Configuring DSO

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.

- **Introduction**
- How DSO Clustering Works
- Platform Concepts
- Hello Clustered World
- **Setup and Configuration**
- Planning for a Clustered App
- Configuring Terracotta DSO
- Configuration Reference
- Installation
- **APIs**
- Using Annotations
- Cluster Events
- Data Locality Methods
- Distributed Cache
- Clustered Async Data Processing
- **Tool Guides**
- Developer Console
- Operations Center
- tim-get (TIM Management Tool)
- Platform Statistics Recorder
- Eclipse Plugin
- Sessions Configurator
- Clustering Spring Webapp with Sessions Configurator
- Maven
- JMX
- **Testing, Tuning, and Deployment**
- Top 5 Tuning Tips
- Testing a Clustered App
- Tuning a Clustered App
- Deployment Guide
- Operations Guide
- **FAQs and Troubleshooting**
- General FAQ
- DSO Technical FAQ
- Troubleshooting Guide
- Gotchas
- Non-portable Classes
- **Reference**
- Migrating From DSO
- Concept and Architecture Guide
- Examinator Reference Application
- Clustered Data Structures Guide
- Integrating Terracotta DSO
- Clustering Spring Framework
- Integration Modules Manual
- AspectWerkz Pattern Language
- Glossary

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

Configuring Terracotta DSO

This guide contains instructions on how to set up your `tc-config.xml` file so that your application will be clustered with Terracotta Distributed Shared Objects (DSO).

- Overview of the Configuration Process
- Identifying a root
- A basic configuration file
- Configure a root
- Starting a Terracotta Server Instance
- Start your application
- Instrumentation and Locking
- Resolving TCNonPortableObjectError
- Resolving UnlockedSharedObjectException - Configuring Locking
- Confirm that state replication was successful

Overview of the Configuration Process

Getting Terracotta integrated is a simple process. Using a configuration file (or annotations), you tell Terracotta which classes need to be instrumented and what locking should be provided to replicate your shared state across many JVMs.

The basic steps of this process are:

1. (optional) Identify a root
2. Create your config file
3. Start a Terracotta Server Instance
4. Start your Application (a Terracotta Client)
5. (iteratively) resolve UnlockedSharedObject and TCNonPortableErrors

The process is approximated by the diagram to the right.

Identifying a root

All Terracotta applications have a root. A root is a core concept in Terracotta. Roots are the top of a clustered graph. They are typically a collection such as a HashMap or ArrayList. They can also be a POJO.

You can read more about roots in the Concept and Architecture Guide.

You may not have to specify a root, depending on your application. If you are using EHCache, and are using the EHCache TIM, then the root selection is automatically made for you inside the EHCache TIM (what this means is that the Integration Module for EHCache already knows how to cluster EHCache instances and does not need an additional definition of a root to function properly).

A basic configuration file

Next you will need a basic configuration file (if you do not already have one). Here is the most basic configuration file (if you already have a configuration file, use that one instead):

tc-config.xml:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd">

  <servers>
       <update-check>
         <enabled>true</enabled>
       </update-check>
  </servers>
</tc:tc-config>
```

Save this file to a file named `tc-config.xml`. You must now update your config file to contain a root.

Configure a root

To configure a root, first identify a point in your application that represents shared state. Usually this shared state is held by a HashMap, an Array, or some other java.util Collection.

For example, in the HashMap Recipe the HashMap held by the field `map` is configured to be a root.

The section in the `tc-config.xml` file that controls roots is called `roots` and should reside inside the `application/dso` xml element.

The HashMap Recipe config is repeated here for reference. Notice the `roots` section, which identifies the `map` field to be a root in the Main class:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd">

  <application>
    <dso>
      <locks>
        <autolock>
           <method-expression>* Main.get(..) </method-expression>
           <lock-level>read</lock-level>
        </autolock>
        <autolock>
           <method-expression>void Main.put(..) </method-expression>
        </autolock>
        <autolock>
           <method-expression>void Main.list(..) </method-expression>
           <lock-level>read</lock-level>
        </autolock>
      </locks>
      <roots>
        <root>
          <field-name>Main.map</field-name>
        </root>
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

Starting a Terracotta Server Instance

By default, server instances use the file `tc-config.xml`, if one exists in the local directory. For this example, be sure the configuration file you edited is saved in $TC_HOME before starting a Terracotta server instance.

There are a number of ways to specify how Terracotta configuration files are loaded. For more information, see *Working with Terracotta Configuration Files* in the

Error rendering macro 'html'

Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED

.

To start a Terracotta server instance:

```
$ $TC_HOME/bin/start-tc-server.sh
```

If the server instance started successfully, you should see output similar to the following:

```
2008-03-15 10:18:50,654 INFO - Configuration loaded from the file at
 '/Users/myDirectory/src/forge/cookbook/helloworld/tc-config.xml'.
2008-02-02 11:52:14,719 INFO - Terracotta 2.5.1, as of 20080128-160143
(Revision 6850 by cruise@rh4mo0 from 2.5)
...
2008-02-02 11:52:17,491 INFO - Terracotta Server has started up as ACTIVE node
on 0.0.0.0:9510 successfully, and is now ready for work.
```

Note the line that states what configuration file was used to start the server. This configuration file should correspond to the one you created.

Start your application

Now that you have a running server instance and a working configuration file, you can start your application. Because we have configured only a root, and not instrumentation or locking, you will encounter exceptions thrown by Terracotta. These exceptions can assist you in configuring your application correctly.

Instrumentation and Locking

During the integration process, it is common to receive the following types of exceptions from Terracotta:

- TCNonPortableObjectError exception
- UnlockedSharedObjectException exception

Don't panic. These are normal!

Next, we will describe how to resolve them.

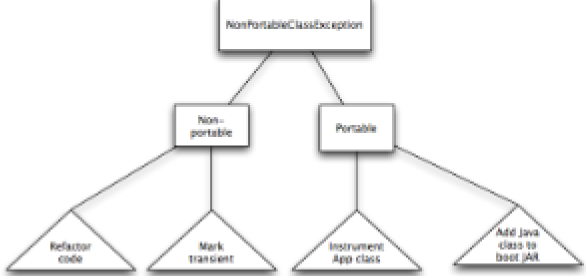
Resolving TCNonPortableObjectError

Terracotta offers Developer Support to assist you with development time issues. If you would like hands-on assistance with your project, contact us at info@terracotta.org.

This error is caused by an attempt to share an object that cannot be shared, or a non-portable object. More specifically, the error has one of the following conditions:

- An attempt was made to share a class that has not been instrumented by Terracotta.
- An attempt was made to share a *non-portable* class that has been instrumented by Terracotta.

To see a solution diagram of how this error is handled, click the following graphic.



Best Practice

Since classes are instrumented by Terracotta *at class load time*, the attempt at sharing the offending class could occur at some point after your application has been up and running. Rigorous testing and exercising of your application *after* it is clustered with Terracotta is the most effective way to avoid later occurrences of TCNonPortableObjectError (see Testing DSO).

To resolve the second case, a non-portable instrumentation error, either mark the section of the graph as transient, or refactor your application. The rest of this section deals with the first case, where a portable class that was not instrumented is being shared.

When you receive TCNonPortableObjectError, Terracotta gives you suggestions of how to resolve it. For example, let's look at the Instrumentation Recipe, which includes a Terracotta configuration file that instruments a class called Counter. For this example, we ignore the locking section and remove the section that instruments Counter, leaving the following configuration:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd">

  <application>
    <dso>
      <roots>
        <root>
          <field-name>Main.instance</field-name>
        </root>
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

Now when we run this application:

```
$ dso-java.sh Main
```

we are rewarded with a NonPortableObjectError that looks like this:

```
Exception in thread "main" com.tc.exception.TCNonPortableObjectError:
*******************************************************************************
Attempt to share an instance of a non-portable class referenced by a portable class. This
unshareable class has not been included for sharing in the configuration.

