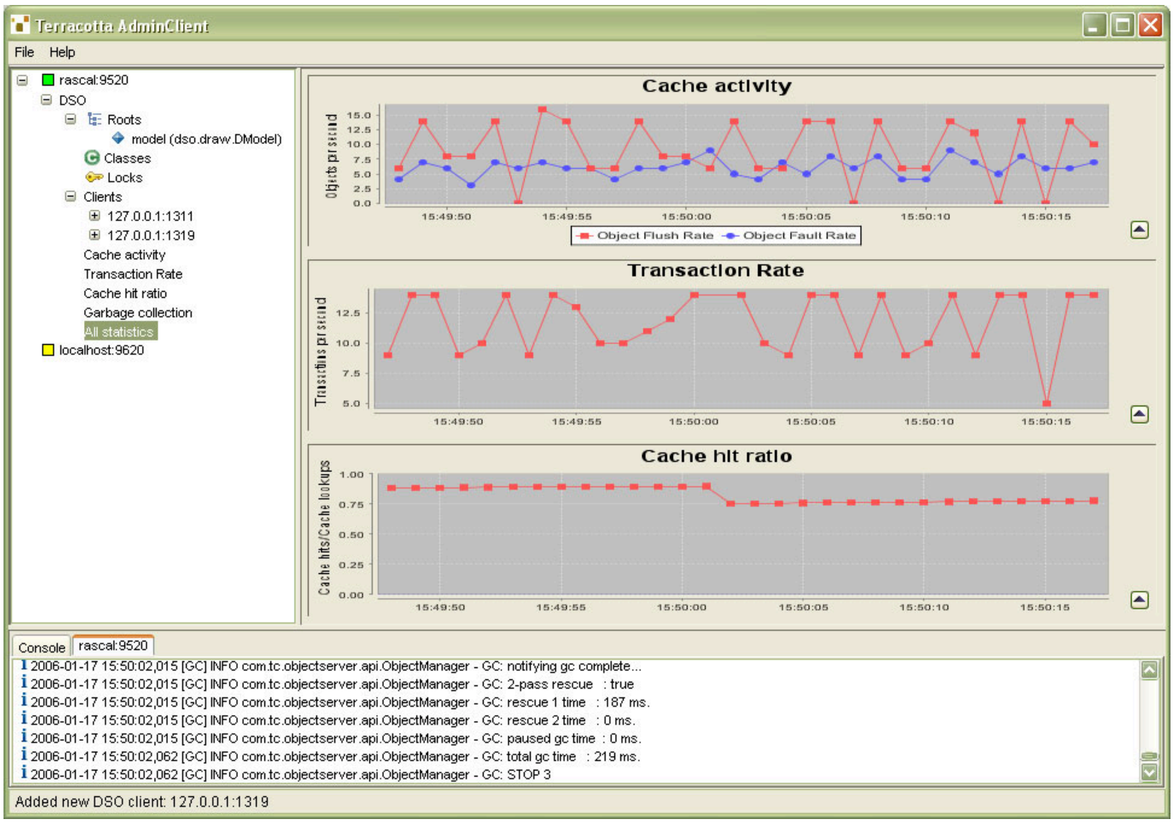
Because Terracotta acts as network-attached memory, you can fit the entire Catalog, no matter how large it gets, into a single clustered object graph. The Catalog need only be populated once, dramatically reducing startup time for additional application instances, but instantly available to every member of the cluster.

Monitoring and Control

The Terracotta server exposes information about the cluster via the standard remote bean property service JMX. This feature gives developers and operators the ability to monitor the cluster at runtime. For developers, this service can be used to inspect the data that resides in the cluster. A common question developers of Web Applications would like to answer is what exactly is being stored in the HTTP Session object. For operators, cluster membership, transaction levels, "faulting" and "flushing" of objects can all be monitored.

Because the monitoring services are exposed via JMX, any standard monitoring tool can be used such as HP OpenView, Hyperic HQ, or even the Java jconsole. Terracotta also ships with a specialized Administration Console designed to connect to the Terracotta server and provide useful statistics. To run the tool, simply run "admin.sh" or "admin.bat" from the bin directory of the Terracotta installation.

