

[Terracotta Documentation Home](#)**Terracotta 3.5.0 Documentation****Terracotta 3.5.**[Support](#)[Service](#)[Training](#)[Contact](#)

Table of Contents

[1 Getting Started](#)

- [1.0.1 Using Product Documentation](#)
- [1.0.2 Using a License File](#)
 - [1.0.2a Explicitly Specifying the Location of the License File](#)
 - [1.0.2b Verifying Products and Features](#)
- [1.0.3 Using Apache Maven](#)
 - [1.0.3a Creating Enterprise Edition Clients](#)
 - [1.0.3b Using the tc-maven Plugin](#)
 - [1.0.3c Working With Terracotta SNAPSHOT Projects](#)
 - [1.0.3d Terracotta Repositories](#)

[2 Enterprise Ehcache With Terracotta Clustering](#)

[2.1 Enterprise Ehcache Installation](#)

- [2.1.1 Step 1: Requirements](#)
- [2.1.2 Step 2: Install the Distributed Cache](#)
- [2.1.3 Step 3: Configure the Distributed Cache](#)
 - [2.1.3a Add Terracotta to Specific Caches](#)
 - [2.1.3b Edit Incompatible Configuration](#)
- [2.1.4 Step 4: Start the Cluster](#)
- [2.1.5 Step 5: Edit the Terracotta Configuration](#)
 - [2.1.5a Procedure:](#)
- [2.1.6 Step 6: Learn More](#)

[2.2 Enterprise Ehcache API Guide](#)

- [2.2.1 Enterprise Ehcache Search API For Clustered Caches](#)
 - [2.2.1a Sample Code](#)
 - [2.2.1b Stored Search Indexes](#)
 - [2.2.1c Troubleshooting Ehcache Search](#)
- [2.2.2 Enterprise Ehcache Cluster Events](#)
 - [2.2.2a Cluster Topology](#)
 - [2.2.2b Cluster Events](#)
 - [2.2.2c Events API Example Code](#)
- [2.2.3 Bulk-Load API](#)
 - [2.2.3a Bulk-Load API Example Code](#)
- [2.2.4 Unlocked Reads for Consistent Caches \(UnlockedReadsView\)](#)
 - [2.2.4a UnlockedReadsView and Data Freshness](#)
- [2.2.5 Explicit Locking](#)
- [2.2.6 Configuration Using the Fluent Interface](#)
- [2.2.7 Write-Behind Queue in Enterprise Ehcache](#)

[2.3 Enterprise Ehcache Configuration Reference](#)

- [2.3.1 Ehcache Configuration File](#)
 - [2.3.1a Setting Cache Eviction](#)
 - [2.3.1b Cache-Configuration File Properties](#)
 - [2.3.1c Exporting Configuration from the Developer Console](#)
 - [2.3.1d Terracotta Clustering Configuration Elements](#)
 - [2.3.1e Controlling Cache Size](#)
 - [2.3.1f Cache Events Configuration](#)
 - [2.3.1g Incompatible Configuration](#)
- [2.3.2 Offloading Large Caches](#)
 - [2.3.2a Tuning Concurrency](#)
- [2.3.3 Non-Blocking Disconnected \(Nonstop\) Cache](#)
 - [2.3.3a Configuring Nonstop](#)
 - [2.3.3b Nonstop Timeouts and Behaviors](#)
- [2.3.4 How Configuration Affects Element Eviction](#)
 - [2.3.4a DCV2, Strict Consistency, UnlockedReadsView, and Bulk Loading](#)
- [2.3.5 Understanding Performance and Cache Consistency](#)
- [2.3.6 Cache Events in a Terracotta Cluster](#)

- [2.3.6a Handling Cache Update Events With DCV2](#)
- [2.3.7 Configuring Caches for High Availability](#)
 - [2.3.7a Using Rejoin to Automatically Reconnect Terracotta Clients](#)
- [2.3.8 Working With Transactional Caches](#)
 - [2.3.8a Strict XA \(Full JTA Support\)](#)
 - [2.3.8b XA \(Basic JTA Compliance\)](#)
 - [2.3.8c Local Transactions](#)
 - [2.3.8d Avoiding XA Commit Failures With Atomic Methods](#)
 - [2.3.8e Implementing an Element Comparator](#)
- [2.3.9 Working With OSGi](#)

[3 Enterprise Ehcache for Hibernate](#)

[3.1 Enterprise Ehcache for Hibernate Express Installation](#)

- [3.1.1 Step 1: Requirements](#)
- [3.1.2 Step 2: Install and Update the JAR files](#)
- [3.1.3 Step 3: Prepare Your Application for Caching](#)
 - [3.1.3a Using @Cache](#)
 - [3.1.3b Using the <cache> Element](#)
 - [3.1.3c Using the <class-cache> Element](#)
- [3.1.4 Step 4: Edit Configuration Files](#)
 - [3.1.4a Hibernate Configuration File](#)
 - [3.1.4b Enterprise Ehcache Configuration File](#)
- [3.1.5 Step 5: Start Your Application with the Cache](#)
- [3.1.6 Step 6: Edit the Terracotta Configuration](#)
 - [3.1.6a Procedure:](#)
- [3.1.7 Step 7: Learn More](#)

