TERRACOTTA

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

EXECUTIVE SUMMARY

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. This pressure has caused a proliferation of solutions that have encumbered the architectures of many Java applications.

Terracotta's Network-Attached Memory (NAM) is an arbitrarily large, network-attached Java heap that lets threads in multiple JVMs on multiple machines interact with each other as if they were all in the same JVM. NAM is a fast, simple, and reliable way to add commodity servers to your application's deployment architecture. NAM provides scalability and high availability without torturing your application architecture, overloading your database, and distracting your development team from delivering features.

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 API. Just regular Java will work.

This white paper serves as a technical introduction to what Terracotta is, how it works, and how you can use it to provide simple fault tolerance and scalability for your application.

A Simple Example

To illustrate network-attached memory in concrete terms, let's start with some sample code. The example domain is an online retail system with a catalog of products that can be added to a customer's shopping cart. The set of active shopping carts can be viewed at any time 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 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, is expressed most naturally as simple Java objects with no backing system of record, since it is in-progress data that doesn't need to be stored after a customer places the order.

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

```
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 a product object. The Catalog can be used to display products and to look them up by SKU so they can 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 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 shared object graphs that are used by this application. Shared object graphs are object graphs that live on the network-attached heap. 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, specifically 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, and 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—although not necessarily resident in RAM—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 of all the active carts requires 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 among the JVMs rather than loaded separately by each JVM.

All of the requirements introduced by deploying the application on multiple application servers 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 that makes this happen looks like.

Configuration for the Example

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

```
<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 interaction with the network-attached heap is injected into your code and why you don't have to use a special API. 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 this example, we use a regular expression that encompasses all methods in all included classes for "autolocking":

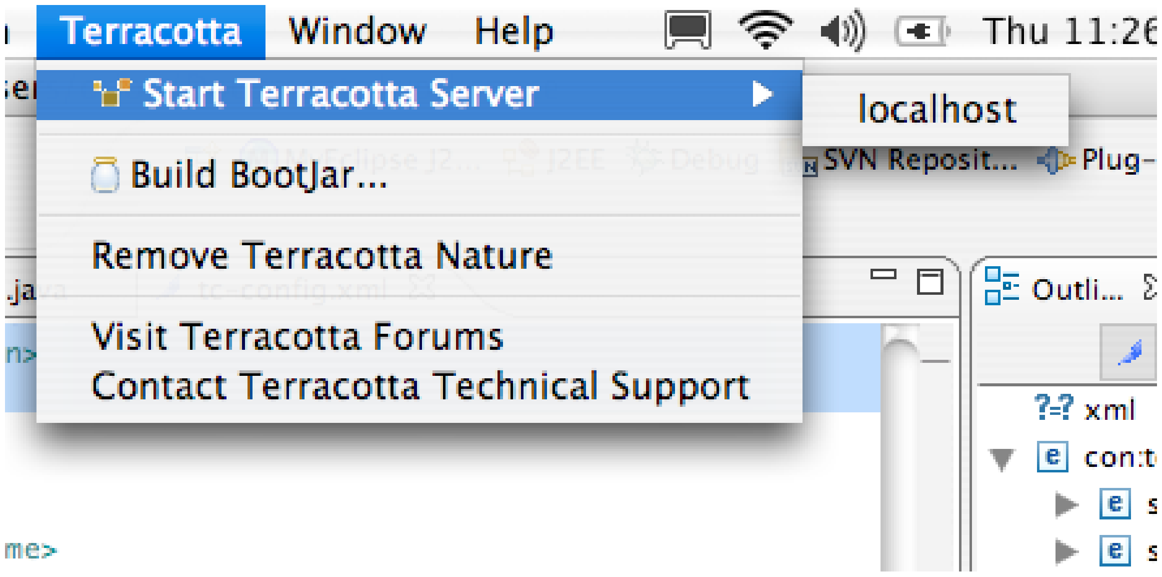
```
<locks>
  <autolock>
    <method-expression>void *..*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
</locks>
```

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

To get a copy of the source code used in this example, go to: http://wiki.terracotta.org/confluence/display/labs/CatalogExample

