Concept and Architecture Guide

About Terracotta Documentation

This documentation is about Terracotta DSO, an advanced distributed-computing technology aimed at meeting special clustering requirements.

Terracotta products without the overhead and complexity of DSO meet the needs of almost all use cases and clustering requirements. To learn how to migrate from Terracotta DSO to standard Terracotta products, see Migrating From Terracotta DSO. To find documentation on non-DSO (standard) Terracotta products, see Terracotta Documentation. Terracotta release information, such as release notes and platform compatibility, is found in Product Information.

Release: 3.6
Publish Date: November, 2011 **Documentation Archive »**

Concept and Architecture Guide

Introduction

This document describes Terracotta's basic concepts and architecture.

Terracotta is an enterprise-class, open-source, JVM-level clustering solution. JVM-level clustering simplifies enterprise Java by enabling applications to be deployed on multiple JVMs, yet interact with each other as if they were running on the same JVM. Terracotta extends the Java Memory Model of a single JVM to include a cluster of virtual machines such that threads on one virtual machine can interact with threads on another virtual machine as if they were all on the same virtual machine with an unlimited amount of heap.

The programming model of applications clustered using Open Terracotta is the same or similar to that of an application written for a single application. There is no developer API specific to Open Terracotta. Terracotta uses bytecode manipulation—a technique used by many Aspect-Oriented Software Development frameworks such as AspectJ and AspectWerkz—to inject clustered meaning to existing Java language features.

For an introduction to Terracotta, see What Is Terracotta.

Core Terracotta Concepts

This section discusses the core concepts and architecture of Terracotta.

Roots

A "root" is the top of a clustered object graph as declared in the Terracotta configuration. Object sharing in Terracotta starts with these declared roots. Any object that is reachable by reference from a Terracotta root becomes shared by Terracotta which means that it is given a cluster-wide object ID and its changes are tracked by Terracotta and made available to the cluster.

Roots are declared in the Terracotta configuration by name. A root declaration in the Terracotta configuration must either declare an explicit field name in a java class or a field expression. Any field may be declared a root and the same root may be assigned to more than one field.

Fields that are declared as roots assume special behavior. This is the one area of Terracotta that diverges significantly from regular Java semantics.

- The first time a root is assigned by the first JVM that assigns it, the root is "created" in the Terracotta cluster and the object graph of the object assigned to the root field becomes the object graph of that root.
- Once assigned, the value of a root field may never be changed (except as noted in Literal Root Mutation). After the first assignment, all subsequent assignments to fields declared to be that root are ignored.
- Configuring a Terracotta root introduces instrumentation to the root's containing class. The effect is that classes which are not explicitly configured to be instrumented by Terracotta are automatically instrumented if they contain a field which is declared a Terracotta root. See the section on instrumentation for an explanation of class instrumentation.

Because roots are durable beyond the scope of a single JVM's lifecycle, we refer to them as "superstatic." This is true no matter what the scope modifier of the root field. If a static field is declared to be a root, that static field will assume a superstatic lifecycle. Likewise, if an instance field is declared to be a root, that instance field will assume a superstatic lifecycle. This can have a significant effect on code written with respect to root objects.

While the reference of the root field itself cannot be changed (except as noted in Literal Root Mutation), the object graph of that root object can. Typically, data structures like Maps, Lists, and other Collections are chosen as root objects—although any portable object can be declared a root. The contents of those data structures can change. It is just the assignment of the root field that cannot be changed.

Literal Root Mutation

The exceptions to this rule are root fields that contain Java primitives (see the description of literal values in the Clustered Objects section below). The value of primitive literal roots can be freely mutated (assignments are never ignored).

Further reading:
Configuring Roots

Clustered Objects

When a root field is first assigned, the Java object assigned to that field and all objects reachable by reference from that object become clustered objects. This means that each object is given a unique cluster-wide ID and all its field data is added to the current Terracotta transaction. When the Terracotta transaction is committed, all of the object data is sent to the Terracotta server. All changes to clustered objects must be made within the context of a Terracotta transaction.

Object data is composed of the values of reference and literal fields. Reference fields are references to other Java objects. Literal fields are data types that Terracotta considers not to be reference types. Terracotta literals are similar to (but not exactly the same as) Java primitives (see Literals).

When an object becomes referenced by a clustered object, it and all of the objects in its object graph become clustered objects themselves. Once a Java object becomes clustered, it is always a clustered object until it becomes garbage collected by the distributed garbage collector. All the changes to clustered objects are recorded to the current Terracotta transaction. For literal fields, the new value of that literal field is recorded in the Terracotta transaction. For reference fields, the cluster object id of the object to which the field refers is recorded in the Terracotta transaction.

There is guaranteed to be one and only one instance of a particular clustered object per ClassLoader per JVM. All changes made to a clustered object anywhere in the cluster will be applied to the same instance of the clustered object for its ClassLoader's context. This is to preserve object identity. See this series of blog entries for more information on Terracotta and object identity:

- Object Identity, Part One
- Object Identity, Part Two
- Object Identity, Part Three

Clustered objects must be portable. An attempt to cluster a non-portable object will result in a runtime exception.

Virtual Heap

Terracotta virtual heap allows arbitrarily large clustered object graphs to fit within a constrained local heap size in client virtual machines. The local heap has a "window" on a clustered object graph. Portions of a clustered object graph are faulted in and paged out as needed. Virtual heap is similar in concept to virtual memory and is sometimes referred to as "Network Attached Memory."

Clustered objects are lazily loaded from the server as they are accessed by a client JVM. This happens by injecting augmented behavior around the GETFIELD bytecode instruction that checks to see if the object referred to by that field is currently instantiated on the local heap. If it isn't on the local heap yet, the client will request the object from the server, instantiate the object on the local heap, and ensure that the field refers to that newly instantiated object.

Conversely, less frequently used objects may be transparently purged from local heap by Terracotta, subject to the whims of the local JVM garbage collector. The amount of clustered object data kept in local heap is determined at runtime. The Terracotta Virtual Memory Manager actively sets references to objects that have fallen out of the cache to null so that they may become eligible for local garbage collection. Because clustered objects may be lazily loaded, purged objects will be transparently retrieved from the server as references to them are traversed.

For more information on how certain object types and values are loaded into memory, see the Clustered Data Structures Guide.

Virtual Memory Manager

The Terracotta Virtual Memory Manager (VMM), also referred to as a cache manager, is designed to allow the constrained heap of your application's JVM to view and manipulate an arbitrarily large clustered object graph. This can allow each JVM in your cluster to work on a window of your clustered data, even if your clustered data is very large. In general the VMM clears references from the heap memory of both Terracotta server instances and clients, making those objects available for JVM garbage collection.