[3.2 Testing and Tuning Enterprise Ehcache for Hibernate](#)

- [3.2.1 Testing the Cache](#)
- [3.2.2 Optimizing the Cache Size](#)
 - [3.2.2a Eviction Parameters](#)
 - [3.2.2b Reducing the Cache Miss Rate](#)
 - [3.2.2c Examiner Example](#)
- [3.2.3 Optimizing for Read-Only Data](#)
- [3.2.4 Reducing Unnecessary Database Connections](#)
 - [3.2.4a Lazy Fetching with Spring-Managed Transactions](#)
 - [3.2.4b Lazy Fetching for Non Spring Applications](#)
- [3.2.5 Reducing Memory Usage with Batch Processing](#)
- [3.2.6 Other Important Tuning Factors](#)
 - [3.2.6a Query Cache](#)
 - [3.2.6b Connection Pools](#)
 - [3.2.6c Local Key Cache](#)
 - [3.2.6d Hibernate CacheMode](#)
 - [3.2.6e Cache Concurrency Strategy](#)
 - [3.2.6f Terracotta Server Optimization](#)
 - [3.2.6g JDK Version](#)
 - [3.2.6h Statistics Gathering](#)
 - [3.2.6i Logging](#)
 - [3.2.6j Java Garbage Collection](#)
 - [3.2.6k Database Tuning](#)
 - [3.2.6l Unwanted Synchronization with Hibernate Direct Field Access](#)
 - [3.2.6m Hibernate Exception Thrown With Cascade Option](#)
 - [3.2.6n Cacheable Entities and Collections Not Cached](#)

[3.3 Enterprise Ehcache for Hibernate Reference](#)

- [3.3.1 Cache Configuration File](#)
 - [3.3.1a Setting Cache Eviction](#)
 - [3.3.1b Cache-Configuration File Properties](#)
 - [3.3.1c Exporting Configuration from the Developer Console](#)
- [3.3.2 Migrating From an Existing Second-Level Cache](#)
- [3.3.3 Cache Concurrency Strategies](#)
 - [3.3.3a READ_ONLY](#)
 - [3.3.3b READ_WRITE](#)
 - [3.3.3c NONSTRICT_READ_WRITE](#)
 - [3.3.3d TRANSACTIONAL](#)
 - [3.3.3e How Entitymanagers Choose the Data Source](#)
- [3.3.4 Setting Up Transactional Caches](#)
- [3.3.5 Configuring Multiple Hibernate Applications](#)
- [3.3.6 Finding Cacheable Entities and Collections](#)
- [3.3.7 Cache Regions in the Object Browser](#)
- [3.3.8 Hibernate Statistics Sampling Rate](#)
- [3.3.9 Is a Cache Appropriate for Your Use Case?](#)

- [3.3.9a Frequent Updates of Database](#)
- [3.3.9b Very Large Data Sets](#)
- [3.3.9c Frequent Updates of In-Memory Data](#)
- [3.3.9d Low Frequency of Cached Data Queries](#)
- [3.3.9e Requirements of Critical Data](#)
- [3.3.9f Database Modified by Other Applications](#)

[4 Quartz Scheduler](#)

[4.1 Clustering Quartz Scheduler](#)

- [4.1.1 Step 1: Requirements](#)
- [4.1.2 Step 2: Install Quartz Scheduler](#)
- [4.1.3 Step 3: Configure Quartz Scheduler](#)
 - [4.1.3a Add Terracotta Configuration](#)
 - [4.1.3b Scheduler Instance Name](#)
- [4.1.4 Step 4: Start the Cluster](#)
- [4.1.5 Step 5: Edit the Terracotta Configuration](#)
 - [4.1.5a Procedure:](#)
- [4.1.6 Step 6: Learn More](#)

[4.2 Quartz Scheduler Reference](#)

- [4.2.1 Quartz Scheduler Where \(Locality API\)](#)
 - [4.2.1a Installing Quartz Scheduler Where](#)
 - [4.2.1b Configuring Quartz Scheduler Where](#)
 - [4.2.1c Understanding Generated Node IDs](#)
 - [4.2.1d Available Constraints](#)
 - [4.2.1e Quartz Scheduler Where Code Sample](#)
 - [4.2.1f Locality With the Standard Quartz Scheduler API](#)
- [4.2.2 Execution of Jobs](#)
- [4.2.3 Working With JobDataMaps](#)
 - [4.2.3a Updating a JobDataMap](#)
 - [4.2.3b Best Practices for Storing Objects in a JobDataMap](#)
- [4.2.4 Cluster Data Safety](#)
- [4.2.5 Effective Scaling Strategies](#)

[5 Clustering Web Applications with Terracotta Web Sessions](#)

[5.0.1 Architecture of a Terracotta Cluster](#)

[5.1 Web Sessions Installation](#)

