Clustering the Java Virtual Machine using Aspect-Oriented Programming

Jonas Bonér, *Terracotta Inc.*
Eugene Kuleshov, *Terracotta Inc.*

ABSTRACT

Clustering (and caching) is a crosscutting infrastructure service that has historically been implemented with API-based solutions. As a result, it has suffered from the same code scattering and tangling problems as other crosscutting concerns.

In this paper we will show how *Aspect-Oriented Programming* (*AOP*) can help to modularize clustering and turn it into a runtime infrastructure *Quality of Service*. We will show how *AOP* can be used to plug in directly into the *Java Memory Model*, which allows us to maintain the key *Java* semantics of *pass-by-reference*, *garbage collection* and *thread coordination* across the cluster, e.g. essentially cluster the *Java Virtual Machine* underneath the user application instead of the user application directly

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.2.10 **[Software Engineering]** Design; D.3.2 **[Programming Languages]** Languages

General Terms

Reliability, Languages, Software Engineering, Separation of Concerns

Keywords

Aspect-Oriented Programming, Clustering, Java, JVM, Distributed Computing

1. INTRODUCTION

Clustering is becoming increasingly important in the world of enterprise application development. Developers continuously need to address questions like: *How do I enhance scalability by scaling the application beyond a single node? How do I guarantee high-availability, eliminate single points of failure, and make sure that the SLAs (Service Level Agreement) defined by the customer are met?* These are all questions that, in one way or the other, imply clustering. Predictable capacity and high availability are operational characteristics that an application in production must exhibit in order to support a sustainable business. Some companies require up to 99.9999 percent uptime in their application; others do not, but all applications need to remain operational for as long as the *SLA*s defines.

What do we mean when we say clustering and how does it differ from caching? The definition of clustering that we use in this paper is: sharing the application state across multiple *Java Virtual Machines* (*JVM*'s), while caching can be defined as: bring the application state closer to its execution context. In this sense, caching is a subset of clustering.

One common approach to address scalability issues has been by "scale-up", meaning add more power in terms of CPU and memory to one single machine. But today most data centers are running cheap commodity hardware and this fact, paired with more demand for high availability and failover, instead implies an architecture that allows you to "scale-out", e.g. adding more power in terms of more machines - which implies using some sort of clustering technology.

The problem is that clustering has been a hard problem to solve. In the context of enterprise applications, this in particular means ensuring high-availability and fail-over of the user state in a performant and reliable fashion. In case of a node failure (the application server or *JVM* crashes), enabling the use of "sticky session" in the load balancer (which means that the load balancer is always redirecting requests from a particular user session to the same node), won't help much. But an efficient way of migrating user state from one node to another in a seamless fashion is needed.

The minimal set of requirements that we think an enterprise-class clustering solution should meet are:

- scalability
- high-availability
- fail-over
- performance
- minimal impact on existing code
- simple deployment and configuration
- runtime visibility (monitoring)

Clustering is a crosscutting infrastructure service that has historically been implemented with API-based solutions. As a result it has suffered from the same code scattering and tangling problems as other cross-cutting concerns.

As we will see, using *Aspect-Oriented Programming (AOP)* [1] can help to modularize clustering so effectively that user code can become oblivious to clustering. This can be done by using *AOP* to plug in to the *Java Memory Model* (*JMM*) [2] and maintain its semantics along with the semantics defined in the *Java Language Specification (JLS)* [3] across a distributed environment.

We will also show that using *AOP* not only helps with modularization and obliviousness, but also turns out to be a key enabler in achieving high performance and scalability.

At *Terracotta* [7], we have a custom load-time weaving framework, built on *AspectWerkz* [4] and custom byte code instrumentation based on *ASM* [5]. This framework is used to capture and modify target application events that are of interest in terms of clustering or distributed computing, e.g. *join points* and *advice*, as well to do necessary enhancements (such as adding interfaces, methods and fields) to the target application, e.g. *Inter-Type Declarations* (*ITD*s).