The VMM is not a substitute for balancing data load across JVMs. Scanning very large object graphs in one JVM should be avoided because of the negative impact this has on performance. To do more work and address more data, add JVMs as necessary to work on new partitions. A good example of this virtuous design pattern is a sticky load-balancer in front of a cluster of application servers where the load balancer partitions user requests by session across the available JVMs. If large object graphs are needed, use strategies such as partial loading, striping, and partitioning to reduce the load on any one JVM.

It is also important to understand how the Terracotta VMM functions with respect to clustered object graphs that do not fit in the local heap of a clustered JVM—specifically if those object graphs contain very large logically managed objects like certain collections. The VMM is able to prune the reference fields of most objects by forcing them to be null, thereby freeing the referenced objects for garbage collection by the local JVM garbage collector. Logically managed objects are an exception and the VMM does not support partial faulting of every managed object type. ConcurrentHashMap, HashMap, Hashtable, LinkedHashMap, and Java arrays may be partially faulted into the local heap of a client JVM, while other logically managed types are faulted into heap in their entirety. See the Clustered Data Structures Guide for more information on how these data structures are handled in Terracotta.

In an application with heavy usage of heap memory, a well-tuned VMM can keep a JVM's heap from becoming full and prevent a Terracotta client or server instance from failing. For more information on tuning VMM, see the Tuning Terracotta.

Distributed Garbage Collection

Every Terracotta server instance runs a garbage collection algorithm to determine which clustered objects are no longer referenceable and can safely be removed both from the memory and the disk store of the server instance. This garbage collection process, known as the Distributed Garbage Collector (DGC), is analogous to the JVM garbage collector but works only on clustered object data in the server instance. The DGC is *not* a distributed process; it operates only on the *distributed* object data in the Terracotta server instance in which it lives. In essence, the DGC process walks all of the clustered object graphs in a single Terracotta server instance to determine what is currently not garbage. Every object that *is* garbage is removed from the server instance's memory and from persistent storage (the object store on disk). If garbage objects were not removed from the object store, eventually the disk attached to a Terracotta server instance would fill up.

A clustered object is considered garbage when it is not reachable from any root (it is *not* part of a root's object graph) AND it is not resident in any client's heap. Objects that satisfy only one of these conditions are safe from a DGC collection. Root objects are never marked as garbage.

To view and manipulate clustered data, your clustered application JVMs will at various times have some of the objects in the clustered object graph(s) physically instantiated on heap. For a variety of reasons, Terracotta server instances know which objects are currently on the physical heap of each connected client JVM. As long as an object remains instantiated on the heap of any clustered JVM, it cannot safely be garbage collected, even if the DGC algorithm determines that it is not referenceable. This is because the object may, through the course of your application at some time in the future, rejoin a clustered object graph and, therefore, no longer be garbage. Only after all instances of a clustered object have been garbage collected by the JVM garbage collectors in any connected client JVM (that has had an instance of that object on heap) can the DGC safely remove a garbage object.

Types of DGC

There are two types of DGC: Periodic and inline. The periodic DGC is configurable and can be run manually (see below). Inline DGC, which is an automatic garbage-collection process intended to maintain the server's memory, runs even if the periodic DGC is disabled.

Running the Periodic DGC

The periodic DGC can be run in any of the following ways:

- Automatically – By default DGC is enabled in the Terracotta configuration file in the <garbage-collection> section. However, even if disabled, it will run automatically under certain circumstances when clearing garbage is necessary but the inline DGC does not run (such as when a crashed returns to the cluster).
- run-dgc shell script – Call the run-dgc shell script to trigger DGC externally.
- JMX – Trigger DGC through the server's JMX management interface.
- Terracotta Administrator Console – Click the **Run DGC** button to trigger DGC.

Tuning the periodic DGC to adapt it to application-data characteristics is normal part of optimizing a cluster. See the Tuning Terracotta for more information on tuning the DGC.

Monitoring the DGC

DGC events (both periodic and inline) are reported in a Terracotta server instance's logs. DGC can also be monitored using the Terracotta Administrator Console and its various stages. See the Terracotta Administrator Console for more information.

Further reading:

- Tuning Terracotta – Instructions on how to tune the DGC
- Configuration Guide and Reference – Instructions on how to configure DGC options
- Terracotta Tools Catalog – See the section on the run-dgc shell script in the Tools Catalog document in
- Terracotta Administrator Console – How to execute DGC operations and view current status, history, and statistics
- JMX Guide – The DGC JMX interface

Locks

Locks in Terracotta perform two duties: to coordinate access to critical sections of code between threads and to serve as boundaries for Terracotta transactions. Terracotta locks are analogous to synchronization in Java. Clustered locking is injected into your application based on the locks section of the Terracotta configuration. Each lock configuration stanza uses a regular expression that matches a set of methods. The locking that occurs in that set of methods is controlled by configuration options specified in the lock configuration stanza.

No cluster behavior will occur in classes that have not been included for instrumentation in the Terracotta configuration. More on instrumentation can be found below.

Summary of Terracotta Locks

Before we begin, let's go over a brief summary of locking in Terracotta.

No cluster lock behavior will occur ...

- if the object is shared *after* it is locked. Instead, an error occurs. See an example in the following gotcha.
- in methods that have no synchronization, unless you configure auto-synchronize to be true.
- if you aren't synchronizing on a clustered object.
- if you synchronize on an object that isn't clustered yet, even if it becomes clustered within the scope of your synchronization. The object must be clustered before you synchronize on it or no cluster behavior will occur.
- in methods that do not match a lock configuration stanza. Of course, you do not need to configure locking for code that is already addressed in the configuration of a Terracotta Integration Module (TIM) that you are using.

If you manipulate a clustered object ...

- outside the scope of a clustered lock, you will get an error.
- in code that is not instrumented by Terracotta, the changes to that object will be invisible to Terracotta. As a result, the clustered object may be in an inconsistent state. Therefore, you MUST ensure that you instrument all classes that manipulate clustered objects. See Gotchas: Uninstrumented Access for more details.

ReentrantReadWriteLock

You can use `java.util.concurrent.locks.ReentrantReadWriteLock` in place of synchronization in your code. You do not need additional configuration or annotations to give `ReentrantReadWriteLock` clustered behavior. The only requirement is that any `ReentrantReadWriteLock` requiring clustered behavior must be a part of the clustered graph. A `ReentrantReadWriteLock` that is not a part of the clustered graph *imparts local locking semantics only*.

Terracotta locking vs. standard Java locking

As with everything Terracotta, Terracotta locking is synonymous with locking in the standard JVM. Generally speaking, you do not have to concern yourself with the differences of locking in a single-JVM context compared to locking with Terracotta in a multiple-JVM context. This is true except for the following cases:

Case 1: Writing to an Object without a Lock Present

Terracotta will not let you write to a shared object without a lock present. Unlike normal Java, which will allows this, Terracotta throws an Exception. You may find it cumbersome at first to have to make sure your locking is set up correctly before your application will run with Terracotta, however in the long-run this precaution will save you from having concurrency problems that are difficult if not impossible to find.