- [5.1.1 Step 1: Requirements](#)
- [5.1.2 Step 2: Install the Terracotta Sessions JAR](#)
- [5.1.3 Step 3: Configure Web-Session Clustering](#)
 - [5.1.3a Jetty, WebLogic, and WebSphere](#)
 - [5.1.3b JBoss AS and Tomcat](#)
- [5.1.4 Step 4: Start the Cluster](#)
- [5.1.5 Step 5: Edit the Terracotta Configuration](#)
 - [5.1.5a Procedure:](#)
- [5.1.6 Step 6: Learn More](#)

[5.2 Web Sessions Reference](#)

- [5.2.1 Additional Configuration Options](#)
 - [5.2.1a Session Locking](#)
 - [5.2.1b Synchronous Writes](#)
- [5.2.2 Troubleshooting](#)
 - [5.2.2a Sessions Time Out Unexpectedly](#)
 - [5.2.2b Changes Not Replicated](#)
 - [5.2.2c Tomcat 5.5 Messages Appear With Tomcat 6 Installation](#)
 - [5.2.2d Deadlocks When Session Locking Is Enabled](#)
 - [5.2.2e Events Not Received on Node](#)

[6 The Terracotta Server Array](#)

[6.1 Working with Terracotta Configuration Files](#)

- [6.1.1 How Terracotta Servers Get Configured](#)
 - [6.1.1a Default Configuration](#)
 - [6.1.1b Local XML File \(Default\)](#)
 - [6.1.1c Local or Remote Configuration File](#)
- [6.1.2 How Terracotta Clients Get Configured](#)
 - [6.1.2a Local or Remote XML File](#)
 - [6.1.2b Terracotta Server](#)
- [6.1.3 Configuration in a Development Environment](#)
 - [6.1.3a One-Server Setup in Development](#)
 - [6.1.3b Two-Server Setup in Development](#)
 - [6.1.3c Clients in Development](#)
- [6.1.4 Configuration in a Production Environment](#)
 - [6.1.4a Clients in Production](#)
- [6.1.5 Binding Ports to Interfaces](#)

- [6.1.6 terracotta.xml \(DSO only\)](#)
 - [6.1.7 Which Configuration?](#)
 - [6.2 Configuring Terracotta Clusters For High Availability](#)
 - [6.2.1 Common Causes of Failures in a Cluster](#)
 - [6.2.2 Basic High-Availability Configuration](#)
 - [6.2.3 High-Availability Features](#)
 - [6.2.3a HealthChecker](#)
 - [6.2.3b Automatic Server Instance Reconnect](#)
 - [6.2.3c Automatic Client Reconnect](#)
 - [6.2.3d Special Client Connection Properties](#)
 - [6.3 Terracotta Server Arrays](#)
 - [6.3.1 Definitions and Functional Characteristics](#)
 - [6.3.2 Server Array Configuration Tips](#)
 - [6.3.3 Backing Up Persisted Shared Data](#)
 - [6.3.4 Client Disconnection](#)
 - [6.3.5 Cluster Structure and Behavior](#)
 - [6.3.5a Terracotta Cluster in Development](#)
 - [6.3.5b Terracotta Cluster With Reliability](#)
 - [6.3.5c Terracotta Server Array with High Availability](#)
 - [6.3.5d Scaling the Terracotta Server Array](#)
 - [6.4 Improving Server Performance With BigMemory](#)
 - [6.4.1 How BigMemory Improves Performance](#)
 - [6.4.2 Requirements](#)
 - [6.4.3 Configuring BigMemory](#)
 - [6.4.3a Configuring Direct Memory Space](#)
 - [6.4.3b Configuring Off-Heap](#)
 - [6.4.3c Maximum, Minimum, and Default Values](#)
 - [6.4.4 Optimizing BigMemory](#)
 - [6.4.4a General Memory allocation](#)
 - [6.4.4b Compressed References](#)
 - [6.4.4c Swapiness and Huge Pages](#)
 - [6.5 Cluster Security](#)
 - [6.5.1 Configuring Security](#)
 - [6.5.1a How to Configure Security Using LDAP \(via JAAS\)](#)
 - [6.5.1b How to Configure Security Using JMX Authentication](#)
 - [6.5.2 Using Scripts Against a Server with Authentication](#)
 - [6.5.3 Extending Server Security](#)
 - [6.6 Changing Cluster Topology in a Live Cluster](#)
 - [6.6.1 Adding a New Server](#)
 - [6.6.2 Removing an Existing Server](#)
 - [6.6.3 Editing the Configuration of an Existing Server](#)
- [7 Developing Applications With the Terracotta Toolkit](#)
 - [7.0.1 Installing the Terracotta Toolkit](#)
 - [7.0.2 Understanding Versions](#)
 - [7.1 Working With the Terracotta Toolkit](#)
 - [7.1.1 Initializing the Toolkit](#)
 - [7.1.2 Using Toolkit Tools](#)
 - [7.1.2a Toolkit Data Structures and Serialization](#)
 - [7.1.2b Maps](#)
 - [7.1.2c Queues](#)
 - [7.1.2d Cluster Information](#)
 - [7.1.2e Locks](#)
 - [7.1.2f Clustered Barriers](#)
 - [7.1.2g Utilities](#)
 - [7.2 Terracotta Toolkit Reference](#)
 - [7.2.1 Client Failures](#)
 - [7.2.2 Connection Issues](#)
 - [7.2.3 Multiple Terracotta Clients in a Single JVM](#)
 - [7.2.3a Multiple Clients With a Single Web Application](#)
 - [7.2.3b Clients Sharing a Node ID](#)
- [8 Terracotta Cluster Tools](#)
 - [8.1 Terracotta Developer Console](#)
 - [8.1.1 Launching the Terracotta Developer Console](#)
 - [8.1.1a The Console Interface](#)
 - [8.1.1b Console Messages](#)
 - [8.1.1c Menus](#)
 - [8.1.1d Context-Sensitive Help](#)
 - [8.1.1e Context Menus](#)
 - [8.1.2 Working with Clusters](#)