The foundation of this framework was built over two years ago and unfortunately there is still no common *AOP* framework that can provide the same features set, which is one of the reasons why such a hybrid approach is being used.

In this paper we are going to focus on issues related to the application transformations required to make the clustering cross-cutting concern orthogonal to the target application. For the purpose of this paper we are going to illustrate the concepts using *AspectJ* [6], this in order to base the discussion on a language that is commonly understood, leaving deployment issues out of scope of this paper. As you will see later, most of the original *Terracotta aspects* can be reimplemented directly in *AspectJ aspects*, but, as we will see, there are some *ITD* transformations that we cannot do in the current version of *AspectJ*.

This paper is focusing on the *AOP* part of the implementation. We will explain the implementation and semantics of each *pointcut* and *advice* in detail, but since a discussion on how other subsystems, like the *garbage collector*, *instance manager*, *lock manager*, *transaction manager* or the *network transport protocol* are out of scope for this paper, we will delegate to the *ClusterManager* abstraction when calls to these subsystems are being made. However, we will explain the semantics and the pre- and post conditions for these API calls.

2. SAMPLE APPLICATION

We have a very simple sample application that will drive the discussion. This sample application will function as the target application that we want to cluster, e.g. weave in our clustering aspect into. It is a simple counter abstraction (sort of reversed *CountDownLatch* [14]) with two methods; *increment()* -- which increments a counter value, and *waitFor()* -- which waits for the counter to reach a specific value. Its sole purpose is to implement the characteristics that are interesting from a clustering standpoint; *state* (that is changing during the lifetime of the application) and *thread coordination* (using *wait/notify* and *synchronized* blocks).

```
public class Counter {
  public final static Counter soleInstance = new Counter();

  private int value = 0;

  public void increment() {
    synchronized(this) {
      this.value++;
      notifyAll();
    }
  }

  public void waitFor(int expected) {
    synchronized(this) {
      while(this.value < expected) {
        try {
          wait();
        } catch(InterruptedException ex) {
        }
      }
    }
  }
}
```

3. HUB AND SPOKE VS PEER TO PEER

Terracotta is using an architecture known as *hub-and-spoke*, which means that it has one central L2 server and N L1 clients (the L1 client is running inside the target *JVM*). This might seem strange, since most clustering solutions on the market today are using *peer-to-peer*, but as we will see, *hub-and-spoke* has some advantages and plays a key role in some of the optimizations that we will talk about later. We will refer to this central server as the *Coordinator*, even though coordination is only half of its job.

First it serves as the coordinator ("the traffic cop") in the cluster. It uses the *lock manager* to keep track of things like; which thread in which node is holding which lock, which nodes are referencing which part of the shared state, which objects have not been used for a specific time period and can be paged out, etc. Keeping all this knowledge in one single place is very valuable and allows for very interesting optimizations.

Second, it serves as a dedicated state database. This means that it stores all the shared state in the cluster. The state server does not know anything about *Java*, but only stores data and IDs. The *Coordinator* itself is clusterable through a *SAN*-based [16] failover mechanism. This means that it is possible to scale-out the *Coordinator* (L2 server) in the same fashion as most *peer-to-peer* solutions but with the advantage of keeping the L2 separate from the L1 (see below for a discussion on some of the problems with not separating them). This is the way that *the Internet* scales.

One of the problems with using *peer-to-peer* is that you need to replicate everything everywhere, basically do multicast. For obvious reasons, this cannot scale very well.

However, it is important to understand that this is only true for pure *peer-to-peer* solutions, and that most clustering solutions on the market today that advertise themselves as *peer-to-peer* are actually using some sort of hybrid mechanism between *hub-and-spoke* and *peer-to-peer* in which they bundle the L2 in the L1. Unfortunately, this hybrid solution has problems such as relying on the notion of an *object home*, with potential secondary homes. The problem with this architecture is that it's really hard to know on which set of nodes the entire representation of an object lives and it is therefore easy to accidentally wipe out an object entirely.