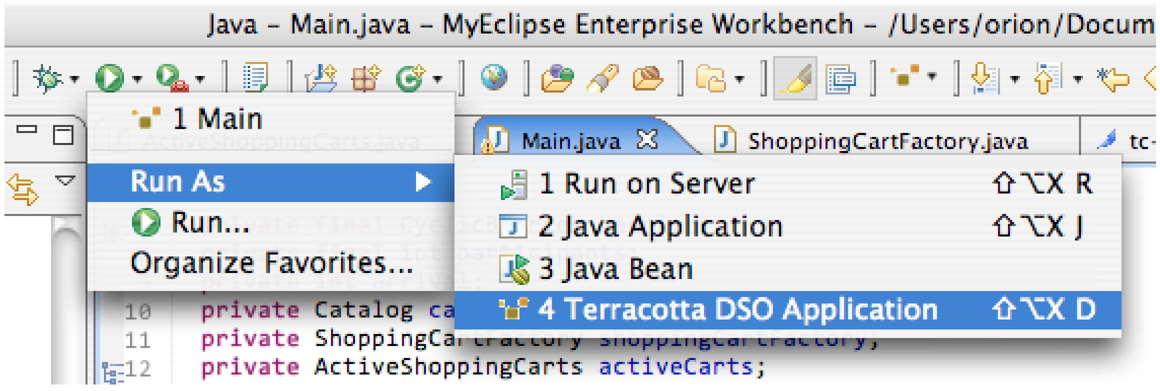
Once the configuration is complete, running the application in a cluster can be done from the command line or from the Terracotta Eclipse plugin. In this article, we'll use Eclipse, but there are multiple examples of starting Terracotta from the command line in the online tutorials and in the download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



Note: In a production environment, or in a development environment that's reached an advanced stage, a Terracotta cluster would include a Terracotta server array. A Terracotta server array is composed of at least two Terracotta server instances working together to ensure cluster availability and the coherency and persistence of shared data. However, for this example, we'll use a single Terracotta server instance.

In Eclipse, starting a Terracotta server instance can be done from the Terracotta menu.

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



```
private Catalog ca
private ShoppingCa
private ActiveShoppingCarts activeCarts;
```

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 attached JVM 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):

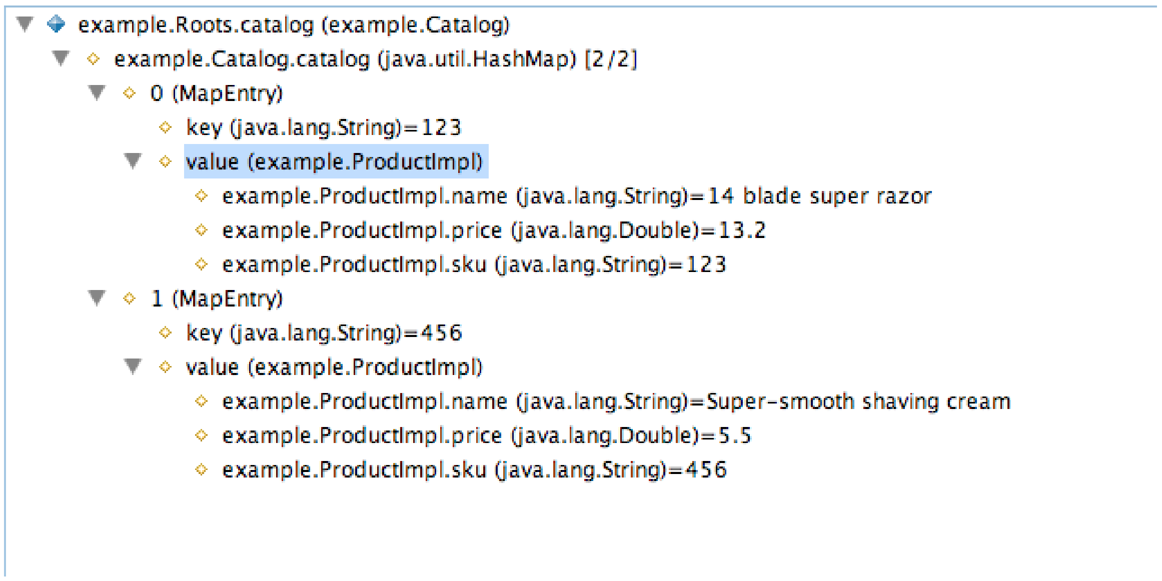
```
Step 1: Waiting for everyone to arrive.  I’m expecting 1 other thread(s)...
We’re all here!

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
```