- [8.1.2a Adding and Removing Clusters](#)
 - [8.1.2b Connecting to a cluster](#)
 - [8.1.2c Connecting to a Secured Cluster](#)
 - [8.1.2d Disconnecting from a Cluster](#)
- [8.1.3 Enterprise Ehcache Applications](#)
 - [8.1.3a Overview Panel](#)
 - [8.1.3b Performance Panel](#)
 - [8.1.3c Statistics Panel](#)
 - [8.1.3d Editing Cache Configuration](#)
- [8.1.4 Enterprise Ehcache for Hibernate Applications](#)
 - [8.1.4a Second-Level Cache View](#)
- [8.1.5 Clustered Quartz Scheduler Applications](#)
- [8.1.6 Clustered HTTP Sessions Applications](#)
- [8.1.7 Working with Terracotta Server Arrays](#)
 - [8.1.7a Server Panel](#)
 - [8.1.7b Connecting and Disconnecting from a Server](#)
 - [8.1.7c Server Connection Status](#)
- [8.1.8 Working with Clients](#)
 - [8.1.8a Client Panel](#)
 - [8.1.8b Connecting and Disconnecting Clients](#)
- [8.1.9 Monitoring Clusters, Servers, and Clients](#)
 - [8.1.9a Real-Time Performance Monitoring](#)
 - [8.1.9b Logs and Status Messages](#)
 - [8.1.9c Operator Events](#)
- [8.1.10 Advanced Monitoring and Diagnostics](#)
 - [8.1.10a Shared Objects](#)
 - [8.1.10b Lock Profiler](#)
- [8.1.11 Recording and Viewing Statistics](#)
 - [8.1.11a Cluster Statistics Recorder](#)
 - [8.1.11b Snapshot Visualization Tool](#)
- [8.1.12 Troubleshooting the Console](#)
 - [8.1.12a Cannot Connect to Cluster \(Console Times Out\)](#)
 - [8.1.12b Failure to Display Certain Metrics Hyperic \(Sigar\) Exception](#)
 - [8.1.12c Console Runs Very Slowly](#)
 - [8.1.12d Console Logs and Configuration File](#)
- [8.1.13 Backing Up Shared Data](#)
- [8.1.14 Update Checker](#)
- [8.1.15 Definitions of Cluster Statistics](#)
 - [8.1.15a cache objects evict request](#)
 - [8.1.15b cache objects evicted](#)
 - [8.1.15c l1 l2 flush](#)
 - [8.1.15d l2 faults from disk](#)
 - [8.1.15e l2 l1 fault](#)
 - [8.1.15f memory \(usage\)](#)
 - [8.1.15g vm garbage collector](#)
 - [8.1.15h distributed gc \(distributed garbage collection, or DGC\)](#)
 - [8.1.15i l2 pending transactions](#)
 - [8.1.15j stage queue depth](#)
 - [8.1.15k server transaction sequencer stats](#)
 - [8.1.15l network activity](#)
 - [8.1.15m l2 changes per broadcast](#)
 - [8.1.15n message monitor](#)
 - [8.1.15o l2 broadcast count](#)
 - [8.1.15p l2 transaction count](#)
 - [8.1.15q l2 broadcast per transaction](#)
 - [8.1.15r system properties](#)
 - [8.1.15s thread dump](#)
 - [8.1.15t disk activity](#)
 - [8.1.15u cpu \(usage\)](#)
- [8.2 Terracotta Tools Catalog](#)
 - [8.2.1 Terracotta Maven Plugin](#)
 - [8.2.2 TIM Management \(tim-get\)](#)
 - [8.2.3 Sessions Configurator \(sessions-configurator\)](#)
 - [8.2.4 Developer Console \(dev-console\)](#)
 - [8.2.5 Operations Center \(ops-center\)](#)
 - [8.2.6 Archive Utility \(archive-tool\)](#)
 - [8.2.7 Database Backup Utility \(backup-data\)](#)
 - [8.2.7a Using the Terracotta Operations Center](#)
 - [8.2.7b Example \(UNIX/Linux\)](#)
 - [8.2.8 Distributed Garbage Collector \(run-dgc\)](#)