4. STATE SHARING

We have split up the problem of clustering into two different parts; *state sharing* and *thread coordination*. In reality they are too tightly integrated to split up, but for simplicity and clarity we will try to discuss each one in isolation.

First we will talk about *state sharing*, meaning how to find out which parts of the *Java* heap has changed, how to track that *change set* in a *Unit of Work,* how to replicate the *Unit of Work* to the parts of the distributed environment that need it - when they need it, and finally how to merge the *change set* into the *Java* heap on these other nodes.

Second, we will talk about *thread coordination*, meaning how to maintain the semantics of the *JMM* in order to ensure correctness and coherence of the data that is shared, throughout the distributed environment.

Traditional clustering and distributed computing platforms in *Java* have been API-based and have in most cases used *Java serialization* [15] to transfer data between different *JVM*s. As we discussed earlier, this introduces problems like code scattering and code tangling. But it also introduces two other, perhaps more subtle and less commonly understood problems. Both of these problems are inherited from the use of *Java serialization*. In the following sections we will discuss these problems as well as what we have done at *Terracotta* to address them.

4.1 Problems with Java Serialization

4.1.1 Breaks Java's pass-by-reference semantics

The first problem with *Java serialization* is that it creates deep clones (copies) of the object graph structure. At first glance, this might not sound like much of a problem but it actually turns out to have a serious effect on application architecture and design.

The *Java* language, as defined by the *JLS*, has *pass-by-reference* semantics. But the problem is that when you are using *Java serialization* you cannot rely on these natural *Java* semantics anymore, since *Java serialization* breaks *pass-by-reference* semantics. In other words; object identity is broken. The implications that this has on the design and architecture of an application should not be underestimated.

If object identity is broken, and one cannot rely on pass-by-reference semantics any longer then developers have to maintain the relational references between objects themselves. This usually forces the developer to layer some kind of primary-key/foreign-key mechanism onto their object model, e.g. break down and maintain the object references using relational maps and almost start thinking like relational database designers.

4.1.2 Can have performance implications

The second problem with API and *Java serialization* based replication is that it is too coarse-grained. There is no way of detecting which parts of the *Java* heap that have actually changed, but the memory allocated for the whole object graph, that is referenced from the "stale" instance, needs to be treated as "stale" and therefore it has to be replicated. On top of this, *Java serialization* can only work on the *Java* object structure level which means that it will not only convert the actual data, but also all information about the class structure etc., into a network transportable format. All these things will force unnecessary data to be sent over the wire, as well as marshaled and unmarshalled, something that can have performance and latency implications.

Another problem with not being able to detect actual changes is that it forces the use of a coarse-grained locking mechanism when marshalling and unmarshalling on modifications. A lock needs to be taken on the top-level object regardless of the scope of change, something that can also cause premature lock contention. These are things that can also have performance implications.

4.2 Roots

For practical reasons, it is impossible to cluster the whole *Java* heap (replicate every single change) and at the same time achieve acceptable performance. This might sound like a serious limitation, but it actually turns out to be seldom, if ever, a real problem, that in real world scenarios. Most applications clearly distinguish between transient data, local data and data that need to remain coherent across the cluster.

In order to identify these parts of the *Java* heap, *Terracotta* introduces a new abstraction called *Root*. A *Root* can be either a static or a member variable in a *Java* class and *Root* represents the top of an arbitrarily large object graph, which state is ensured to be made consistent across all nodes in the cluster. All instances referenced from that graph, directly or indirectly, will become clustered.