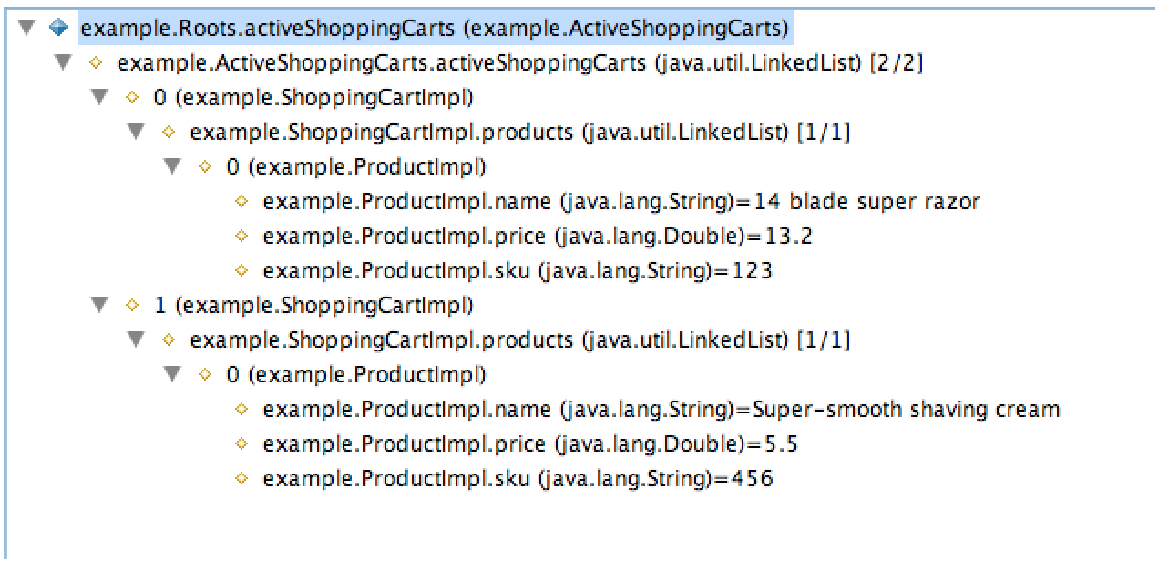
In Step 1, the two threads in the different application instances were waiting for each other to start and arrive at the same rendezvous point. In Step 2, the first thread created Product objects and added them to the clustered Catalog. In Step 3, each thread pulled a Product object out of the clustered 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 into 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
```

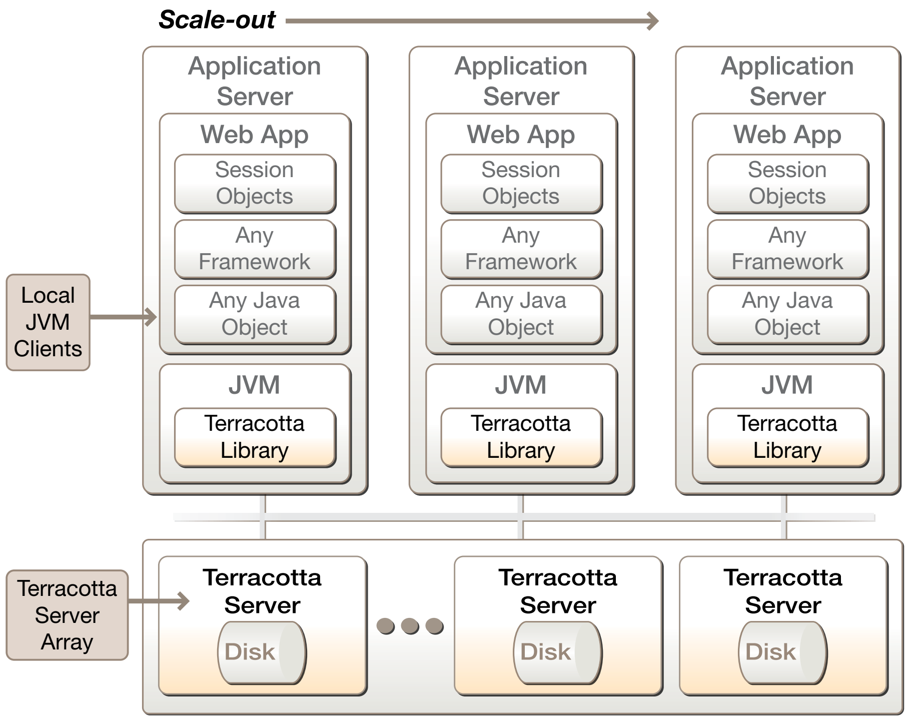
This view shows the Products referenced by the LinkedList inside the ShoppingCart.

HOW TERRACOTTA WORKS

Now that we've seen Network-Attached Memory in action, let's discuss the particulars of how it works.

Architecture

The contents of the network-attached heap and access to it are controlled by the Terracotta server array, which is composed of Terracotta server instances connected by a fast network.



For fault tolerance, the Terracotta server array can have any number of “hot standby” servers waiting to take over should an active Terracotta server instance go offline. The simplest Terracotta server array has one active Terracotta server instance and one standby server instance. The standby immediately becomes active upon failure of the initial server instance, and all of the attached JVMs automatically reconnect to the new server and seamlessly continue working. Because the network-attached heap state can be made persistent, the entire system, including all connected JVMs and the Terracotta servers, can be completely shut down and then restarted to resume work at the pre-shutdown state.

Each attached JVM automatically connects to a Terracotta server instance at startup. The Terracotta server array stores object data and coordinates thread actions between JVMs. The Terracotta DSO libraries inside the application JVMs handle the bytecode instrumentation at class load time and transfer object data and the lock and unlock requests at synchronization boundaries. They also transfer the `wait()` and `notify()` requests between the application JVM and the Terracotta server array at runtime.

Bytecode Instrumentation

NAM awareness is injected into an application using standard bytecode manipulation techniques as classes are loaded into the JVM. 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. (For the purposes of this article, the description of the Terracotta bytecode modifications is simplified.)

To maintain object changes, the `PUTFIELD` and `GETFIELD` bytecode instructions are overloaded. `PUTFIELD` instructions are trapped to record changes to a shared 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 local physical 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 will hold an exclusive lock on that object until the `MONITOREXIT` instruction is executed for that object. If the object in question happens to be a shared object, Terracotta ensures that, in addition to requesting the local lock on that object, the thread also blocks until it receives the exclusive shared lock across all connected JVMS on that object. When the thread releases the local lock, it releases the corresponding shared 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 to be locked with a specific name or to inject synchronization on ‘this’ in order to add synchronization to application code that doesn’t already have it explicitly

The call sites of the `wait()` and `notify()` methods are also instrumented. When a `wait()` method is called on a shared object, a Terracotta server instance adds the calling thread to the set of threads in all connected JVMs waiting on that object. When one of the `notify()` methods is called, the server instance 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 instance causes all threads waiting on that object on all JVMs to be notified.

Roots and Shared Object Graphs