The following screenshot shows some of the capabilities of the Terracotta server:



```
2006-01-17 15:50:02,015 [GC] INFO com.tc.objectserver.api.ObjectManager - GC: notifying gc complete...
2006-01-17 15:50:02,015 [GC] INFO com.tc.objectserver.api.ObjectManager - GC: 2-pass rescue   : true
2006-01-17 15:50:02,015 [GC] INFO com.tc.objectserver.api.ObjectManager - GC: rescue 1 time   : 187 ms.
2006-01-17 15:50:02,015 [GC] INFO com.tc.objectserver.api.ObjectManager - GC: rescue 2 time   : 0 ms.
2006-01-17 15:50:02,015 [GC] INFO com.tc.objectserver.api.ObjectManager - GC: paused gc time  : 0 ms.
2006-01-17 15:50:02,062 [GC] INFO com.tc.objectserver.api.ObjectManager - GC: total gc time   : 219 ms.
2006-01-17 15:50:02,062 [GC] INFO com.tc.objectserver.api.ObjectManager - GC: STOP 3
```

Added new DSO client: 127.0.0.1:1319

How and When Do You Use Terracotta?

There are four main use-cases where Terracotta is most effective:

- HTTP Session Replication
- Distributed Cache
- POJO Clustering
- Collaboration, Coordination, and Events

HTTP Session Replication

Perhaps the most familiar use-case is HTTP Session replication in a multiple application server environment fronted by a sticky load-balancer. Keeping sessions available across application servers in the event of an application server failure has long been an expensive and difficult problem to solve. The current trend in application design has been to move away from storing application state—like our shopping cart—in the session and towards what has been called a "stateless" application design where the application state is stored in some external system like a database.

This "stateless" approach isn't really stateless—the application state hasn't disappeared, it has just been externalized from the application. This has two unfortunate side effects. The first is performance: writing application state to a database makes that database a bottleneck so that scaling a cluster by adding application servers suffers from diminishing returns. The second is that the programming architecture of the application gets polluted with the APIs required to externalize the application state.

It's much easier to put session-scoped data into the HTTP Session as simple Java objects. That's what it was originally designed for. Terracotta session replication allows the software architecture of a web application to remain simple by keeping the application session data where and in what form it belongs.

Terracotta Session replication provides high-availability by allowing any application server to reach any active session, regardless of where that session was created. It scales well because only the data that has changed in a session is replicated and it is only sent where it's needed. If no data has changed in the session, then no data is written anywhere. If one byte is changed in the session, then only that byte—not the entire session graph—is sent to the Terracotta server. If no other application server has that session in heap (as is the common case for a cluster fronted by a sticky load-balancer), then no other application server will be sent the changes. Only in the event of an application server failure will session data be replicated to another application server and that will happen on demand only.

Because Terracotta clusters regular Java classes, you don't have to chop your session data up into artificially segmented attributes. Your session objects can be as simple or as complex as suits your application. You can make updates to a deep object graph and those changes will be available to the cluster without having to remember to call setAttribute() on the Session. Because Terracotta clusters objects without using Java serialization, you can put objects into the session that don't implement Serializable. For example, the shopping cart can be put into the session as is, without having to cut it up into separate attributes for efficient replication. Changes to the shopping carts from anywhere in the application anywhere in the cluster are automatically reflected without having to call setAttribute().

While the cluster is running, you can look at contents of all of the sessions in the entire cluster. At development time, this has been used to discover abuses of the session where objects that shouldn't have been placed in the session graph accidentally made it in. In production, you can see instantly how many active sessions there are and watch them change in real-time. You can, for example, look to see what people are putting in their shopping carts from the Terracotta console.

Terracotta works with many popular web frameworks like Struts, Spring Web Flow, and Wicket.

POJO and Spring Clustering

All of the objects in the example application are simple POJOs (Plain Old Java Objects). Terracotta makes working with POJOs in a cluster as simple as working with them in a single JVM. This is especially true if your application uses Spring beans.

Terracotta for Spring maintains the non-invasiveness of the Spring Framework in a cluster by enabling clustered Spring beans. You can develop single-JVM Spring applications as usual, then define which Spring application contexts and which beans in those contexts you want to cluster. Terracotta will cluster those beans and the application context events transparently and with the same semantics across the cluster as on a single JVM. Terracotta for Spring also contains support for Spring Web Flow and continuations. This feature enables failover of the conversational state of a Spring web application when that application is hosted on a Terracotta cluster.

Distributed Cache

Clustered objects graphs make naturally good distributed caches. If the data for the Catalog in our example is loaded from a database, populating the Catalog once for the entire cluster instead of once for each application instance in the cluster decreases the load on the database at startup time. Because the entire cache fits into a clustered data structure, application instances don't have to dump the parts of the cache that don't fit into heap and then reload that data from the database when it's needed again. Because Terracotta preserves object identity, the Catalog data can be updated in the simplest possible way using standard Java library data structure semantics and those updates are available instantly to every reference to every object in the clustered object graph in the entire cluster. Maintaining clustered object consistency and cache coherency across a Terracotta cluster is as simple and as easy to understand as the basic data structures that developers use every day.