In our sample application we could pick the static field holding the sole instance of the *Counter* in the *Counter* class to be the *Root*, which will make sure that the state for the *soleInstance* field (and its references) is consistent throughout the whole cluster. In order to do that, we need some additional metadata for this field. The *Terracotta* framework is using its own *XML*-based [17] descriptors to configure the required metadata, but to keep things simple, we will in our illustration make use of a field level annotation [8] called *@Root*:

```
@Root
public final static Counter soleInstance = new Counter();
```

The *@Root* annotation can be also injected using an *ITD* like this:

```
declare @field : * Counter.soleInstance : @Root;
```

We can now define the *AspectJ advice* for maintaining the semantics of a *Root* field like this:

```
Object around() : get(@Root * *.*) {
  String name = thisJoinPointStaticPart.getSignature().toLongString();
  return ClusterManager.getOrCreateRoot(name);
}
```

As you can see, the *pointcut* for this *advice* picks out all *join points* where a field annotated with the annotation *@Root* is accessed. When the *join point* is executed, the *advice* invokes a method in the *ClusterManager* that short circuits the real field access and based on the signature of the field returns the clustered object instance instead. This ensures that the *ClusterManager* can maintain the current semantics for the *Root* field; if the field has never been accessed before then create and return the clustered instance, else always return the same clustered instance.

4.3 Configuration

In the previous section we introduced the *@Root* annotation as a configuration element. Before we continue the discussion, we need to introduce one additional configuration element; the *@Clustered* annotation. This annotation identifies classes that access, modify, or can potentially join, the distributed object graph that is referenced from a *Root* field. It should be used to not only identify classes whose state (i.e. instance fields) should be distributed, but also on classes that can access distributed objects and needs to preserve the semantics of the *JMM* across the cluster, including synchronization and wait/notify calls. Here we apply the class level *@Clustered* annotation to the *Counter* class.

```
@Clustered
public class Counter {
  @Root
  public final static Counter soleInstance = new Counter();
  ...
}
```

All the *pointcuts* are completely generalized and rely only on these annotations. As mentioned earlier, both annotations can be used either explicitly in the source code or they can be injected using an *ITD*. *AspectJ* allows the annotation *ITD*s to be defined using patterns, which is something that simplifies this style of configuration.

Note: our use of *Java* annotations will require using *Java* 1.5 or above, however this is not a limitation of the original *Terracotta* runtime, which is using *XML* descriptors

4.4 Maintaining Java's pass-by-reference semantics

Now, when we have identified the *Root* of the distributed object graph and all the classes that can access, modify or potentially join the distributed object graph we have all mechanisms in place in order to detect when a new instance is attached to a distributed object graph. This allows us to maintain a cluster-wide identity for each shared instance.

For convenience, we will define *isClustered()* pointcut that will be used in several other *advice*:

```
pointcut isClustered() : @within(Clustered);
```

Now, we need to define a *pointcut* that picks out all *join points* where a field on an object that is already attached to the distributed object graph, is modified (starting from the root).

When any of these *join points* are executed, we need to dispatch to an *advice* that notifies the *ClusterManager* about the field change, passing in the object instance, field name and the new value. The *ClusterManager* would delegate to the *object manager* subsystem to propagate these changes to other nodes.

If the field is not of type *literal* (a *Terracotta* term which means that it is a primitive or in some cases an immutable object, see below), but a regular object instance then the *ClusterManager* records that this object has joined the object graph for this specific *Root*. However, if it is a *literal* then the *ClusterManager* will record the data (the bytes that have changed in the *Java* heap) for this *literal*, along with the name of the field and the id for its enclosing object instance, in the *Unit of Work* for this specific critical section (synchronized block).

If it is a newly created object that has not yet been distributed, the *object manager* would have to send the bytes allocated for the entire object, along with its unique id, to the *Coordinator*.

In some cases, immutable objects (such as *java.lang.String*) can be treated as primitive types. *Terracotta* is using the term *literal* for those special cases. However those implementation details are out of the scope of this paper.