Shared 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 and encapsulation—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 shared objects. Their field data is sent to a Terracotta server instance and stored. Once a root has been created in any JVM, all other assignments to that field are ignored and the value of the shared root object is assigned to that field instead. This happens in the example code 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 by the constructor (shown in the source code), the Terracotta transparency libraries retrieve the root from the Terracotta 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 an unshared object becomes referenceable from a shared object, the new object and the entire graph of objects reachable by reference by the new object become shared on the network-attached heap. Once an object becomes shared, it is assigned a unique object ID by Terracotta and is shared for the remainder of its lifecycle. When an object becomes unreachable by any root graph and there are no instances of it on any attached JVMs, then it is eligible for removal by the distributed garbage collector in the Terracotta server array.

Synchronization, 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 shared lock request such that the calling thread will block until it is granted both the local lock and the shared 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

Terracotta manages changes to shared objects as efficiently as possible. Changes are managed at the object field level and sent across the network only to where they are needed. The transactions that contain changes to objects contain only the data of the fields that have changed. These transactions are sent to a Terracotta server instance and to the other attached JVMs to keep the cluster consistent. The server instance 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 on which the transaction 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 a Terracotta server instance. The server instance 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 things we had to ship around the cluster were 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. Had the Product object been a deep graph with lots of references to other objects (which in turn could have references to other objects), Java serialization would have serialized the entire deep graph—all for a change to a `double` value.

Terracotta's change management 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 clustering technologies require a GET/PUT API. With such an API, 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 shared 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 local heap. When changes are made remotely to that object, the transaction is received by the local JVM and applied directly to the existing object, which is already on the heap. There is one and only one instance of a shared object on a local heap at a given time. The situation gets slightly more complicated when multiple classloaders come into play, but that's beyond the scope of this article.

Using Network-Attached Memory, 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. Because there are no copies—a shared object is just a plain object on the heap—shared objects behave just like any other object: any changes you make to a shared 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 local heap as the Products in the shopping cart.

This preservation of object identity makes applications running on multiple JVMs connected to the network-attached heap behave like regular, single-JVM applications. This simplicity and the power of object identity preservation allows for the deployment architecture of you application to be made orthogonal to your application's object and programming model. Infrastructure concerns such as how many machines will be deployed to serve the application are logically pushed down into the Terracotta layer at the JVM-level, melting away into the infrastructure where they belong. Much like garbage collection allowed memory management to disappear from application code, Terracotta allows distributed computing concerns to disappear as well.

Virtual Heap

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 shared 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 Terracotta server array.

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 shared object graph. The Catalog need only be populated once, but is instantly available to every attached JVM. This dramatically reduces startup time for additional application instances.

USE CASES AND CASE STUDIES

Use Case: Replicated HTTP Sessions