8.2.8a	Further Reading
8.2.9	Start and Stop Server Scripts (start-tc-server, stop-tc-server)
8.2.9a	Further Reading
8.2.10	Version Utility (version)
8.2.11	Server Status (server-stat)
8.2.11a	Example
8.2.12	Cluster Statistics Recorder (tc-stats)
8.2.13	DSO Tools
8.2.13a	Sample Launcher (samples)
8.2.13b	Make Boot Jar Utility (make-boot-jar)
8.2.13c	Scan Boot Jar Utility (scan-boot-jar)
8.2.13d	Boot Jar Path Utility (boot-jar-path)
8.2.13e	DSO Environment Setter (dso-env)
8.2.13f	Java Wrapper (dso-java)
9	Terracotta DSO Installation
9.0.1	Standard Versus DSO Installations
9.0.2	Overview of Installation
9.1	Performing a DSO Installation
9.1.1	Prerequisites
9.1.1a	Enterprise Ehcache Users
9.1.1b	Quartz Scheduler Users
9.1.2	Step 1: Configure the Terracotta Platform
9.1.2a	TIMs for Clustering Enterprise Ehcache
9.1.2b	TIMs for Clustering Quartz Scheduler
9.1.2c	TIMs for Integrating an Application Server
9.1.2d	Clustering a Web Application with Terracotta Web Sessions
9.1.3	Step 2: Configure Terracotta Products
9.1.3a	Enterprise Ehcache Configuration
9.1.3b	Enterprise Ehcache for Hibernate Configuration
9.1.3c	Quartz Scheduler Configuration
9.1.3d	Web Sessions Configuration
9.1.4	Step 3: Install the TIMs
9.1.4a	Location of TIMs
9.1.5	Step 4: Start the Cluster
9.1.6	Quartz Scheduler DSO Installation

1 Getting Started

Terracotta product documentation focuses on the use of Terracotta products in a Terracotta cluster. Product documentation covers the following products and core components:

- Enterprise Ehcache - Standards-based Java cache. Separate chapters cover Enterprise Ehcache and Enterprise Ehcache for Hibernate.
- BigMemory - Massive boost for data in server memory without Java GC constraints. This product is covered in the Terracotta Server Array chapter. Documentation on BigMemory for Ehcache is covered by [Ehcache documentation](#).
- Quartz Scheduler - Scalable Java job scheduler. Chapter covers how to install and use the TerracottaJobStore for Quartz Scheduler.
- Web Sessions - Solution for clustering web sessions.
- Terracotta Server Array - The Terracotta platform forming the backbone of Terracotta clusters.

Terracotta products are Enterprise Edition (ee) versions of Terracotta software, also known as *commercial* versions. Users of the Terracotta Enterprise Suite have access to all of the listed products, including all of the features available with those products. Users of open-source versions of Terracotta software can also use this documentation, but have access to a limited number of products and features. See <http://www.terracotta.org/products> for more information.

1.0.1 Using Product Documentation

Product documentation is aimed at helping you quickly get a clustered application up and running. Start by choosing the installation section in the chapter for your Terracotta product. Following installation, or if you experience trouble, see the reference sections that follow the installation section.

NOTE: Standard and DSO Installations

The installation procedures given in the chapters on Terracotta products are the recommended standard installations. If you require object identity, must share non-serializable objects, or have other requirements that can only be met by using a cluster based on Terracotta Distributed Shared Objects (DSO), see the chapter on installing with DSO. DSO uses object identity, instrumented classes (byte-code instrumentation), object-graph roots, and cluster-wide locks to maintain data coherence.

The threshold for successfully setting up a DSO cluster can be substantially higher than for a non-DSO cluster due to DSO's stricter code and configuration requirements. It is recommended that if possible you use the standard installation (also called *express* installation) to set up a non-DSO cluster. Use the DSO installation only if your deployment requires the features of DSO.

You cannot combine a standard installation with a DSO installation.

To learn more about cluster configuration options, production architectures, security, High Availability, and performance optimization, see the chapter on Terracotta Server Arrays.

To learn more about using the tools available through the Terracotta API in your application, see the chapter on the Terracotta Toolkit. This chapter is appropriate for developers who want to integrate Terracotta functionality directly.

A number of useful Terracotta tools are available with the kit, including the Developer Console. See the chapter on tools for more information on Terracotta tools.

1.0.2 Using a License File

A Terracotta license file is required to run enterprise versions of Terracotta products. Note the following:

- The name of the file is `terracotta-license.key` and must not be changed.
- The number of Terracotta clients that can run simultaneously in the cluster is fixed by the file and cannot be changed without obtaining a new file.
- Trial versions of Terracotta enterprise products expire after a trial period. Expiration warnings are issued both to logs and standard output to allow enough time to contact Terracotta for an extension.

Each Terracotta client and server instance in your cluster requires a copy of the license file or configuration that specifies the file's location. By default, the file is provided in the root directory of the Terracotta kit. To avoid having to explicitly specify the file's location, you can leave it in the Terracotta kit's root directory. Or more generally, ensure that the resource `/terracotta-license.key` is on the same classpath as the Terracotta Toolkit runtime JAR. (The standard Terracotta Toolkit runtime JAR is included with the Terracotta kit. See the installation section in the chapter for your Terracotta product for more information on how to install this JAR file). For example, the license file could be placed in `WEB-INF/classes` when using a web application.