Similarly, on field access, we need to make sure that the field always holds the most recent, up-to-date, value. That excludes the *Root* fields handled by the *advice* discussed in the previous section.

```
pointcut isClusteredTarget(Transparent o) :
   isClustered() &&
   @target(Clustered) &&
   target(o);
```

```
pointcut setClusteredField(Object o) :
   isClusteredTarget(o) &&
   set(!@Root * *.*);

before(Object o, Object value) :
   setClusteredField(o) && args(value) {
  String name = thisJoinPointStaticPart
     .getSignature().toLongString();
  ClusterManager.fieldChange(o, name, value);
}
```

It won't be efficient enough to do reconciliation on every field access but all we need to do is to update all changed fields at once, in one atomic operation. In order to enable that, we define a *TransparentAccess* interface that we will introduce (using an *ITD*) on every class annotated with the *@Clustered* annotation. This interface has two methods - *getFields()* and *setFields()* which retrieves or writes a set of field values in one single atomic operation:

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

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

public Map TransparentAccess.getFields() {
  return ClusterManager.getFields(this);
}
```

The implementation above is good for illustration purposes and is the best you can achieve with *AspectJ*, but it is obviously not the most performant one. To solve this in *Terracotta* we have used a special type of *ITD*, which generates a class specific, optimized implementation of the *getFields()* and *setFields()* methods that does not use *Java Reflection* [18] to access or modify instance fields. This can not be efficiently done in *AspectJ* because it needs to know actual field names for the target class. Method *setFields()* takes *java.util.Map* of field values and sets them to the corresponding fields. Method *getFields()* does the opposite and returns a *java.util.Map* with the current values for the clustered fields.

Now we can write a generic *pointcut* and *advice* for field access which retrieves clustered state:

```
pointcut getClusteredField(TransparentAccess o):
   isClusteredTarget(o) &&
   get(!@Root * *.*);

before(TransparentAccess o) :
   getClusteredField(o) {
  String name = thisJoinPointStaticPart.getSignature().toLongString();
  Map fields = ClusterManager.reconcile(o,name);
  o.setFields(fields);
}
```

This gives us both very fine-grained control of the object state changes, as well the ability to batch change deltas before sending them to the remote nodes. This ensures both good performance (on a single node) as well as allowing the *Coordinator* to optimize the communication between different nodes in the cluster. For instance, it can bring data locally, or collect usage statistics and predict ahead of time what data will be needed locally.

In our *Counter* sample class, this would mean that the *ClusterManager* will detect that the only change that has occurred in the object graph that is reachable from the *Counter.soleInstance* static field (that we marked as being *Root)* is the *Counter.value* member field, and will make sure that only this *integer* of four bytes is sent to the *Coordinator*.

4.5 Object identity

The different *advice* that we have discussed in the previous section relies heavily on some sort of cluster-wide *object identity*. For example, the *fieldChange()* method in the *ClusterManager* needs to have a way to uniquely identify the modified instances when committing the changes made to the local heap, to the *Coordinator*. I.e., it needs to be able to map a specific *change set* to a specific *object identity*. The same *object identity* is then used by the *Coordinator* when the *change set* is replicated to the different nodes in the cluster, as well as in the *reconcile()* method in the *ClusterManager* when this data in the *change set* is merged back into the local heap on the specific nodes. This task is challenging, because *object identity* should be reproducible on any node in the cluster and should guarantee object uniqueness for the entire cluster. These are challenges that we are facing when implementing the *object manager* and *garbage collection* subsystems, but since their implementations are out of scope of this paper, we will only discuss them briefly.

4.5.1 Class loader identity