One of the most common places that web application state gets kept is in the HTTP session. Much of the push towards stateless architectures is in direct response to the difficulty of maintaining session state across application servers. Taking application data out of the HTTP session is in large part what people mean when they say “make your application stateless.”

However, the HTTP session is a natural feature of web applications. Conversational state likes to live in the session. In fact, many web frameworks are difficult or impossible to use without it.

Terracotta’s Network-Attached Memory takes nearly all the pain out of scaling out a session-rich application. Assuming requests associated with the same session are routed to the same application server when possible by a sticky load balancer, only that application server knows anything about that session. If requests for that session happen to be made against another application server instance (because, for example, the original application server instance was taken down for maintenance), that other application server will---on the fly---automatically fault in the required session objects from the Terracotta server array.

From your application’s perspective, you can treat the session data like regular objects on your JVM heap. Your session objects don’t have to be Serializeable so you can put almost anything you want into the session and it will be available anywhere in the cluster. And, you don’t have to remember to call `Session.setAttribute(...)` when you’ve made changes to an object in the session. Object changes are captured and replicated on a per-field basis. Changes to the Terracotta virtual heap are just like changes to the regular Java heap. But, those changes aren’t replicated to any other member of the cluster until they are actually needed, so there is no network bottleneck or extra overhead to propagate changes. The only data that needs to move is the changed fields and those changes only need to be sent exactly where they are needed, when they are needed.

Because the shared object data is homed inside the Terracotta server array, you can take down some or all of your web cluster and the session data will still be there when you start it up again. Shared data will even survive a restart of the entire Terracotta server array. In addition, the Terracotta server array’s high-availability architecture allows the cluster to survive a hardware failure or a maintenance shutdown of particular Terracotta server instances. Another Terracotta server instance automatically take the place of lost ones, and the web cluster continues normal operation.

Network-attached memory makes session replication simple, efficient, reliable, and takes all of the session load off of your database.

Use Case: Clustered Cache

Besides session data, a large part of application state is in caches of various kinds. These can be database caches, caches of the results of web services calls, caches of parsed documents, or metadata about filesystems, etc.; most web applications employ caches of one kind or another.

As architectures scale out, these caches become less and less effective. We call this the 1/*n* effect: the effectiveness of a cache is 1/*n*, where *n* is the number of instances of the cache (i.e., the number of application servers in your cluster), especially if not all of the data to be cached actually fits into RAM so that some elements of the cache must be evicted. The reason the 1/*n* effect applies is that each instance of the cache must be populated independently from the data source. The number of times the cache data must be read from its source grows with the number of cache instances. If the cached data set is larger than will fit in RAM, the situation is even worse as the data must be continually loaded from its source as old cache entries are evicted to make room for new ones.

These degrading caches can result in decreasing throughput, ever-slower startup times and, at its worst—for example, in the case of a full cluster restart where all of the application servers are starting at the same time and loading their caches independently from the data source—can completely saturate the data source, causing it to fail.

