Cluster Events

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 »**

Platform Cluster Events

Introduction

A cluster-events API that allows events to be communicated using a standard Java programming approach is available as part of the Terracotta API, introduced with Terracotta 3.0.0 and now part of the Terracotta Toolkit. This Java-based events API can be more easily integrated into a Java applications environment.

Previous versions of Terracotta relied on JMX-based cluster events. Beginning with Terracotta 3.0.0, these JMX-based cluster events are no longer supported.

Applications that rely on the obsoleted JMX API for cluster events must be rewritten to use the new Java API. Support for the JMX cluster events will be removed without further notice. Application code using the TerracottaCluster MBean or registering notifications with it will receive an UnsupportedOperationException.

The remainder of this document discusses the use of the cluster-events API.

The latest Javadoc for the cluster-events API can be seen here (see the `org.terracotta.cluster` package).

Events Types

All Terracotta cluster events contain an instance of `DsoNode` corresponding to the current node. To identify the node, this instance can access the node's unique ID, hostname, and IP address.

The following table defines Terracotta cluster events and their characteristics.

Event	Receiver	Purpose	Definition	Frequency
nodeJoined	All nodes in cluster	Topology change	Indicates to the cluster and the current node that it is now part of the cluster.	Once per the current node's lifetime.
nodeLeft	All nodes in cluster	Topology change	Indicates to the cluster and the current node that it has been evicted from the cluster.	Once per the current node's lifetime. This event **may never be received** (see #Handling nodeLeft Not Received).
operationsEnabled	Current node	Operational change	Indicates to the current node that its operations are being propagated through the cluster.	Unlimited after a nodeJoined or an operationsDisabled and before a nodeLeft event affecting the current node.
operationsDisabled	Current node	Operational change	Indicates to the current node that its operations are not being propagated through the cluster.	Unlimited after an operationsEnabled and before a nodeLeft event affecting the current node.

The event flow is shown in the diagram to the right. Note that operations enabled/disabled events can recur in a loop.

Example Eventing Sequence – Current Node

The following example illustrates the types of events that can be received by an application about the current node (Node1).

1. Node1 disconnected from the cluster – *node operations disabled* generated.
2. Node1 reconnects to the cluster before the timeout window closes – *node operations enabled* generated.
3. Node1 disconnected from the cluster – *node operations disabled* generated.
4. Node1 reconnects (or attempts to reconnect) to the cluster after the timeout window closes – *node left* generated.

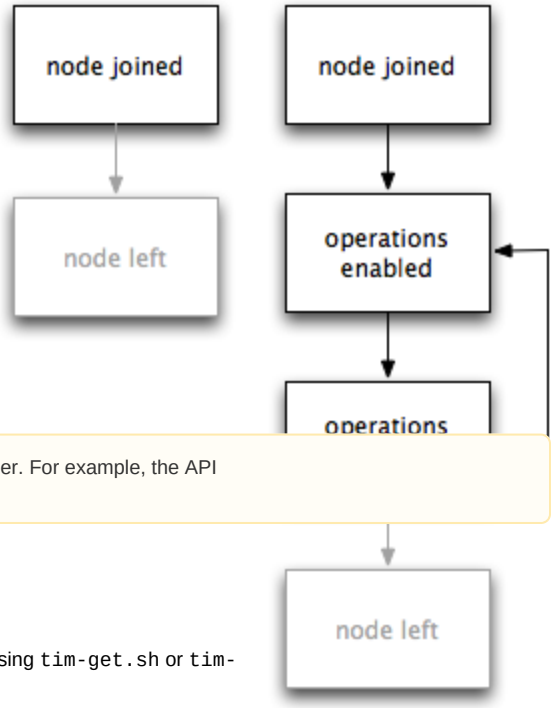
nodeLeft Not Received

The nodeLeft event **may never be received** by the current node (see #Handling nodeLeft Not Received). However, if the client reconnect window is not infinite, other nodes in the cluster should receive a nodeLeft event informing them that Node1 has left. But if the current node *and* the ACTIVE Terracotta server fail, and a backup (also known as "passive") Terracotta server takes over, the nodeLeft event is *not* received by the nodes that connect to the new ACTIVE server.

Example Eventing Sequence – Remote Node

The following example illustrates the types of events that can be received by an application about a remote node.

1. Node1 disconnected from the cluster – *node left* received by Node2.
2. Node1 reconnects to the cluster before the timeout window closes – *node joined* received by Node2.
3. Node1 disconnected from the cluster – *node left* received by Node2.
If the client reconnect window is infinite, other nodes in the cluster may not receive a nodeLeft event informing them that Node1 has left. Also, if Node1 *and* the ACTIVE Terracotta server disconnect from the cluster, and a backup (also known as "passive") Terracotta server takes over, the nodeLeft event is *not* received by the nodes that connect to the new ACTIVE server.
4. Node1 restarted and reconnects to the cluster (under a new node ID) – *node joined* received by Node2.



Obtaining the Terracotta API

Cluster events are supported by the Terracotta API, which is part of the Terracotta Toolkit.