Java's flexible and in some sense complex class loading architecture makes it hard to identify the *ClassLoader* that has loaded a specific instance. For example, if you have the same class instantiated in two different class loaders then you will get two distinct instances. We need to be able to uniquely identify each one of those. In some cases it is possible to use information from the application container. For example, each *Java Enterprise Edition (JEE)* [8] bundle (*WAR* or *EAR* module) has its own class loader associated with the application name. But solving the problem generically is harder.

4.5.2 Managed components

JEE application servers and *Dependency Injection (DI)* [10] containers such as *The Spring Framework* [11] introduce another class of objects, a.k.a. components, in which the instances are not created by the user, but their creation is delegated to a container which takes care of injecting all of its necessary dependencies. Maintaining a cluster-wide identity for these components requires special support, in order for the semantics of the component's life cycle to be preserved.

4.6 Virtual heap

As we have seen in previous sections, the *ClusterManager* has full control over the state in each clustered object instance, something that allows it to enrich the regular semantics of *Java* memory management.

It can for example bookkeep memory access, e.g. keep a record of how often a specific object instance, including all its references, has been accessed. This allows it to detect if a specific sub-tree of a distributed object graph has not been used for a certain amount of time and it can then decide to page out all the memory allocated on the *Java* heap for this specific sub-graph, to the *Coordinator* (L2 state server). This is done by setting all the fields in the top level object in the sub-graph to *null* using the *TransparentAccess.setFields(Map) ITD*.

If some object later is trying to access a paged-out sub-graph (with a top level field set to *null*) then this access operation will be guarded by the *ClusterManager* which will make sure that all the necessary state is first retrieved from the *Coordinator* and paged back into the local *Java* heap before it hands out a reference to the sub-graph to be used.

This feature allows an application to work with a *virtual heap* that is much larger (sometimes orders of magnitude larger) than the actual physical heap that *JVM* provides, but still have acceptable performance.

4.6.1 Distributed garbage collection

Heap virtualization also affects the garbage collection. To allow garbage collection of the local objects, *Terracotta* is using *java.lang.ref.WeakReference*s and monitors when referenced objects are collected using a *java.lang.ref.ReferenceQueue*. This way the *Coordinator* always knows if nodes have local references even for distributed objects that are detached from the distributed object graph.

To collect the objects eligible for garbage collection from the virtual heap, the *Coordinator* runs a special process that gets the set of objects currently in the shared graph. Then it removes objects that are held by all nodes and then traverses the distributed object graph from the root, removing all reachable objects. After that, anything that left in the set can be deleted from the virtual heap.

5. THREAD COORDINATION

The *JMM* determines what values can be read at every point in the program. In multi-threaded environments it also allows complete prediction of the values that are seen by each thread. We already saw how object reads and writes can be made transparent on the cluster. However, to preserve the semantics of the *JMM*, we also need to support locking for both synchronized statements and synchronized methods.

5.1 Locking and synchronization

In *AspectJ*, *join points* matching critical sections, in *Java* called *synchronized blocks*, can be picked out by the *lock* and *unlock Pointcut Designators (PCD)*, which match the *MONITOR_ENTER* and the *MONITOR_EXIT* bytecode instructions, respectively.

Note: this is currently an experimental feature which can be enabled using the *-Xjoinpoints:synchronization AspectJ* compiler flag.

Here are the two *pointcuts* that pick out all the critical sections (entry and exit) where a clusterable instance is being locked on (and most likely a shared instance is being modified):