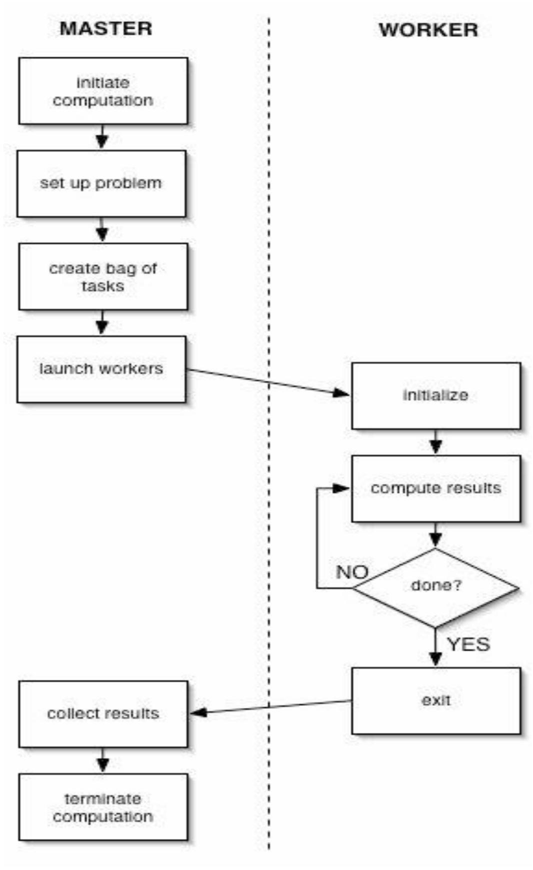
Collaboration, Coordination, and Events

The clustered concurrency features of Terracotta make it ideal for signaling between JVMSs. In the example code, we used the stock CyclicBarrier class to coordinate between threads in separate JVMs. A clustered event mechanism may be implemented in a similarly straightforward fashion.

A common way to add parallelism is for a manager entity to farm out tasks to multiple concurrent workers and collect the results. The Master-Worker pattern is one of the most common ways this is reduced to practice.

The Master-Worker pattern consists of two logical entities: a Master and one or more instances of a Worker. The Master initiates the computation by creating a set of tasks, puts them in some shared space, then waits for the tasks to be completed by the workers.
The shared space is usually some sort of shared queue. One of the advantages of using this pattern is that the algorithm automatically balances the load. Since the work set is shared, the workers continue to pull work from the set until there is no more work to be done. The algorithm usually has good scalability characteristics as long as the number of tasks far exceeds the number of workers and the tasks are roughly similar in the amount of time they take to complete.

Common applications of this pattern are, for example, financial risk analysis and other simulations, search and aggregation on large datasets, as well as sales order pipeline processing.



Libraries currently exist in Java to support building a Work/Manager system based on the Master-Worker pattern. Terracotta enables such a system to be deployed on a scaled-out cluster so that the work can be farmed out not only to workers on a single JVM, but also to workers on the entire cluster of JVMs. By simply clustering the work and results queues, all of the processors in a worker farm can be fully utilized and more can be added as needed without a large disruption in application architecture.

For more information on building Work/Manager applications on Terracotta, see our Master-Worker tutorial on terracotta.org:

http://wiki.terracotta.org/confluence/display/orgsite/TutorialTerracottaDsoWorkManager1

Conclusion

Terracotta is the foundation of a new lightweight software stack. JVM-level clustering enables open-source components like Tomcat, Spring, Geronimo, and a host of open-source application frameworks to be assembled together and deployed with enterprise-class availability and scalability that's easy to use.

To download, find out more, or become a contributor to Terracotta, visit http://www.terracotta.org.

Other References

Terracotta.org
Visit http://www.terracotta.org for downloads, OSS Integrations, documentation, community support, professional support, Terracotta Enterprise Edition and more.

The POJO Container: POJO Clustering at the JVM Level
POJO clustering at the JVM level not only provides a simple way for applications to achieve scale-out, but also sets the stage for a stack of enterprise-scale services and practices based on simple POJOs—the POJO container.

by Ari Zilka

http://www.devx.com/Java/Article/33495/0

How to Build a POJO-based Data Grid using Terracotta
By Jonas Bonér

http://jonasboner.com/2007/01/29/how-to-build-a-pojo-based-data-grid-using-open-terracotta/