1.0.2a Explicitly Specifying the Location of the License File

If the file is in the Terracotta installation directory, you can specify it with:

```
-Dtc.install-root=/path/to/terracotta-install-dir
```

If the file is in a different location, you can specify it with:

```
-Dcom.tc.productkey.path=/path/to/terracotta-license.key
```

Alternatively, the path to the license file can be specified by adding the following to the beginning of the Terracotta configuration file (`tc-config.xml` by default):

```
<tc-properties>
  <property name="productkey.path" value="path/to/terracotta-license.key" />
  <!-- Other tc.properties here. -->
</tc-properties>
```

To refer to a license file that is in a WAR or JAR file, substitute `productkey.resource.path` for `productkey.path`.

1.0.2b Verifying Products and Features

There are a number of ways to verify what products and features are allowed and what limitations are imposed by your product key. The first is by looking at the readable file (`terracotta-license.key`) containing the product key.

Second, at startup Terracotta software logs a message detailing the product key. The message is printed to the log and to standard output. The message should appear similar to the following:

```
2010-11-03 15:56:53,701 INFO - Terracotta license loaded from /Downloads/terracotta-ee-3.4.0/terracotta-
license.key
Capabilities: DCV2, authentication, ehcache, ehcache monitor, ehcache offheap, operator console, quartz,
roots, server array offheap, server striping, sessions
Date of Issue: 2010-10-16
Edition: FX
Expiration Date: 2011-01-03
License Number: 0000
License Type: Trial
Licensee: Terracotta QA
Max Client Count: 100
Product: Enterprise Suite
ehcache.maxOffHeap: 200G
terracotta.serverArray.maxOffHeap: 200G
```


Terracotta server information panels in the Terracotta Developer Console and Terracotta Operations Center also contain license details.

1.0.3 Using Apache Maven

Apache Maven users can set up the Terracotta repository for Terracotta artifacts (including Ehcache, Quartz, and other Terracotta projects) using the URL shown:

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

A complete repository list is given below. Note the following when using Maven:

- The repository URL is not browsable.
- If you intend to work with Terracotta SNAPSHOT projects (usually in trunk), see [1.0.3c Working With Terracotta SNAPSHOT Projects](#).
- Coordinates for specific artifacts can be found by running tim-get with the `info` command:

UNIX/LINUX

```
${TERRACOTTA_HOME}/bin/tim-get.sh info <name of artifact>
```

MICROSOFT WINDOWS

```
%TERRACOTTA_HOME%\bin\tim-get.bat info <name of artifact>
```

You can generate a complete list of artifacts by running tim-get with the `list` command:

UNIX/LINUX

```
${TERRACOTTA_HOME}/bin/tim-get.sh list
```

MICROSOFT WINDOWS

```
%TERRACOTTA_HOME%\bin\tim-get.bat list
```

- You can use the artifact versions in a specific kit when configuring a POM. Artifacts in a specific kit are guaranteed to be compatible.

NOTE: Errors Caused by Outdated Dependencies

Certain frameworks, including Hibernate and certain Spring modules, may have POMs with dependencies on outdated versions of Terracotta products. This can cause older versions of Terracotta products to be installed in your application's classpath ahead of the current versions of those products, resulting in `NoClassDefFound`, `NoSuchMethod`, and other errors. At best, your application may run but not perform correctly. Be sure to locate and remove any outdated dependencies before running Maven.

1.0.3a Creating Enterprise Edition Clients

The following example shows the dependencies needed for creating Terracotta 3.4.0 ee clients, not clients based on the current Terracotta 3.5.0 kit. Version numbers can be found in the specific Terracotta kit you are installing. Be sure to update all artifactIds and versions to match those found in your kit.

```
<dependencies>
  <!-- The Terracotta Toolkit is required for running a client. The API version for this Toolkit is 1.1. -->
  <dependency>
    <groupId>org.terracotta</groupId>
    <artifactId>terracotta-toolkit-1.1-runtime-ee</artifactId>
    <version>2.0.0</version>
  </dependency>

  <!-- The following dependencies are required for using Ehcache. Dependencies not listed here include the SLF4J API JAR (version 1.5.11) and an SLF4J binding JAR of your choice. These JARs specify the logging framework required by Ehcache. It also does not include the explicit-locking JAR.-->
  <dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache-core-ee</artifactId>
    <version>2.3.0</version>
  </dependency>
</dependencies>
```

```

    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache-terracotta-ee</artifactId>
    <version>2.3.0</version>
  </dependency>

  <!-- The following dependencies are required for using Quartz Scheduler. -->
  <dependency>
    <groupId>org.quartz</groupId>
    <artifactId>quartz</artifactId>
    <version>1.8.4</version>
  </dependency>
  <dependency>
    <groupId>org.quartz</groupId>
    <artifactId>quartz-terracotta</artifactId>
    <version>1.2.1</version>
  </dependency>

  <!-- The following dependencies are required for using Terracotta Sessions. -->
  <dependency>
    <groupId>org.terracotta</groupId>
    <artifactId>terracotta-session</artifactId>
    <version>1.1.1</version>
  </dependency>
</dependencies>

```

Open-source clients can be created using non-ee artifacts.