```
private pointcut clusteredMonitorEnter(Object o):
   isClustered() &&
   lock() &&
   @args(Clustered) &&
   args(o);

private pointcut clusteredMonitorExit(Object o) :
   isClustered() &&
   unlock() &&
   @args(Clustered) &&
   args(o);
```

Then we can write a *before* and *after advice* that will delegate to the *monitorEnter()* and *monitorExit()* methods in the *ClusterManager*:

```
before(Object o) : clusteredMonitorEnter(o) {
  ClusterManager.monitorEnter(o);
}

after(Object o) : clusteredMonitorExit(o) {
  ClusterManager.monitorExit(o);
}
```

These calls will be delegated to the *lock manager* in the *Coordinator*, which will use the *object identity* of the *pointcut* argument instance to acquire (or release) a distributed cluster-wide lock on the instance.

In our *Counter* sample class, this would mean capturing the execution of the *MONITOR_ENTER* and *MONITOR_EXIT join points*, e.g. the *synchronized* blocks that are guarding *'this'*, in the *increment()* and *waitFor(int)* methods.

5.2 Locking and Unit of Work

It is worth mentioning that locking allows implementing certain optimizations. For example, the *lock manager* can be coordinated with the *object manager* and can use synchronization boundaries to create a *Unit of Work*. Then the *object manager* can batch all the field modifications that happen within the specific *Unit of Work* and propagate them to the other nodes in the cluster when the *Unit of Work* is completed upon distributed and local lock release.

We can define several lock types with different semantics:

- *Write lock* has the same semantics as regular *Java* synchronization and allows at most one single thread to acquire a lock.
- *Read lock* reconciles object changes but does not require blocking of the execution, e.g. multiple threads can acquire the same lock.
- *Concurrent lock* is a looser form of *write lock*. Multiple threads can be inside a concurrent lock at the same time, and can all write to the data, but, only the changes made by the last thread to exit the block actually take effect. This is nondeterministic and thus is generally used only with considerable caution.

Locks can be configured using the same configuration mechanisms as *Roots* and *Clustered* objects.

The *lock manager* could apply several optimizations, such as escape analysis and reentrant locking. In a distributed system, *greedy locking* can give significant performance advantages. In this case, the lock is owned by the particular node until it is requested by another node and then it is transferred to that node. This allows us to reduce the number of calls to the *lock manager*.

5.3 Wait/Notify

Apart from locking (*synchronized*), the two most important primitives that are missing from the *JMM*, are *wait* (waiting for a lock to be released) and *notify/notifyAll* (wake up other threads to contend for a lock that was being held). The semantics of these primitives also need to be maintained across the cluster.

In order to make this transparent for the target application we can define the following *advice* that will use the *ClusterManager* to delegate to the *lock manager* that is implementing the necessary execution semantics.

```
void around(Object o) :
   isClusteredTarget(o) &&
   call(void Object.wait()) {
  ClusterManager.objectWait(o);
}

void around(Object o, long t) :
   isClusteredTarget(o) &&
   call(void Object.wait(long)) &&
   args(t) {
  ClusterManager.objectWait(o, t);
}

void around(Object o, long t, int n) :
   isClusteredTarget(o) &&
   call(void Object.wait(long,int)) &&
   args(t,n) {
  ClusterManager.objectWait(o, t, n);
}

void around(Object o) :
   isClusteredTarget(o) &&
   call(void Object.notify()) {
  ClusterManager.objectNotify(o);
}

void around(Object o) :
   isClusteredTarget(o) &&
   call(void Object.notifyAll()) {
  ClusterManager.objectNotifyAll(o);
}
```

5.4 Distributed Method Invocations

In some cases, a distributed system does not have state to distribute, but may be required to distribute actions. One of the most common scenarios is when *listeners* (for example *ActionListener* in *Java Swing* [12]) or *observers* (for example in the *Observer Pattern* [13]) need to be notified when some event has been triggered. Because those events could happen anywhere in the cluster, *listeners* (or *observers)* need to be notified on every node.

For these scenarios, *Terracotta* has introduced a special abstraction called *Distributed Method Invocation* (*DMI*). This simply means that if method, that is marked for *DMI*, is called on one cluster node, all other cluster nodes will also invoke the exact same method invocation (with the exact same parameters).

Here we can mark such methods with special *@DMI* annotation. We can then write a *pointcut* that picks out all *join points* where one of these methods is executed and then finally bind an *after advice*, which delegates the invocation call to the *object manager,* to this *pointcut*.

```
pointcut distributedMethodInvocation(Object o) :
   isClustered() &&
   execution(@DMI * *.*(..)) &&
   this(o);