These problems will manifest as corrupt data or incorrect application behavior, and will likely only show up sporadically when your application is in production. This is **not** the time you want to find out you have a data corruption or application behavior bug.

Case 2: Standard Java synchronization and Read/Write Locking Semantics

Standard Java synchronization allows only mutual exclusion. Until Java 1.5, and the introduction of the java.util.concurrent classes, there was no standard way to enable read and write locking.

But standard Java synchronization can be converted to have read/write locking semantics. With Terracotta, you can convert a synchronized block into a read lock. As before, however, if your application attempts to perform a write while only holding a read lock, you will get an UnlockedSharedObjectException, which prevents your application from writing to a shared object while only holding a read lock.

This is a benefit: You do not want to find these kinds of bugs in your application in production.

Case 3: Forced Synchronization

With Terracotta, you must **always** have a lock present to update the field of a shared object, as mentioned in this case. This is a *good thing*. However, if you are integrating Terracotta into an application that uses code lacking synchronization, and it is undesirable or impossible to update that code to have proper locking, Terracotta provides two options:

1. Named Locks
2. Auto-synchronized

Because these options can have a severe impact on performance, they should **not** be relied on to avoid refactoring your code. They are intended for situations where code cannot be refactored, such as when third-party libraries are used.

Named Locks can be introduced on any method. A Named Lock is a global lock. This means that any object instance in the cluster that executes a method with a Named Lock will prevent any other instance - across the cluster - from executing that method until it is finished.

Auto-synchronized is similar to Named Locks, except it is more granular. Auto-synchronized can be used to add synchronization to a method at run time, and then auto-lock on that synchronized method. It has the same effect that changing the code, by adding a synchronized keyword to the method, would have. An auto-synchronized method may be read or write locked.

A Short Java Locking Primer

As a refresher course on Java synchronization, consider the following code:

`Person` v1:
This code is not thread safe.

```
package example;

public class Person {
  private String name;

  public void setName(String theName) {
    name = theName;
  }

  public String getName() {
    return name;
  }
}
```

This code is not thread safe because it doesn't synchronize access to data. This means that, if multiple threads execute this code concurrently on the same `Person` object, there are no guarantees about the state of that person object; the value of `Person.name` is indeterminate. Obviously, this isn't ideal for a multi-threaded application. There are a number of things you can do to make this code thread safe.

Add Locking Inside The Class

`Person` v2:
This code is *thread safe.*

```
package example;

public class Person {
  private String name;

  public synchronized void setName(String theName) {
    name = theName;
  }

  public synchronized String getName() {
    return name;
  }
}
```

For purposes of example, we are discussing a simple bean-style class. However, in actual practice, you don't need to synchronize every method of a bean-style class like this in order to get thread safety. Often, synchronization will actually happen in a more coarse-grained fashion. See the discussion below on coarse grained and fine grained locking.

Putting the `synchronized` modifier on a method is actually just shorthand for the following (the compiled bytecode ends up looking the same):

`Person` v3:
This is equivalent to synchronized methods.

```
package example;

public class Person {
  private String name;

  public void setName(String theName) {
    synchronized (this) {
      name = theName;
    }
  }

  public String getName() {
    synchronized (this) {
      return name;
    }
  }
}
```

Another alternative is to lock on another object. This has the advantage of entirely encapsulating the locking within the class:

`Person` v4:
This is an encapsulated private lock.

```
package example;

public class Person {
  private final Object myLock = new Object();
  private String name;

  public void setName(String theName) {
    synchronized (myLock) {
      name = theName;
    }
  }

  public String getName() {
    synchronized (myLock) {
      return name;
    }
  }
}
```

It's important to note here that you don't need to synchronize on the object you are changing in order to be thread safe. You just need to synchronize *on the same object*, whatever that object is, before you manipulate (read OR write) the same data. This means that you can view or modify whatever you want within the scope of a lock as long as you acquire the same lock wherever you view or modify the data in question.

Add Locking Outside The Class

The following is an example of unencapsulated locking.

```
package example;

public class PersonHandlerA {
  private final Person person;

  public PersonHandlerA(final Person thePerson) {
    this.person = thePerson;
  }

  public void doStuff() {
    synchronized (person) {
      System.out.println("The name is: " + person.getName());
    }
  }
}




// ... defined in another class file...
package example;

public class PersonHandlerB {
  private final Person person;

  public PersonHandlerB(final Person thePerson) {
    this.person = thePerson;
  }

  public void doOtherStuff(final Person thePerson) {
    System.out.println("Changing person's name to Hildegard...\n";
    synchronized (person) {
      person.setName("Hildegard");
    }
  }
}
```

This code is thread safe, but the locking isn't encapsulated within the class that holds the data. However, it has the advantage that the operation that occurs after reading `person.name` (namely, printing it to `System.out`) is guaranteed to have an up-to-date view of that value, since both the name read operation and the print operation happen within the scope of the lock.

To make this code even better, the handler classes would use their own objects to lock on rather than locking on the person object, since other code elsewhere (e.g., the `Person` class itself) might lock on that object in ways that the handler classes don't expect.

Use The Higher-Order `java.util.concurrent` Libraries

In Java 1.5, the excellent concurrency library, `java.util.concurrent` was added to give developers a standard set of concurrency tools. These tools implement a lot of useful, higher-order concurrency algorithms that you previously had to implement yourself using the Java language concurrency primitives. Terracotta supports many of these useful constructs out of the box.

Let's look at our locking example using `java.util.concurrent.locks.ReentrantReadWriteLock`:

`Person` v5:
Locking using `ReentrantReadWriteLock`.

```
package example;

import java.util.concurrent.locks.ReentrantReadWriteLock;
import java.util.concurrent.locks.ReentrantReadWriteLock.ReadLock;
import java.util.concurrent.locks.ReentrantReadWriteLock.WriteLock;

public class Person {
  private final ReentrantReadWriteLock myLock = new ReentrantReadWriteLock();
  private final ReadLock readLock = myLock.readLock();
  private final WriteLock writeLock = myLock.writeLock();
  private String name;

  public void setName(final String theName) {
      writeLock.lock();
      name = theName;
      writeLock.unlock();
    }

  public String getName() {
    String rv;
    readLock.lock();
    rv = name;
    readLock.unlock();
    return rv;
  }
}
```

This code has the advantage of different concurrency levels: the read operation uses a read lock and the write operation uses a write lock. This allows multiple threads through the `getName()` method concurrently, as long as no method has acquired the write lock. Likewise, no thread can acquire the write lock in the `setName()` method unless no thread holds the read lock. If your application does a lot of concurrent reading and not as much concurrent writing, you will have much more parallelism in your application.

Now, let's introduce Terracotta and how it works with the locking that's already present in your code to make your code clustered.