> ⚠ The Terracotta API version number is independent of the Terracotta kit version number. For example, the API version is 1.1.0 in Terracotta 3.1.0.

The following methods are available for obtaining the Terracotta API.

Terracotta Kit

The Terracotta Toolkit JAR is installed as a dependency for many common TIMs you install using `tim-get.sh` or `tim-get.bat`. To install the Terracotta Toolkit directly, use:

UNIX/Linux

```
[PROMPT] {dist}/bin/tim-get.sh install terracotta-toolkit-1.2
```

Microsoft Windows

```
[PROMPT] {dist}\bin\tim-get.bat install terracotta-toolkit-1.2
```

If you are not certain about the API version in the Terracotta Toolkit JAR's name, use the following command to view the JARs available for your kit:

UNIX/Linux

```
[PROMPT] {dist}/bin/tim-get.sh list terracotta-toolkit
```

Microsoft Windows

```
[PROMPT] {dist}\bin\tim-get.bat list terracotta-toolkit
```

Do not confuse the terracotta-toolkit JAR intended for DSO installations with the terracotta-toolkit "runtime" JAR intended for the standard Terracotta cluster.

Maven

For Maven users, configure the following dependency:

```
<dependency>
  <groupId>org.terracotta.toolkit</groupId>
  <artifactId>terracotta-toolkit-1.1</artifactId>
  <version>1.0.0</version>
</dependency>
```

See the Terracotta kit version you plan to use for the correct API and JAR versions to specify in the dependency block.

The repository is given by the following:

```
<repository>
  <id>terracotta-repository</id>
  <url>http://www.terracotta.org/download/reflector/releases</url>
  <releases>
     <enabled>true</enabled>
  </releases>
</repository>
```

If you use `${tc-api.version}` for the <version> specification, you must extend from tim-parent or tim-system-tests parent POMs. For a POJO application, simply specify the Terracotta API version as shown in the example.

The `${tc-api.version}` works only if you extend from tim-parent or tim-system-tests parent POMs. For a POJO application, specify the Terracotta API version.

Configuring Cluster Events

Allowing a class to listen to Terracotta cluster events (making it "cluster aware") means injecting the DsoCluster interface into the class. Only instrumented classes — classes configured for sharing in the Terracotta configuration file — can be injected with DsoCluster. In addition, static classes cannot be injected.

DsoCluster must be injected into instrumented classes. It cannot be instantiated directly by application code.

There are two ways to inject DsoCluster into a class: annotations and configuration.

Annotations

To inject a class for cluster awareness, use `@InjectedDsoInstance`:

```
import com.tc.cluster.DsoCluster;
import com.tc.cluster.DsoClusterListener;
import com.tc.injection.annotations.InjectedDsoInstance;

  public class ClusterAwareClass implements DsoClusterListener {
          @InjectedDsoInstance
     private DsoCluster cluster;
 ...
}
```

In the code above, ClusterAwareClass is instrumented for injection. When an instance of ClusterAwareClass is constructed or faulted onto a shared object graph, it will be injected with cluster awareness.

Assigning a value to a DsoCluster reference fails (the value is dropped or ignored) in an application running in a Terracotta cluster. The value is assigned successfully if the application is running without Terracotta.

Configuration

You can configure injected classes using the Terracotta configuration file. Using the <injected-instances> section, you can add Terracotta functionality directly in your application. The functionality is injected into a specified field, while the field's type determines which instance is actually injected.

The <injected-instances> section has the following elements:

- <injected-instances> – Encapsulates any number of <injected-field> sections.
- <injected-field> – Encapsulates one <field-name> element.
- <field-name> – Specifies the fully qualified name of the class being injected.
- <instance-type> – Specifies the injected instance type. Useful in cases where the target field's type is insufficient to determine the correct instance to inject.

Classes that are *not* configured for instrumentation by Terracotta cannot be injected using this configuration.

Example: Injecting Cluster Awareness

You can use the <injected-instances> section to inject instrumented classes for cluster awareness, giving them the ability to listen for cluster events. For example, to inject the classes `ClusterAwareClass` and `OtherClusterAwareClass`, add the following <injected-instance> subsection to the Terracotta configuration file's clients/dso section:

```
<clients>
 <dso>
  ...
   <injected-instances>
     <injected-field>
       <field-name>com.mypackage.myClasses.ClusterAwareClass.theField</field-name>
     </injected-field>
     <injected-field>
       <field-name>com.mypackage.myClasses.OtherClusterAwareClass.theOtherField</field-name>
     </injected-field>
   </injected-instances>
   ...
 </dso>
</clients>
```

In the code above, ClusterAwareClass and OtherClusterAwareClass are instrumented for injection. When an instance of either class is constructed or faulted onto a shared object graph, it will be injected with cluster awareness.

Both `ClusterAwareClass` and `OtherClusterAwareClass` must also be configured for Terracotta instrumentation (see the Terracotta Configuration Guide and Reference).

Obtaining Cluster Topology

You can return a snapshot of active client nodes (also known as DSO nodes, or Terracotta clients) currently in the cluster using the `getNodes()` method. This method returns a Collection containing DsoNode instances, each one corresponding to an active node. Each DsoNode instance can identify its associated node by node ID, IP address, and hostname.