Keeping many separate cache instances up to date represents another significant challenge. If your strategy for keeping your cache coherent with the underlying data source is a time-based purging mechanism, then you will have to go back to the data source ever more as you increase the number of cache instances. If you use an invalidation mechanism, you have to multiplex the invalidation messages to each instance of the cache. If you use an update mechanism where the contents of the cache are updated as the data source changes, you can end up with a flurry of update messages that can overwhelm the message bus.

In any case, populating and keeping multiple instances of a cache up to date is a chore that NAM can handle for you gracefully, eliminating the 1/*n* effect and the cache update problem.

Because the cache data is available to all connected JVMs, it only needs to be loaded once for the entire cluster, not once per application server, alleviating significant load from the cache data source (for example, your database or a web service). The automatic memory management of Terracotta lets each application server view the portion of the cache that it cares about, so the cache can be MUCH larger than the available RAM of your app servers. Because of this, caches of large datasets don't need to be purged and reloaded due to memory constraints, further alleviating load on the cache data source.

Just like the cache only needs to be loaded once for the entire cluster, it likewise only needs to be updated once for the entire cluster as the source data changes. Because Terracotta moves clustered object data efficiently at a field level, you can write the cache updating mechanism as if there were only a single cache instance to worry about and Terracotta will transparently and efficiently keep all instances up to date.

Finally, since the cache is clustered, the work required to load the cache can be offloaded, if need be, from your application servers to its own, dedicated machine.

Use Case: Very Large Object Graphs

Because Network-Attached Memory gives you an arbitrarily large virtual heap, it allows you to create and manipulate very large object graphs without requiring enough physical RAM to fit that entire object graph in memory at once. The window into the very large graph of objects that you are actually manipulating at any given time is automatically materialized onto the physical heap of your JVM. This behavior is similar to the way filesystems (including network-attached storage filesystems) cache active portions of the filesystems in RAM for quick access.

This capability allows your application to deal with very large object graphs gracefully. You don't have to manually break your data structures up and fault them from and flush them to an external data store—that happens for you as a natural effect of using objects on the network-attached heap.

Use Case: Collaboration, Coordination, and Events

Because these objects are available on the network, multiple instances of your applications on different physical machines can attach to the networked heap and manipulate the shared objects as if they were all running in the same physical JVM.

This shared heap would be useless without some way of protecting portions of it from access by multiple concurrent readers or writers. As described earlier, Terracotta extends the existing concurrency semantics inherent in the Java language and the Java Virtual Machine to have an equivalent meaning across threads in multiple virtual machines on multiple physical machines. These concurrency semantics continue to behave as they do across threads in the same virtual machine.

This cross-JVM signaling behavior makes Terracotta an excellent utility for collaborating, coordination, and eventing across virtual machines. The same tools you would use in a single JVM to provide signaling (for example, queues, barriers, latches, rendezvous, etc.) between threads in a single virtual machine can be used unchanged in a context of multiple virtual machines.

Consider an event mechanism where an event publisher pushes events onto a LinkedBlockingQueue and a listener thread takes events off of that queue and dispatches them in a while loop. The event publishing code might look something like this (simplified for purposes of example):