Autolocks

Locking in Terracotta is configured on a method level using "autolocks." The configuration file has a "locks" section which can contain zero or more lock stanzas. Methods that match an autolock stanza are augmented by Terracotta to acquire a cluster-wide lock on a clustered object wherever there is synchronization on that object.

For a complete reference guide to Terracotta locking configuration, see the Configuration Guide and Reference.

To add Terracotta locking to `Person` v2, v3, and v4 you would add the following to your Terracotta configuration file:

```
<locks>
  <autolock auto-synchronized="false">
    <method-expression>* example.Person.*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
</locks>
```

The method or methods that this autolock stanza applies to is determined by the "method-expression" element. The method-expression is declared using the AspectWerkz pattern selection language. Using this pattern selection language, you can apply an autolock configuration stanza to a precise set of methods in your code. For more information, see the AspectWerkz Pattern Language guide.

The method expression in the example above, `* example.Person.*(..)`, matches all methods in the `example.Person` class. The leading `*` in the expression matches all possible return types. The ending .* matches all methods in the `example.Person` class. The `(..)` segment matches all possible parameters, including no parameters.

The methods whose signature matches the method-expression in this example will be augmented with clustered locking, allowing Terracotta to find all of the `synchronized` blocks in the method (including the implied `synchronized` block surrounding a `synchronized` method) and add cluster-wide synchronization behavior to them.

Cluster-wide synchronization only happens if the object that is being synchronized on is already a clustered object.

To add Terracotta locking to the person handlers that lock `Person` objects outside the `Person` class, you would add the following to your Terracotta configuration file:

```
<locks>
  <autolock auto-synchronized="false">
    <method-expression>* example.PersonHandlerA.*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
  <autolock auto-synchronized="false">
    <method-expression>* example.PersonHandlerB.*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
</locks>
```

Since `Person v1` has no synchronization, a regular autolock stanza will have no effect. In such cases, you can add Terracotta locking to `Person v1` by using the "auto-synchronized" attribute of the autolock element like so:

```
<locks>
  <autolock auto-synchronized="true">
    <method-expression>* example.Person.*(..)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
</locks>
```

Autolocking with Automatic Synchronization

Autolocked methods can be configured to auto-synchronize on the object in which the method is declared. This is useful if you don't have synchronization in an object that you want to cluster. It is equivalent to applying "`synchronized`" modifier to the autolocked methods or, equivalently, to wrapping the method in a "`synchronized (this) {...}`" block. Using the `auto-synchronized="true"` attribute tells Terracotta to insert locking where there isn't any. This effectively turns `Person v1` into `Person v2`.

USE AUTO-SYNCHRONIZED SPARINGLY, IF AT ALL. This feature is designed to allow Terracotta to be used on code that cannot be changed to synchronize properly—for example, third-party libraries for which you don't have the source code.

How Autolocking Works

Autolocking in Terracotta works by adding a request for a clustered object's lock from the server around the MONITORENTER bytecode instruction and releasing that lock around the MONITOREXIT bytecode instruction. You can think of autolocks as an extension of a method's existing Java synchronization to have a cluster-wide meaning.

Autolocks are fine-grained locks in the sense that the lock requested is the lock for a particular clustered object. The lock acquired is constructed based on the unique cluster ID of the object being synchronized on. In autolocked methods, if the object being synchronized on is not a clustered object, then only the regular, local JVM lock is acquired. Likewise, autolocks applied to methods that have no synchronization in them will have no clustered locking.

Named Locks

In addition to autolocks, Terracotta provides a feature called "named locks" where method access is protected by acquiring a lock of a particular name. A thread that attempts to execute a method that matches a named lock stanza must first acquire the lock of that name from the Terracotta server. Named locks are very coarse-grained and should only be used when autolocks are not possible.

USE OF NAMED LOCKS IS STRONGLY DISCOURAGED

USING NAMED LOCKS CAN HAVE A SEVERE IMPACT ON YOUR APPLICATION'S PERFORMANCE. The named lock feature is intended for third-party code which you cannot change. Be sure to thoroughly investigate your requirement for named locks before introducing this feature into your configuration.

The following example shows how a read lock named "lockOne" is configured on all get methods in the package pkgb. The leading * indicates that the target get methods can have any return type, while (int) restricts the lock to methods with an argument of type int.

```
<named-lock>
   <lock-name>lockOne</lock-name>
   <method-expression>* com.foolib.pkgb.get*(int)</method-expression>
   <lock-level>read</lock-level>
</named-lock>
```

Multiple named locks with the same name behave as the same lock.

Lock Level

One of the configuration options in the lock configuration stanza is the lock "level." The lock level lets you specify different levels of concurrency on a per lock basis. Unlike Java, Terracotta locks come in four levels: write, synchronous-write, read, and concurrent.

While omitting read locks may seem harmless because "dirty" reads can succeed, this practice is highly discouraged because it introduces the danger of threads reading stale data. Using read locks when reading shared data ensures that the data is always fresh. Up-to-date data *cannot be guaranteed* if read locks are omitted.

Write Locks

Write locks act like regular Java locks (mutexes). They guarantee that only one thread in the entire cluster can acquire that lock at any given time.

The locking examples we have seen so far use write locks.

Read Locks

Read locks allow multiple threads to acquire the lock at a time, but those threads are not allowed to make any changes to clustered objects while holding the read lock. No thread may acquire a write lock if any thread holds a read lock. No thread may acquire a read lock if any thread holds a write lock.

If a thread attempts to make modifications while holding a read lock, an exception will be thrown. Likewise, if a thread currently holds a read lock and crosses another lock acquisition boundary on the same lock (e.g., a synchronization on the same object), but that new lock acquisition boundary is configured to be a write lock, an exception will be thrown.

Read locks offer a significant performance advantage over write locks when multiple threads concurrently execute code that does not modify any clustered objects.

To configure different read and write concurrency levels in v2 and v3, add the following to your Terracotta configuration file:

```
<locks>
  <autolock auto-synchronized="false">
    <method-expression>void example.Person.setName(java.lang.String)</method-expression>
    <lock-level>write</lock-level>
  </autolock>
  <autolock auto-synchronized="false">
    <method-expression>java.lang.String example.Person.getName()</method-expression>
    <lock-level>read</lock-level>
  </autolock>
</locks>
```

The `setName()` method is configured as a write lock and the `getName()` method is configured as a read lock.

If you use `ReentrantReadWriteLock`, you don't need to specify locking in your configuration—Terracotta supports its locking out of the box. However, the `ReentrantReadWriteLock` that you lock on must already be a clustered object. As a consequence, `Person` v5 requires no Terracotta configuration to enable clustered locking.

Synchronous-Write Locks

Synchronous-write locks add the further guarantee that the thread holding the lock will not release the lock until the changes made under the scope of that lock have been fully applied and ACKed by the server.