The following example code shows how `getNodes()` is used by an instance of a cluster-aware class:

```
import com.tcclient.cluster.DsoNode;
import com.tc.cluster.DsoCluster;
import com.tc.cluster.Dso
import com.tc.injection.annotations.InjectedDsoInstance;

public class ClusterAwareClass {

   // Regardless of how many injections are added,
   // only one DsoCluster instance is instantiated per node.

   @InjectedDsoInstance
   private DsoCluster cluster;

   public ClusterAwareClass() {

      Collection<DsoNode> nodes = cluster.getClusterTopology().getNodes();

       // now do something with nodes...
   }
```

Note the following about using the `getNodes()` method:

- No information on the Terracotta servers in the cluster is returned.
- The returned snapshot is valid only for the time the method is executed; the snapshot gives no information on the previous or future topology.

Handling nodeLeft Not Received

You can design an application to take certain actions whenever a nodeLeft event is received. However, under certain circumstances no nodeLeft event is received despite a failure that should generate one. For example, an application "sudden death," such as from a `kill -9` command, can prevent a nodeLeft event from being received.

If, after receiving an operationsDisabled event, an application node's connection to the cluster fails before your application can receive a nodeLeft event, the application may not take appropriate action. However, your application can handle this situation by starting a timer leading to a self-generated nodeLeft event that can trigger the required action.

The following pseudocode shows how such a timer can be implemented:

```
...
public void operationsEnabled(DsoClusterEvent event) {
        cancelTimer();
        operationsEnabledAction();
}

public void operationsDisabled(final DsoClusterEvent event) {
        // force a nodeLeftAction if no operations_enabled received after 10s
        startTimer(10 /* seconds */, nodeLeftAction());
        operationsDisabledAction();
}


public void nodeLeft(final DsoClusterEvent event) {
        nodeLeftAction();
}
```

An advantage to adding this type of expiration timer is that it is easy to remove should Terracotta API support a built-in solution of the nodeLeft-not-received situation.

Sample Code

The following sample class illustrates the injection of the DsoCluster interface into a class to make that class cluster aware. Note that implementing the DsoClusterListener interface allows the class to utilize listening methods able to receive Terracotta cluster events.

```
import com.tc.cluster.DsoCluster;
import com.tc.cluster.DsoClusterEvent;
import com.tc.cluster.DsoClusterListener;
import com.tc.injection.annotations.InjectedDsoInstance;

 // Any class can implement the DsoClusterListener interface
// and be used as a listener.

 public class ClusterAwareClass implements DsoClusterListener {
   @InjectedDsoInstance
   private DsoCluster cluster;  // Regardless of how many injections are added,
                                // only one DsoCluster instance is instantiated per node.


 public ClusterAwareClass() {

   cluster.addClusterListener(this);
 }

 // The following method finds the nodes that have the shared objects specified in the argument.

 public void localityAwareFoo(final Collection<Object> objects) {
   cluster.getNodesWithObjects(objects);
 }

 // Following are the methods that receive and process events.

 public void nodeJoined(final DsoClusterEvent event) {

  /* Do stuff such as obtain the node for further processing:
  *     System.out.println("The node " + event.getNode() + " joined the cluster");
  */

 }

 public void nodeLeft(final DsoClusterEvent event) {

  /* Do stuff such as obtain the node for further processing:
  *     System.out.println("The node " + event.getNode() + " left the cluster");
  */

 }

 public void operationsDisabled(final DsoClusterEvent event) {

  /* Do stuff such as obtain the node for further processing:
  *     System.out.println("The node " + event.getNode() + "has ceased operations.");
  */

 }

 public void operationsEnabled(final DsoClusterEvent event) {

  /* Do stuff such as obtain the node for further processing:
  *     System.out.println("The node " + event.getNode() + "has started operations.");
  */

 }
 }
```

Testing Cluster Awareness Without a Cluster

If a class is instrumented with DsoCluster and run without Terracotta clients and servers, it cannot be tested for cluster awareness. However, a simulation class called SimulatedDsoCluster is available for testing under this "clusterless" condition. This class is part of the simulation API you can install with tim-get:

UNIX/Linux

```
   [PROMPT] {dist}/bin/tim-get.sh install simulated-api
```

Microsoft Windows

```
   [PROMPT] {dist}\bin\tim-get.bat install simulated-api
```

The simulated API does require a specific version of the Terracotta kit.

SimulatedDsoCluster can be run on your application server to simulate a Terracotta client running in a cluster with operations enabled and completely local data. To create an example of using SimulatedDsoCluster:

```
 public class ClusterAwareClass implements DsoClusterListener {
   @InjectedDsoInstance
   private DsoCluster cluster;  // Only one DsoCluster instance per application server!
   private SimulatedDsoCluster simcluster = SimulatedDsoCluster();

// since @InjectedDsoInstance issued, a field of type DsoCluster
//will be injected when ClusterAwareClass is instrumented
```