```
package event;

import java.util.concurrent.LinkedBlockingQueue;

public class EventSink {
     private final LinkedBlockingQueue<Event> messageQueue;

     public EventSink(LinkedBlockingQueue<Event> q) {
            this.messageQueue = q;
     }

     public void fireEvent(event.Event event) {
            try {
                   messageQueue.put(event);
            } catch (InterruptedException exception) {
                   // Do something useful
                   throw new RuntimeException(exception);
            }
     }
}
```

The event listener/dispatcher might look something like this:

```
package event;

import java.util.Collection;
import java.util.Iterator;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.LinkedBlockingQueue;

public class EventDispatcher implements Runnable {
     private final LinkedBlockingQueue<Event> eventQueue;
     private final Collection<EventListener> listeners;

     public EventDispatcher(LinkedBlockingQueue<Event> q, Collection<EventListener>
listeners) {
            this.eventQueue = q;
            this.listeners = listeners;
     }

     public void run() {
            while (true) {
                   try {
                          Event event = eventQueue.take();
                          for (Iterator<EventListener> i =
listeners.iterator(); i.hasNext();) {
                                 i.next().handleEvent(event);
                          }
                   } catch (InterruptedException exception) {
                          // Do something useful...
                          throw new RuntimeException(exception);
                   }
            }
     }
}
```

Using Terracotta to make the eventQueue a shared object on the virtual heap, multiple JVMs can instantiate instances of the EventSink and fire events that will get dispatched on the JVM that has instantiated the EventDispatcher. Or, if you want multiple JVMs to listen to the same event queue, you can instantiate multiple instances of the EventDispatcher all taking off of the same shared queue.

The thread coordination that already exists in LinkedBlockingQueue is used by Terracotta to make shared queues behave the same across JVMs as they do in the same JVM. Similarly simple mechanisms can be created with CyclicBarrier and other utilities in the java.util and java.util.concurrent packages. While Terracotta works well with many of the standard Java library classes, if you want to write your own concurrency utilities, Terracotta will support those, too.

With Network-Attached Memory, creating a signaling mechanism that works across multiple virtual machines can be as simple as creating one for a single virtual machine.

Use Case: Distributed Work Management

Using a similar strategy as discussed above with the distributed event queue, you can also use Terracotta to distribute work across multiple virtual machines. Creating a "worker" JVM can be as simple as instantiating threads that take jobs off of a shared queue. Sending work to the worker JVMs can similarly be as simple as queuing jobs to the appropriate work queue. Results can be retrieved via a results queue and aggregated, if necessary, on a master JVM. Just as threads can be added in a single JVM to handle work, Terracotta lets you add the power of the threads in multiple JVMs as needed to scale out as workload increases.

Use Cases Are Fundamentally Open-Ended

The use cases of Network-Attached Memory are essentially open-ended. Its most obvious application is for simple availability and scalability, but that's really just part of what Terracotta can be used for. Anywhere that an arbitrarily large, shared, networked heap coupled with transparent cross-JVM coordination and communication is useful, Terracotta will come in handy. For example, one of the first Terracotta-enabled applications ever written was our own distributed testing framework. We use Terracotta to test itself. The uses of Network-Attached Memory are limited only by your imagination.

CONCLUSION

Terracotta makes high availability and scalability for Java simple, fast, and reliable. Network-Attached Memory enables open-source technologies 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, visit http://www.terracotta.org.

ABOUT TERRACOTTA, INC.

Terracotta is infrastructure software that provides affordable and scalable high-availability for Java applications. Companies use Terracotta to offload work from databases and application servers and to reduce their development efforts. Founded in 2003, Terracotta, Inc. is a private firm headquartered in San Francisco. More information is available at www.terracottatech.com. Terracotta's open-source community is at www.terracotta.org.