Concurrent Locks

Concurrent locks are, essentially, just transaction boundaries. They do not provide any locking at all. As such, they are only safe to use when write-write conflicts between threads are acceptable. They are a legacy feature that are now meant to be used only within the Terracotta libraries.

CONCURRENT LOCKS ARE FOR ADVANCED USE ONLY AND STRONGLY DISCOURAGED. THIS FEATURE MAY BE REMOVED WITHOUT NOTICE.

Clustered `wait()` And `notify()`

A thread that holds the clustered lock for a clustered object may call `wait()` on that object. This will cause the thread to commit the current Terracotta transaction, release the clustered lock, and pause execution. Any other thread in the cluster that holds the clustered lock on that object may call `notify()` on that object. The Terracotta server will select a waiting thread (in the case of a `notify()` call) or the set of all waiting threads (in the case of a `notifyAll(` call) and ensure that the appropriate waiting threads throughout the Terracotta cluster are notified. A thread that is notified will contend for the clustered lock and resume execution when granted that lock.

When Object Changes Are Visible Across The Cluster

Terracotta extends the Java Memory Model to have a cluster-wide meaning. As such, the same rules apply to object change visibility between threads across the Terracotta cluster in different JVMs as they do in the same JVM.

All changes made under a particular lock are made visible to a thread acquiring the lock. E.g.:

```
Node 1 - Thread 1 : acquires lock on object 1234
Node 1 - Thread 1 : makes changes to object 1234 and object 4567
Node 1 - Thread 1 : release lock on object 1234
==> changes are recorded as Transaction ID 1
Node 2 - Thread 4 : acquires lock on object 1234
==> all changes made under Transaction ID 1 are applied and visible to Node 2 - Thread 4

Node 3 - Thread 2 : requests lock on object 1234
Node 3 - Thread 2 : is "parked" waiting for lock on object 1234
Node 2 - Thread 4 : makes changes to object 7890
Node 2 - Thread 4 : releases lock on object 1234
==> changes are recorded as Transaction ID 2
Node 3 - Thread 2 : acquires lock on object 1234.
==> all changes made under Transaction ID 1 are applied and visible to Node 3 - Thread 2
==> all changes made under Transaction ID 2 are applied and visible to Node 3 - Thread 2
```

In this example, when Node 3 - Thread 2 acquires the lock all changes recorded in Transaction ID 1 (changes to object 1234 and object 4567) and Transaction ID 2 (changes to object 7890) are visible. Using a read lock when Node 3 - Thread 2 reads the data in the changed objects ensures that the data is fresh. While not using a read lock (doing a "dirty" read) is technically possible, it is highly discouraged because of the danger of reading stale data.

For more information on transactions, see Transactions.

Greedy Locks

Terracotta makes certain runtime optimizations regarding locking that can significantly improve the performance of locking. One of those optimizations is known as "greedy locks."

When a thread acquires a lock that is not in contention in any other JVM in the cluster, the Terracotta server will grant the lock to the requesting JVM as a "greedy lock" such that subsequent lock requests by that JVM will be granted locally without a network call to the Terracotta server. This has the effect of making greedy lock acquisitions happen at memory, rather than network speed.

Coarse Grained And Fine Grained Locking

You don't have to synchronize every method of your code in order to be thread safe. Often, you only need to obtain a single lock around a set of operations. The person handler example code above shows synchronization can occur around multiple operations using slightly coarser locking:

```
package example;

public class PersonHandlerA {
  private Object myPrivateLock;
  private Person person;

  // ... some initialization code...

  public void makeSomeChangesToPerson(String firstName,
                                      String lastName,
                                      int age) {
    synchronized (myPrivateLock) {
      person.setFirstName(firstName);
      person.setLastName(lastName);
      person.setAge(age);
    }
  }

  public void printPersonRecord() {
    final StringBuffer output = new StringBuffer();

    synchronized (myPrivateLock) {
      output.append("First Name: " + person.getFirstName() + "\n";
      output.append("Last Name : " + person.getLastName() + "\n";
      output.append("Age       : " + person.getAge() + "\n";
    }

    System.out.print(output);
  }
}
```

This is coarser locking than locking inside every method of the `Person` class. Plus, as long as no other class reads or writes the `Person` class, it guarantees a consistent view of the person object, since you are making all changes and performing all reads atomically within the synchronization.

Another example of coarser-grained locking is using Terracotta to cluster HTTP sessions. Terracotta automatically acquires a lock on the user session at the beginning of the request handling cycle and releases it at the end. As a result, access to user session data is all protected by that widely-scoped lock. Therefore, you don't need any synchronization in your application code that manipulates session objects within the scope of a normal request handling cycle.

In idealized code, it looks something like this:

```
HttpServletSession session = sessions.getSession(userSessionId);
synchronized (session) {
  servlet.service(request, response);
}
```

In this simplified code, everything that happens in the service method (i.e., your application code) is protected by the lock on the session (the actual code is actually more sophisticated). If you had a `Person` object in your session, you wouldn't need any synchronization in it because it is protected by the session lock.

Performance Considerations:

The scope of your locking can have an impact on the performance of your cluster. See the Tuning Terracotta Guide for more information on lock tuning.

Further reading:
Configuring Locks

Transactions

Terracotta transactions are sets of clustered object changes that must be applied atomically. Transactions are bounded by lock acquisition and release. When a clustered lock is acquired by a thread, a Terracotta transaction is started and all changes to clustered objects made within the scope of that lock are added to that transaction. When the lock is released, the Terracotta transaction is committed.

Since Terracotta transactions are batches of writes – with one Terracotta transaction potentially holding thousands of deltas to shared data – Terracotta transactions are not the equivalent of application transactions and cannot be used in application transactions-per-second computations.

When a lock is acquired, Terracotta guarantees that all of the transactions made under the scope of that lock in every JVM in the Terracotta cluster are applied locally on the heap of the acquiring thread. This ensures that threads always see a consistent view of clustered objects.

All changes to clustered objects must happen within the context of a Terracotta transaction. This means that a thread must acquire a clustered lock prior to modifying the state of any clustered objects. If a thread attempts to modify a clustered object outside the context of a terracotta transaction, a runtime exception is thrown. A special case of this is worth mentioning. If a thread synchronizes on an object that is not yet clustered, that thread does not acquire a clustered lock and does not start a Terracotta transaction. That thread may not make modifications to any shared objects, even if the object the thread has synchronized becomes shared within the scope of that synchronization.