1.0.3b Using the tc-maven Plugin

The tc-maven plugin can simplify the process of integrating and testing Terracotta products and other assets. The plugin supplies a number of useful tasks, including starting, stopping, and pausing Terracotta servers. To integrate the plugin, add the following to your project's POM:

```

<plugin>
  <groupId>org.terracotta.maven.plugins</groupId>
  <artifactId>tc-maven-plugin</artifactId>
  <version>1.6.1</version>
</plugin>

```

If you are using the tc-maven plugin with an ee kit, you must have the `terracotta-ee-<version>.jar` file in your project. This JAR file is not available from a public repository. You must obtain it from your Terracotta representative and install it to your local repository. For example, to install version 3.4.0 of this JAR file:

```

mvn install:install-file -Dfile=terracotta-ee-3.5.0.jar -DpomFile=terracotta-ee-3.5.0.pom
-Dpackaging=jar -Dversion=3.5.0

```

This command format assumes that the JAR and POM files are available in the local directory. If they are not you must provide the files' paths as well.

1.0.3c Working With Terracotta SNAPSHOT Projects

If you intend to work with Terracotta SNAPSHOT projects (usually in trunk), you must have the following settings.xml file installed:

```

<settings xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <profiles>
    <profile>
      <id>terracotta-repositories</id>
      <repositories>
        <repository>
          <id>terracotta-snapshots</id>
          <url>http://www.terracotta.org/download/reflector/snapshots</url>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>terracotta-snapshots</id>
          <url>http://www.terracotta.org/download/reflector/snapshots</url>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
</settings>

```

```

</profiles>
<activeProfiles>
<activeProfile>terracotta-repositories</activeProfile>
</activeProfiles>
</settings>

```

1.0.3d Terracotta Repositories

The following contains all of the Terracotta repositories available:

```

<repositories>
  <repository>
    <id>terracotta-snapshots</id>
    <url>http://www.terracotta.org/download/reflector/snapshots</url>
    <releases><enabled>false</enabled></releases>
    <snapshots><enabled>true</enabled></snapshots>
  </repository>
  <repository>
    <id>terracotta-releases</id>
    <url>http://www.terracotta.org/download/reflector/releases</url>
    <releases><enabled>true</enabled></releases>
    <snapshots><enabled>false</enabled></snapshots>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <id>terracotta-snapshots</id>
    <url>http://www.terracotta.org/download/reflector/snapshots</url>
    <releases><enabled>false</enabled></releases>
    <snapshots><enabled>true</enabled></snapshots>
  </pluginRepository>
  <pluginRepository>
    <id>terracotta-releases</id>
    <url>http://www.terracotta.org/download/reflector/releases</url>
    <releases><enabled>true</enabled></releases>
    <snapshots><enabled>false</enabled></snapshots>
  </pluginRepository>
</pluginRepositories>

```

2 Enterprise Ehcache With Terracotta Clustering

Enterprise Ehcache with Terracotta clustering combines the power of the Terracotta platform with the ease of Ehcache application-data caching. By integrating Enterprise Ehcache with the Terracotta platform, you can:

- linearly scale your application to grow with requirements;
- rely on data that remains consistent across the cluster;
- offload databases to reduce the associated overhead;
- increase application performance with distributed in-memory data.

To install Enterprise Ehcache, see [Enterprise Ehcache Installation](#).

2.1 Enterprise Ehcache Installation

This document shows you how to add Terracotta clustering to an application that is using Ehcache.

Use this express installation if you have been running your application:

- on a single JVM, or
- on a cluster using Ehcache replication.

To set up the cluster with Terracotta, you will add a Terracotta JAR to each application and run a Terracotta server array. Except as noted below, you can continue to use Ehcache in your application as specified in the [Ehcache documentation](#).

To add Terracotta clustering to an application that is using Ehcache, follow these steps:

2.1.1 Step 1: Requirements

- JDK 1.5 or higher.

- [Terracotta 3.5.0](#)
Download the kit and run the installer on the machine that will host the Terracotta server.
- All clustered objects must be serializable.
If you cannot use Serializable classes, you must use an identity cache with a custom installation (see [Terracotta DSO Installation](#)). Identity cache, which requires DSO, is not supported with this installation.

2.1.2 Step 2: Install the Distributed Cache

For guaranteed compatibility, use the JAR files included with the Terracotta kit you are installing. Mixing with older components may cause errors or unexpected behavior.