For more information on this issue, please visit our Troubleshooting Guide at:
http://terracotta.org/kit/troubleshooting

Referring class       : Main
Referring field       : Main.counter
Thread                : main
JVM ID                : VM(1)
Non-portable root name: Main.instance
Non-included class    : Counter

Action to take:

1) Reconfigure to include the unshareable classes
   * edit your tc-config.xml file
   * locate the <dso> element
   * add this snippet inside the <dso> element

       <instrumented-classes>
         <include>
           <class-expression>Counter</class-expression>
         </include>
       </instrumented-classes>

   * if there is already an <instrumented-classes> element present, simply add
     the new includes inside it

It is possible that some or all of the classes above are truly non-portable, the solution
is then to mark the referring field as transient.
```

Notice at the end of the exception, it tells us how to fix our config file. In general, there are three ways to deal with TCNonPortableObjectError:

- **Add Instrumentation:** update your tc-config.xml with the appropriate configuration
- **Mark Transient:** mark the offending object (or object graph) as transient in your tc-config.xml
- **Refactor:** refactor your code to avoid putting the offending instance into the clustered graph

If the error message instructs you to add a class to the Terracotta boot JAR, see the boot jar section in the Troubleshooting Guide.

Add Instrumentation

We need to add the `Counter` class to the instrumentation list. This is done by adding the snippet of configuration provided in the Exception to the `application/dso` XML element. Our resulting `tc-config.xml` file now looks like this:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd">

  <application>
    <dso>
      <instrumented-classes>
        <include>
          <class-expression>Counter</class-expression>
        </include>
      </instrumented-classes>
      <roots>
        <root>
          <field-name>Main.instance</field-name>
        </root>
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

In your application, you may have to repeat this process several times until all of the classes that are being shared are accounted for.

Note that it is also possible to use wildcards to identify classes.

The Wildcard Recipe shows an example of how to use pattern matching. Wildcards can be used, for example, to identify an entire package of classes. Our sharededitor sample, which is shipped with Terracotta, makes use of this short-cut.

Instead of listing each class, you can use a wildcard to match all of the classes. For example, an entire package can be identified using a single pattern such as `demo.sharededitor.models.*`. A specific pattern language, called AspectWerkz, is supported for matching on the fully qualified names of instrumented classes, locks, roots, and more.

However, there is a performance cost to over-instrumenting, which can result from a match that is too broad. At the testing and tuning phases of your project, you should revisit the instrumentation portion of the Terracotta configuration.

Once you have updated your configuration file, restart your application again and continue to resolve any further Exceptions.

Mark Transient

Certain classes causing the TCNonPortableObjectError exception are truly non-portable and cannot be instrumented. Terracotta can be configured to avoid trying to share these classes using transience. Terracotta can recognize transient classes if they are declared transient in code or in `tc-config.xml`.

Common Declared Transients

Here are some common classes whose instances, if referenced by shared objects, will most likely need to be declared transient: Loggers, File Handlers, Sockets, java.io.**, java.net.**, java.nio.**, java.sql.Connection, javax.net.**, javax.print.*, java.lang.reflect, java.lang.Thread, java.lang.Runtime.

Be sure to mark the offending class highest in the entire object graph that's being added. Marking the immediate offending class may not prevent the TCNonPortableObjectError exception.

If the class that implements the field has not been instrumented and is required in the boot JAR, add it to the <additional-boot-jar-classes> section of the configuration file and regenerate the boot JAR (see this troubleshooting issue).

In some cases, Terracotta must add explicit support for a core Java class — simply adding to the boot JAR does not work. Before you contact Terracotta, see Unsupported Classes for a list of unsupported classes.

For more information on how to configure transient classes, see the instrumented classes and the transient fields sections of the Configuration Guide and Reference.

For more information on transience, see Transience in Terracotta in the DSO Concept and Architecture Guide.

Once you have updated your configuration file, restart your application again and continue to resolve any further Exceptions.

Refactor

If instrumenting or marking as transient are insufficient for eliminating the problem you are running into, consider refactoring your code. Code that was not written to cleanly separate shared state from non-shared state may be too problematic to fix using the instrumentation and transient strategies.

More Information on Portability

- The non-portable-dump section of the Configuration Guide and Reference shows how to analyze the log when a TCNonPortableError occurs.
- The DSO Concept and Architecture Guide explains portability in Terracotta.
- The DSO Troubleshooting Guide has example code and a discussion of how TCNonPortableError could be generated and resolved.

Resolving UnlockedSharedObjectException - Configuring Locking

An UnlockedSharedException simply means that Terracotta has detected that our application has tried to update a value in a shared object without a Terracotta lock present.

Terracotta locks are very much the same as standard Java synchronization. The big difference between Terracotta locking and Java synchronization is that Terracotta **requires** that you have a lock to update a shared object, while Java will let this happen without complaining.

This is actually a *good thing*, because with Terracotta, your application is now running in a clustered environment, and a clustered environment is equivalent to having multiple threads. When multiple threads have write access to shared data, just as in normal Java, you must always protect access to that data. Terracotta saves you from finding out race conditions or data corruption at run-time by detecting and reporting attempts to update shared data outside of a lock.

For more details on the similarities, and differences, between Terracotta locking and Java locking, read the DSO Concept and Architecture Guide Locks Section.

Let's use the Instrumentation Recipe again. So far, we have added the proper instrumentation configuration to our config file. However, when we run this example again, it tries to update a counter in a shared object, so we now get an UnlockedSharedException, which looks like this:

```
com.tc.object.tx.UnlockedSharedObjectException:
********************************************************************************
Attempt to access a shared object outside the scope of a shared lock.
All access to shared objects must be within the scope of one or more shared locks
defined in your Terracotta configuration.

Please alter the locks section of your Terracotta configuration so that this access
is auto-locked or protected by a named lock.

For more information on this issue, please visit our Troubleshooting Guide at:
 http://terracotta.org/kit/troubleshooting


    Caused by Thread: main  in  VM(2)
    Shared Object Type: Counter
********************************************************************************

       <elided>
        at com.tc.object.TCObjectImpl.objectFieldChanged(TCObjectImpl.java:320)
        at com.tc.object.TCObjectImpl.intFieldChanged(TCObjectImpl.java:360)
        at Counter.__tc_setcount(Counter.java)
        at Counter.count(Counter.java:13)
        at Main.count(Main.java:14)
        at Main.main(Main.java:20)
```

This tells us several things. First of all, we know that our application tried to update a field on a shared object without a Terracotta lock present.

This can be due to one or more of the following:

1. We did not configure Terracotta locking for this code.
2. The code itself does not have synchronization that Terracotta can use as a boundary.
3. The class doing the locking must be included for instrumentation.
4. The object was first locked, then shared (for an example, see this gotcha).
5. This error can also be caused by not specifying the fully qualified method name in a lock expression.

Here's how we can resolve these issues. First, identify the method upon which the lock should be applied.

Identify the method for configuration

We can identify what method was in question by analyzing the stack trace. Let's look at the stack-trace:

```
        <elided>
         at com.tc.object.TCObjectImpl.objectFieldChanged(TCObjectImpl.java:320)
         at com.tc.object.TCObjectImpl.intFieldChanged(TCObjectImpl.java:360)
         at Counter.__tc_setcount(Counter.java)
         at Counter.count(Counter.java:13)
         at Main.count(Main.java:14)
         at Main.main(Main.java:20)
```

In this stack trace, the method that is causing the problem is `Counter.count()`. We can figure this out by looking for the line directly below the first line of Terracotta injected code. Terracotta inject code always begins with `_tc` so the first line in this stack trace of Terracotta injected code is at Counter. __tc_setcount(Counter.java)}}.

The line directly below this code is `Counter.count(Counter.java:13)` meaning the `Counter.count()` method is to blame for this particular exception. Furthermore, we can actually make a pretty good guess as to what went wrong - first of all, we can see that the exact spot in the code that tripped us up was line 13. If you have the source code, pull up the Counter class and you can see what the code is that triggered this exception.

Here it is, from the Counter.java class in the Instrumentation recipe:

```
line 13:       count++;
```

Even if we don't have access to the source, we can still make a pretty good guess what happened. This is because the Terracotta injected code gives us a clue. The injected code for a field setter will always take the form of `"__tc_setfieldname"` where `fieldname` is the name of the field, so we know from this particular code that the `Counter` class was trying to update the `count` field when the exception was thrown.

So, now we know what method is causing our problem. Depending on the method, there may or may not be synchronization appropriate for auto-locking. If there is, proceed to the next section. If there is not, proceed to the #Adding Terracotta locking for code without synchronization section.

Adding Terracotta auto-locking for code with synchronization

If your code already has synchronization, then the only thing you have to do is add the proper Terracotta configuration to auto-lock the code.

Auto-locking is a simple process - it tells Terracotta that for any instance of synchronization it finds within the method you configure, add a Terracotta lock boundary to that code. In effect, this converts a single JVM synchronized boundary into a cluster-wide lock boundary.

In our example above, we can see that in fact the Counter class already has synchronization on it, so we just need to add the correct Terracotta configuration to make this example work.

We add the auto-lock to the `application/dso/locks` section (if there is not a locks section, you will need to add it). We must specify the method-expression field, and a lock-level. The lock-level defaults to write, so you can leave that off if you like. When we are done, our config file will look like this:

```
tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd">

  <application>
    <dso>
      <instrumented-classes>
        <include>
          <class-expression>Counter</class-expression>
        </include>
      </instrumented-classes>
      <locks>
        <autolock>
           <method-expression>void Counter.count()</method-expression>
           <lock-level>write</lock-level>
        </autolock>
      </locks>
      <roots>
        <root>
          <field-name>Main.instance</field-name>
        </root>
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

Adding Terracotta locking for code without synchronization

If no synchronization exists in the method, you can add a Terracotta lock in one of two ways:

1. Use auto-synchronized
2. Use named locks

Adding locking using auto-synchronized

Auto-synchronized has the same effect as adding the synchronized keyword to the method before adding Terracotta locking. You can imagine as your class is loaded, Terracotta instruments the class and adds synchronization to the method you have specified as auto-synchronized. Then it adds the auto-lock.

To auto-synchronize a method, use the `auto-synchronized="true"` attribute on the auto-lock. Adding auto-synchronized, instead of just an auto-lock, to the Counter example, the `tc-config.xml` file would look like this:

```
tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd">

  <application>
    <dso>
      <instrumented-classes>
        <include>
          <class-expression>Counter</class-expression>
        </include>
      </instrumented-classes>
      <locks>
        <autolock auto-synchronized="true">
           <method-expression>void Counter.count()</method-expression>
           <lock-level>write</lock-level>
        </autolock>
      </locks>
      <roots>
        <root>
          <field-name>Main.instance</field-name>
        </root>
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

Adding locking using named locks

If none of the above methods are sufficient for your needs, you can resort to adding a named lock. **You should avoid named locks unless absolutely necessary** as named locks are global, meaning the lock is very coarse-grained, and may impact the performance of your application.

Adding named locks is very similar to adding auto-synchronized, however in the auto-synchronized case, the lock is acquired on the shared object, whereas a named lock is acquired globally for all methods locked with that name.

To add a named lock, use the named lock syntax, like so:

```
tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd">

  <application>
    <dso>
      <instrumented-classes>
        <include>
          <class-expression>Counter</class-expression>
        </include>
      </instrumented-classes>
      <locks>
        <named-lock>
            <lock-name>lockOne</lock-name>
           <method-expression>void Counter.count()</method-expression>
           <lock-level>write</lock-level>
        </named-lock>
      </locks>
      <roots>
        <root>
          <field-name>Main.instance</field-name>
        </root>
      </roots>
    </dso>
  </application>
</tc:tc-config>
```

Confirm that state replication was successful

Once you are finished updating your configuration file, and your application appears to run correctly with Terracotta installed, you can now test to see if the state is truly replicated.

Start a Console Session

Launch the Terracotta Developer Console and inspect the data in the object browser. This is a good way to use a single JVM process to verify that the data you expect to be replicated is in fact shared with Terracotta.

Start a second JVM

Once your application is working with a single JVM, you must check it with 2 (or more) JVMs. Start your application again. Of course, depending on your application, you may or may not be able to do this without changes to your application code.

For example, you may need to assign different responsibilities to each JVM. It may be that the first JVM may need to be a Producer, and the second a Consumer. Or it may be that all JVMs are equivalent (e.g. a Web Application processing User Requests). Depending on your application, you may need to factor each Main process into one or more differing startup modes.