For example, this code will result in a runtime exception:

```
// create a new Foo object which I will add to a clustered object graph.
Foo foo = new Foo();
synchronized (foo) {
  foo.setBar("Hey, there.  I'm setting the bar attribute.");
  // XXX: If I try to put foo into the clustered object "myRoot", this will throw an exception
  myRoot.put("foo", foo);
}
```

This code will resolve the issue:

```
// create a new Foo object which I will add to a clustered object graph.
Foo foo = new Foo();
synchronized (foo) {
  foo.setBar("Hey, there.  I'm setting the bar attribute.");
  synchronized (myRoot) {
    myRoot.put("foo", foo);
  }
}
```

Distributed Method Invocation (DMI)

Any method of an object contained in a shared object graph can be distributed, meaning that an invocation of that method in one virtual-machine will trigger the same method invocation on all the mirrored instances in other virtual-machines. This is a useful mechanism for implementing a distributed listener model. It is important to note that distributed methods work in the context of a shared instance that defines that method. It is not sufficient that the instance's class be instrumented, it must also be contained in a shared object graph.

USE DMI SPARINGLY, IF AT ALL. DMI is not the preferred way to send messages across JVMs. Use normal cross-thread communication mechanisms via data structures for that purpose. DMI is specifically designed for cases where that is not practical, for example, firing a redraw cycle in a clustered Swing application.

Further reading:
Configuring Distributed Method Invocation

Bytecode Instrumentation

Terracotta's clustering behavior is injected into application code at runtime by the use of bytecode instrumentation. Before the bytecode of a class is loaded by the JVM, Terracotta manipulates the bytecode of that class according to the Terracotta configuration. This includes acquiring clustered locks and pushing changes to clustered objects among other things.

Terracotta can be configured to instrument any number of the classes loaded into the JVM. If you instrument all classes, all classes will have clustering behavior. To restrict clustering behavior to classes that actually need to have that behavior injected into them, instrument only that subset of classes. Narrowing the set of classes included for instrumentation by Terracotta reduces the overhead introduced by the Terracotta class instrumentation process at class load-time. It is also useful for reducing any runtime overhead introduced by the clustering code injected into classes. While the overhead is minimal for code that doesn't manipulate shared objects or acquire shared locks, excluding classes that don't need Terracotta instrumentation eliminates that overhead entirely.

A class must match an "include" pattern in order to be instrumented, unless a field in that class has been declared a Terracotta root. With this one exception, if a class doesn't match the included set, no instrumentation will occur, regardless of any further configuration that might pertain to that class. For example, configuring locks for synchronized methods in the class does not cause the class to be instrumented.

Further reading:

- The instrumented-classes section of the Configuration Guide and Reference
- Debugging options in the Configuration Guide and Reference

Literals

The following is a list of the data types that Terracotta treats as primitive literals (root re-assignment allowed):

- int
- long
- char
- float
- double
- byte
- short
- boolean

The following is a list of the data types that Terracotta treats as literals (root re-assignment ignored):

- java.lang.Integer
- java.lang.Long
- java.lang.Character
- java.lang.Float
- java.lang.Double
- java.lang.Byte
- java.lang.Short
- java.lang.Boolean

- java.math.BigInteger
- java.math.BigDecimal

- java.lang.String
- java.lang.Class
- java.lang.StackTraceElement

- java.util.Currency

- All Enum types

This list is a superset of the standard Java literals. Terracotta treats these literals differently from other data types. Literal data types are not assigned object IDs. Terracotta does not use object references for literal data types, even for non-primitives such as Strings. However, Terracotta 2.6 and greater respects String.intern(), avoiding the duplication in memory of interned strings.

The following sections summarize the main characteristics of Terracotta literals.

Literals as Terracotta Roots

While non-primitives configured as roots can only be assigned a value once, Terracotta primitive literals (Java primitives) can be reassigned any number of times. For more information on roots, see Roots.

Mutability of Strings

At this time, the values of String objects and other non-primitive Terracotta literals is not mutable. See the status of the following JIRA issue regarding a resolution:

Unable to locate Jira server for this macro. It may be due to Application Link configuration.

Locking on Literals

If code synchronizes on a shared object that's a Terracotta literal, and that object is autolocked, the lock is on the object's value, not on its reference. This is unlike locking behavior in Java, where the lock is on the object reference. For example, if two methods synchronize on shared strings having the same value, they will hold a common lock on that value, whereas standard Java behavior would dictate two different locks on two different references to the same value.

Reference Equality

If Terracotta literals with equal values appear in more than one shared collection, *those values are duplicated*. Standard Java behavior does *not* duplicate object values in collections, but stores only the object references.

Portability

Terracotta can cluster most Java objects, but there are limits to what can be added to a Terracotta cluster. An object that can be clustered by Terracotta is called "portable". In order for an object to be portable, its class must be instrumented. That is, Terracotta must weave in special modifications to the bytecode of an object's class before the class is loaded. Whether or not a class is instrumented is determined by the Terracotta configuration file. In addition, some classes are automatically instrumented by Terracotta.

Instances of most instrumented classes are portable, but there are a few constraints on object portability. Some objects are inherently non-portable because they represent JVM-specific or host machine-specific resources. Some of the filesystem-related classes such as java.io.FileDescriptor are examples of host machine-specific resources that are inherently non-portable. Thread and Runtime are examples of JVM-specific resources that are inherently non-portable.

Other non-portable objects are instances of classes that extend inherently non-portable, uninstrumented classes, or logically instrumented classes described in the topic *Physically vs. Logically Managed Objects*.

A portable object that becomes clustered by Terracotta is said to be "managed". A portable object becomes managed if it becomes reachable from another managed object. Everything reachable by a managed object is part of a "managed graph" of objects. Terracotta keeps track of changes made to managed objects by way of special functionality injected into regular classes when they are loaded. The classes of all portable objects must be instrumented in this way; likewise, all classes that directly access fields of managed objects must also be instrumented - even if they themselves will never be shared.

Boot JAR Classes

For most classes that need Terracotta functionality injected into them, this instrumentation can be performed transparently by Terracotta when the class is loaded. Some classes, however, are loaded too early in the lifecycle of the JVM for Terracotta to hook into their loading process. These are classes that are loaded by the boot classloader. Such classes cannot be instrumented at classload time, but must be pre-instrumented and placed in a special JAR file that is then prepended to the boot classpath.

This JAR file, called the "boot JAR", is created by the Terracotta boot JAR tool. Some of the classes in the Terracotta boot JAR are placed in it automatically by the boot JAR tool. Other classes may be added to the boot JAR by augmenting the Terracotta configuration to include them in the boot JAR section and then running the boot JAR tool.

A class that is loaded by the boot classloader cannot be shared by Terracotta unless it is in the boot JAR. Likewise, a class that has a superclass that is loaded by the boot classloader cannot be shared by Terracotta unless that superclass is in the boot JAR.

Physically vs. Logically Managed Objects

Most objects are managed by moving their field data around the Terracotta cluster. These classes of objects are described as "physically managed" because Terracotta records and distributes changes to the physical structure of the object. When a field of a physically managed object changes, the new value of the field is sent to the Terracotta server and to other members of the Terracotta cluster that currently have the changed object in memory.