after(Object o) :
   distributedMethodInvocation(o) {
  String name = thisJoinPointStaticPart.getSignature().toLongString();
  ClusterManager.distributedMethodInvocation(
    o, name, thisJoinPoint.getArgs());
  }
```

This could be further extended to allow different semantics. For example, the default semantics for a *DMI* is that they are invoked asynchronously, but it could also be useful to provide a synchronous implementation, or to maintain a *Unit of Work* and wait while all invocations are propagated before releasing the lock.

It is important to understand that *DMI* has nothing to do with managing correctness and coherence of state in the cluster, but only to trigger a cluster-wide action (behavior).

6. DISCUSSION

So far we have only used standard *AspectJ* features, which have worked nicely for our simple *Counter* example. However, there are several limitations that prevent us from using a plain *AspectJ*-based solution for real-world applications. We already mentioned the problems with the *TransparentAccess ITD*, that required us to generate custom code for each advised class, but at least we were able to work around it with reflection-based code.

Other critical *AspectJ* limitations include restrictions on instrumenting *java.** and *javax.** classes and the absence of *pointcuts* for array creation and for access to the array elements. For the latter issue, there is experimental support for array creation (which can be enabled with *-Xjoinpoints:arrayconstruction* compiler flag), but there is still no support for array access.

It is also worth mentioning the load-time weaving overhead for *AspectJ*. In some of our tests, application startup time with *AspectJ* load-time weaving was from two to three times slower and memory overhead was about sixty times bigger (600% vs. 10% overhead after *JVM* garbage collection), when comparing with similar transformations done with either the *Terracotta* runtime or the *AspectWerkz AOP* engine.

7. CONCLUSIONS

In this paper we have shown how *AOP* technology can be used to implement the cross-cutting concern of transparent clustering for any arbitrary *Java* code. The described approach has been initially implemented and proven on real applications in *Terracotta*'s *Distributed Shared Objects (DSO)* product. The use of *AOP* technology allowed us to focus on implementation details for the services used in the clustering runtime and transparently weave in the glue code into the application in order to get high-availability, scalability and failover.

This approach to clustering provides great benefits to the end users, who can focus solely on implementing their business logic and still get the benefits of clustering without polluting their code with traditional API-based clustering.

As we have shown, limitations of the existing *AOP* frameworks for the *Java* platform give enough justification to implement a hybrid solution, but we hope that in the future we might be able to use a general purpose *AOP* language, such as *AspectJ*. While being a great language and compiler, load-time weaving in *AspectJ* is still suffering from performance problems (at start-up time) and memory issues. These need to be addressed in future versions of *AspectJ*. Also, to completely cover scenarios like those described in this paper, *AspectJ* would have to support advising access to array elements and open up the instrumentation pipeline to allow plugging-in custom optimizations for *inter-type declarations*, e.g., based on code-generation.

We believe that further evolution of the standard *AOP* tools would make it possible to implement reusable aspects for application clustering, failover and high-availability.

8. ACKNOWLEDGEMENT

First we would like to thank Steve Harris, the architect and lead developer of *Terracotta DSO*, as well as Saravanan Subbiah, Tim Eck and Orion Letizi from the *DSO* team for sharing their technical excellence and for being great and humble team players. Without them there would be no *Terracotta DSO*. We would also like to thank Ron Bodkin and Dean Wampler for feedback on this paper.

9. REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In 1997 European Con I. on Object-Oriented Programming (ECOOP '97), pages 220-242. Springer Verlag, 1997.

[2] Java Memory Model (JCP 133). http://jcp.org/en/jsr/detail?id=133

[3] Java Language Specification. http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[4] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In Proceedings of the 3rd international conference on Aspect-oriented software development, 2005

[5] E. Kuleshov. Using ASM toolkit for bytecode manipulation. http://www.onjava.com/lpt/a/5250

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. Getting Started with AspectJ. Communications of the ACM, 44(10):59–65, October 2001.

[7] Terracotta. http://terracottatech.com/

[8] Java Annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

[9] Java Enterprise Edition. http://java.sun.com/javaee/technologies/javaee5.jsp

[10] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern, http://www.martinfowler.com/articles/injection.html

[11] R. Johnson. Introduction to the Spring Framework, http://www.theserverside.com/articles/article.tss?l=SpringFramework

[12] Java Swing. http://en.wikipedia.org/wiki/Swing_(Java)

[13] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series. 1995.

[14] Java 5 concurrency libraries. http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/CountDownLatch.html

[15] Java Object Serialization Specification version 1.5.0.

http://java.sun.com/j2se/1.5.0/docs/guide/serialization/spec/serialTOC.html

[16] SAN – Storage Area Network. http://en.wikipedia.org/wiki/Storage_area_network

[17] Extensible Markup Language (XML). http://www.w3.org/XML/

[18] Java Reflection API. http://java.sun.com/docs/books/tutorial/reflect/index.html