Some classes of objects are not shared this way, however. Instead of moving the physical structure of such objects around, we record the methods called on those objects along with the arguments to those methods and then replay those method calls on the other members of the Terracotta cluster. These classes of objects are described as "logically managed" because Terracotta records and distributes the logical operations that were performed on them rather than changes to their internal structure.

Objects are logically managed either for performance reasons or because their internal structure is JVM-specific. Classes that use hashed structures such as java.util.Hashtable, java.util.HashMap, or java.util.HashSet are logically managed. The hashcodes used to create the internal structure of these classes are JVM-specific. If an object that has a structure based on JVM-specific hashcodes were physically managed, its structure on other JVMs in the Terracotta cluster would be incorrect, since the hashcodes would be different on the other JVMs.

Non-portable and Logically Managed Classes

Classes that have a non-portable class in their type hierarchy are not portable. Sub-classes of non-instrumented classes fall into this category as do sub-classes of inherently non-portable classes like java.lang.Thread. A subclass of Thread is not portable because Thread itself is not portable, even if it is instrumented.

While logically managed classes are themselves portable, there are some restrictions on the portability of classes that have logically managed classes in their type hierarchy. This is due to technical details of the Terracotta implementation of logically managed classes.

If the subclass of a logically managed class has declared additional fields, that class is not portable if:

- it directly writes to a field declared in the logically managed superclass
- it overrides a method declared in the logically managed superclass

If you find that a class is not portable because it inherits from a class that is non-portable due to being uninstrumented, you can modify the Terracotta configuration to include all of the classes in the type hierarchy for instrumentation. However, if you find that a class is not portable because of the other inheritance restrictions, you must refactor the class you want to make portable so that it does not violate the inheritance restriction.

Portability Contexts

There are a number of contexts in which objects become shared by Terracotta. In each of these contexts, the portability of the objects that are about to be shared is checked. If there is a portability error, Terracotta throws an exception (TCNonPortableObjectError). See Configuring Terracotta for a practical discussion on how to prevent this exception.

Field Change

When a field of an object that is already shared changes, the object being newly assigned to that reference is checked for portability. If that object is not portable, Terracotta throws a portability exception.

If the newly referenced object is itself portable, then its object graph is traversed. If any object reachable by the newly referenced object is not portable, Terracotta throws a portability exception.

Logical Action

Methods called on logically-managed shared objects that change the state of that object are termed "logical actions". When such a method is called, the arguments to that method are checked for portability. If any of those objects are not portable, Terracotta throws a portability exception.

If the argument objects are themselves portable, then their object graphs are traversed. If any object reachable by the argument objects is not portable, Terracotta throws a portability exception.

Object Graph Traversal

When a top-level object becomes shared in any of the above contexts, Terracotta traverses all of the objects reachable in that top-level object's graph. During that traversal, if any non-portable object is encountered, Terracotta throws a portability exception.

Transience in Terracotta

A single reference from an entire object graph can prevent your application from sharing anything in that graph (see #Portability). This is a very similar situation to Java serialization, where a single reference to a non-serializable object can prevent an entire graph from being serialized.

Like Java serialization's use of the transient modifier, Terracotta provides a mechanism, Terracotta transience, to allow certain fields to be skipped during sharing. Terracotta also provides a richer model than Java serialization, allowing you to automatically run various methods or snippets of code when an object is loaded so that you can assign appropriate values to transient fields.

Although Terracotta transience and Java transience are similar, by default, Terracotta does not skip fields that are marked with the Java transient modifier when sharing an object. This is because Java serialization and Terracotta sharing are significantly different, and just because a field should not be serialized does not mean Terracotta should not share it.

Making Fields Transient

Making a field transient in Terracotta is similar to declaring a field transient for Java serialization. When the traverser reaches a transient field, it skips that field. The result is that the object referred to by that field reference and all objects that are only referenceable by that transient object are not checked for portability and they are not shared.

Consider the following example code:

```
class Person {
   String firstName;
   String lastName;
... }

class Customer extends Person {
   long customerID;
... }

class MyProperties extends java.util.Properties {
... }

class MyThread extends Thread {
... }

class Address {
   private Logger logger;
   private String stree1;
   private String street2;
   private State state;
... }

class Logger {
   private FileOutputStream out;
... }
```

Fields of classes can be made transient in the <includes> section of the Terracotta configuration in one of two ways:

- Set an include declaration to honor the built-in Java transient field modifier. In this case, if your source code uses the transient field modifier on the Address.logger field, Terracotta will automatically make the Address.logger field transient.
- Declare specific fields to be transient by name. This allows you to make fields transient that haven't been declared transient in the source code.

All classes that match a given <include> declaration will be given the declared field transience behavior. Similarly, just because a field is declared to be transient to Terracotta does not prevent it from being serialized when using Java serialization. The two are different concepts, and the only way in which they are connected is that you can tell Terracotta to honor the Java transient keyword for Terracotta transience, as explained in the Declaring On-Load Behavior topic.

If an attempt is made to declare a field a root *and* make it transient, then *all* of the following happens:

- the field is made a root;
- the field is **not** treated as transient;
- a warning is logged.

Setting Transient Fields

A transient field will only be changed in the local materialization of a shared object, it is never broadcast to other nodes. If the shared object is flushed from memory and then later retrieved into the same JVM, the transient field
value will be reset. Reestablishing transients in accessor methods is one approach, but requires null checks. Other methods are summarized below.

Declaring On-Load Behavior

Just like Java serialization, when a shared object containing a transient reference is materialized on another node in the cluster, by default that transient reference is null (or zero for primitives). If that object is being shared using Java serialization, there are two ways to initialize the Address.logger field before the object is used:

- Refactor the code to make the logger accessible only through a method that checks to see if the logger is null and performs lazy initialization of the logger if it is. This solution requires a fair amount of code change and also distorts the object model since you cannot refer to the logger field the way you normally would.
- Define a special method on the `Address` class named `readObject(java.io. ObjectInputStream)`. This method is guaranteed to be called when the `Address` object is deserialized, so you can initialize the `logger` field in this method before any thread has access to the newly deserialized object.

Terracotta has a similar feature that is configurable in the `<includes>` section of its configuration file. Each <include> declaration can be augmented with on-load behavior which determines what further steps Terracotta takes beyond applying shared object data when loading a shared object into a virtual machine.

There are two flavors of Terracotta on-load behavior. The first is to declare a method by name to be called when an object is loaded. This is similar to the special method readObject used by Java serialization except that Terracotta lets you choose the method that initializes the transient field.

If you don't have a suitable method already defined, or if, for some reason, you can't add such a method to the class in question, you can specify a BeanShell script in the Terracotta configuration that will be executed when the object is loaded.

Do not use an on-load method or a BeanShell script to create an object that is part of the shared object graph (or that is referenced by the original shared object). Terracotta server instances will not know
of the existence of the newly created shared object and therefore cannot update other Terracotta clients about it. If necessary, only transient objects should be created by an on-load method or BeanShell script.

BeanShell Scripting

Java BeanShell is a scripting language similar to ordinary Java code but has the advantages of being more dynamic. For example, in BeanShell you don't have to declare types for your variables, and the script can be executed at runtime. Java BeanShell has been standardized in JSR-274. A full description of the BeanShell is beyond the scope of this document, but you can view an introduction, tutorial, complete documentation, and the full specification at http://www.beanshell.org/.

Beyond the basic specification, Terracotta has made a few small modifications to the BeanShell environment to permit better use in the Terracotta run-time environment.

Referencing Shared Object in BeanShell

For technical reasons, it is not possible to refer to the object being shared as `this` in BeanShell code used in an on-load declaration in the Terracotta configuration file. Instead, you must refer to it as `self`. For example, to initialize the `Address.logger` field you might use the following on-load BeanShell script:

```
self.logger = Logger.getLogger(self.getClass());
```

Using an Applicator Class

You can write an *applicator* class to inject behavior into the Terracotta hydrate/dehydrate process. The applicator class, which is integrated using a TIM, does three main things:

1. Encodes newly shared object into the Terracotta internal data format (called DNA).
2. Consumes DNA and materialize the object.
3. Applies discreet updates.

In addition, your applicator could also do things specific to your types when objects are being faulted into memory.

The Hibernate and cglib TIMs have their own applicators. These can be viewed as examples.

Terracotta Limitations

This section describes limitations in Distributed Shared Objects technology.

Unknown Not-Instrumentable Classes

Some user-defined classes may not be shareable in ways we cannot detect. They may, for example, have native methods in user-defined classes that make them impossible to share correctly.

Classes That Should Be Logically Managed

Terracotta cannot detect all cases where a user-defined or third-party library class should be logically managed. An example of such a case is user code that explicitly examines an object's hash code. Because the hash code of an object can be a VM-specific value, it might not be safe to share such objects. Typically, this occurs in collection libraries like GNU Trove.

There is currently no customer-facing way to make new classes logically-managed. There is an ongoing effort to add logically-managed support for commonly-used third-party libraries that require it. Terracotta does have limited support for the GNU Trove collections, for example. For more specific information about exactly which third-party classes have been adapted by Terracotta to be logically managed, please contact Terracotta support at support@terracottatech.com.

Non-Static Inner Classes

If a non-static inner class is included for instrumentation, its containing class must also be included for instrumentation and vice versa. Terracotta does not currently have a way to enforce this, so it is possible to start a Terracotta client in this state.

The symptom of this mis-configuration is NoSuchMethodErrors being thrown in instrumented methods that use direct field access on the uninstrumented inner or outer class's instance fields.

Performance Considerations

Although convenient for experimentation and prototyping, a class include pattern of `*..*` can have negative performance impacts. The optimal set of included classes is only those types which be will shared in Terracotta. In particular those classes that comprise the implementation of the web container should be excluded. For example the classes matching patterns such as `org.apache.catalina..*` or `org.apache.jasper..*` should be excluded unless absolutely necessary (since these are core classes of the Tomcat web container). The default configuration file included with Sessions Configurator contains exclude patterns suitable for many popular containers.

Terracotta for Sessions Concepts

For a quick-start introduction to Terracotta Sessions, see the Sessions Tutorial.

Terracotta Cluster Architecture

A Terracotta cluster is composed of one or more Terracotta server instances and one or more Terracotta clients, all sharing the same object data and locks. The applications running in the client JVMs may be separate instances of the same logical application, or they may be separate logical applications that share some distributed object data and thread coordination – the Terracotta cluster does not make a distinction between these modes.

The following sections give a brief introduction into the functioning of cluster components. For a more detailed discussion on cluster behavior, including how to set up a Terracotta cluster and configure High Availability, see the Platform chapter in .

Terracotta Server Instances

Terracotta server instances together form a Terracotta server array, which is the heart of a Terracotta cluster. Each Terracotta server instance performs two basic functions:

1. Cluster-wide thread coordination and lock management
2. Clustered object data management and storage

A server instance brokers between all threads on all client JVMs for lock requests. It keeps track of which locks are held by which threads on the client JVMs and accepts and responds to lock requests. It also keeps track of which threads in the client JVMs are waiting on clustered objects. When notify() or notifyAll() is called on a clustered object on a thread in a client JVM, the server instance chooses which set of threads in the cluster ought to be notified and sends notifications up to the appropriate clients.

The server instance also manages object data and the persistence of that data. As clustered objects are changed on the client JVMs, the server instance receives those changes, stores them to disk as necessary, and sends them to other client JVMs in the cluster that need them. It also keeps track of the set of clustered objects currently resident in heap for each client JVM and responds to requests from client JVMs for objects that they don't currently have in their local heap.

A Terracotta server array suffers no service interruption if an active server instance goes down or is restarted if a standby server has been configured. Current clients continue as before and new clients can join the cluster as usual.

Client reconnection is controlled by a configurable time window. While the window is open, the server instance waits and previously connected clients are allowed to reconnect. When all previously connected clients have reconnected or the client reconnect time has elapsed, a new or restarted server instance completes the handshakes with its reconnected clients and the cluster assumes normal operation. All previously connected clients that fail to reconnect by the time the client reconnect window closes are perceived by the server instance as having died. When the client reconnect window closes, the server instance assumes normal operation.

See Configuring Terracotta for High Availability in for more information.

Terracotta Client

A Terracotta 'client' is a JVM that participates in a Terracotta cluster and that your application runs in. You application may run in a standalone JVM or in an application server. From the Terracotta cluster's perspective, they are both Terracotta clients.

On startup, a Terracotta client JVM initiates a network connection with a Terracotta server instance. Once the connection is made, the client is allowed to proceed with its normal startup operations. As classes are loaded into the client JVM, they are instrumented with Terracotta bytecode modifications according to the Terracotta configuration. For more information on bytecode modifications, see the instrumented-classes section of the Configuration Guide and Reference.

If a client JVM's network connection to all Terracotta server instances is disconnected, the client attempts to reconnect for a configurable number of seconds. While the client is disconnected, any thread that attempts to acquire a shared lock will block. When the client perceives server death, the client will try to connect to another active server following these steps:

1. Attempt to connect to an active Terracotta server instance.
2. If a successful connection is made, initiate a handshake procedure.
3. If the handshake procedure is completed successfully, continue normal operation.

Cluster Events

Cluster events provides a mechanism for a Terracotta client to be notified of significant events that occur in the cluster, such as client connects and disconnects. Application-specific behavior can then be defined based on cluster membership events. Examples include:

Terracotta provides a cluster-events API allowing you to implement cluster "awareness" and event responses directly into your application code.