To install the distributed cache in your application, add the following JAR files to your application's classpath:

- `${TERRACOTTA_HOME}/ehcache/lib/ehcache-terracotta-ee-<version>.jar`
<version> is the current version of the Ehcache-Terracotta JAR.
- `${TERRACOTTA_HOME}/ehcache/lib/ehcache-core-ee-<ehcache-version>.jar`
- The Ehcache core libraries, where <ehcache-version> is the current version of Ehcache (2.4.1 or higher).
- `${TERRACOTTA_HOME}/ehcache/lib/slf4j-api-<slf4j-version>.jar`
The SLF4J logging facade allows Ehcache to bind to any supported logger used by your application. Binding JARs for popular logging options are available from the [SLF4J project](#). For convenience, the binding JAR for `java.util.logging` is provided in `${TERRACOTTA_HOME}/ehcache` (see below).
- `${TERRACOTTA_HOME}/ehcache/lib/slf4j-jdk14-<slf4j-version>.jar`
An SLF4J binding JAR for use with the standard `java.util.logging`, also known as JDK 1.4 logging.
- `${TERRACOTTA_HOME}/common/terracotta-toolkit-<API-version>-runtime-ee-<version>.jar`
The Terracotta Toolkit JAR contains the Terracotta client libraries. <API-version> refers to the Terracotta Toolkit API version. <version> is the current version of the Terracotta Toolkit JAR.

If you are using the open-source edition of the Terracotta kit, no JAR files will have "-ee-" as part of their name.

If you are using a WAR file, add these JAR files to its `WEB-INF/lib` directory.

NOTE: Application Servers

Most application servers (or web containers) should work with this installation of Enterprise Ehcache. However, note the following:

- GlassFish application server - You must add the following to `domains.xml` :

```
<jvm-options>-Dcom.sun.enterprise.server.ss.ASQuickStartup=false</jvm-options>
```

- WebLogic application server - You must use the [supported version](#) of WebLogic.

2.1.3 Step 3: Configure the Distributed Cache

The Ehcache configuration file, `ehcache.xml` by default, must be on your application's classpath. If you are using a WAR file, add the Ehcache configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.

Create a basic Ehcache configuration file called `ehcache.xml` :

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache name="myCache"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ehcache.xsd">
  <defaultCache
    maxElementsInMemory="0"
    eternal="false"
    timeToIdleSeconds="1200"
    timeToLiveSeconds="1200">
    <terracotta />
  </defaultCache>
  <terracottaConfig url="localhost:9510" />
</ehcache>
```

This defaultCache configuration includes Terracotta clustering. The Terracotta client must load a Terracotta configuration (separate from the Ehcache configuration) from a file or a Terracotta server. The value of the `<terracottaConfig />` element's `url` attribute should contain a path to that file or to the address and DSO port (9510 by default) of a server. In the example value, "localhost:9510" means that the Terracotta server is on the local host. If the Terracotta configuration source changes at a later time, it must be updated in configuration.

TIP: Terracotta Clients and Servers

In a Terracotta cluster, the application server is also known as the client.

2.1.3a Add Terracotta to Specific Caches

For any cache that should be clustered by Terracotta, add the sub-element `<terracotta />` to that cache's `<cache>` block in `ehcache.xml`. For example, the following cache is clustered with Terracotta:

```
<cache name="myCache" maxElementsInMemory="1000"
      maxElementsOnDisk="10000" eternal="false" timeToIdleSeconds="3600"
      timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
  <!-- Adding the element <terracotta /> turns on Terracotta clustering for the cache myCache. -->
  <terracotta />
</cache>
```

See [2.3.1e Controlling Cache Size](#) for information on using configuration to tune memory and disk storage limits.

2.1.3b Edit Incompatible Configuration

For any clustered cache, you must delete, disable, or edit configuration elements that are incompatible when clustering with Terracotta. Clustered caches have a `<terracotta>` or `<terracotta clustered="true">` element.

The following Ehcache configuration attributes or elements should be deleted or disabled:

- DiskStore-related attributes `overflowToDisk` and `diskPersistent`.
The Terracotta server automatically provides a disk store.
- Replication-related attributes such as `replicateAsynchronously` and `replicatePuts`.
- The attribute `MemoryStoreEvictionPolicy` must be set to either LFU or LRU.
Setting `MemoryStoreEvictionPolicy` to FIFO causes the error `IllegalArgumentException`.

2.1.4 Step 4: Start the Cluster

1. Start the Terracotta server:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat
```

2. Start the application servers.
3. Start the Terracotta Developer Console:

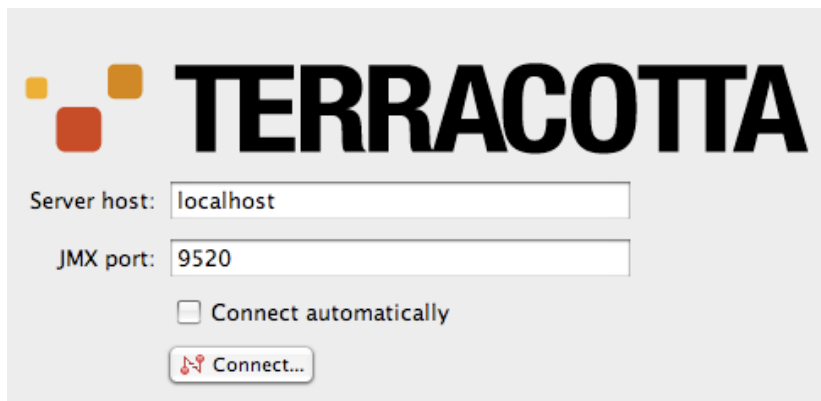
UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

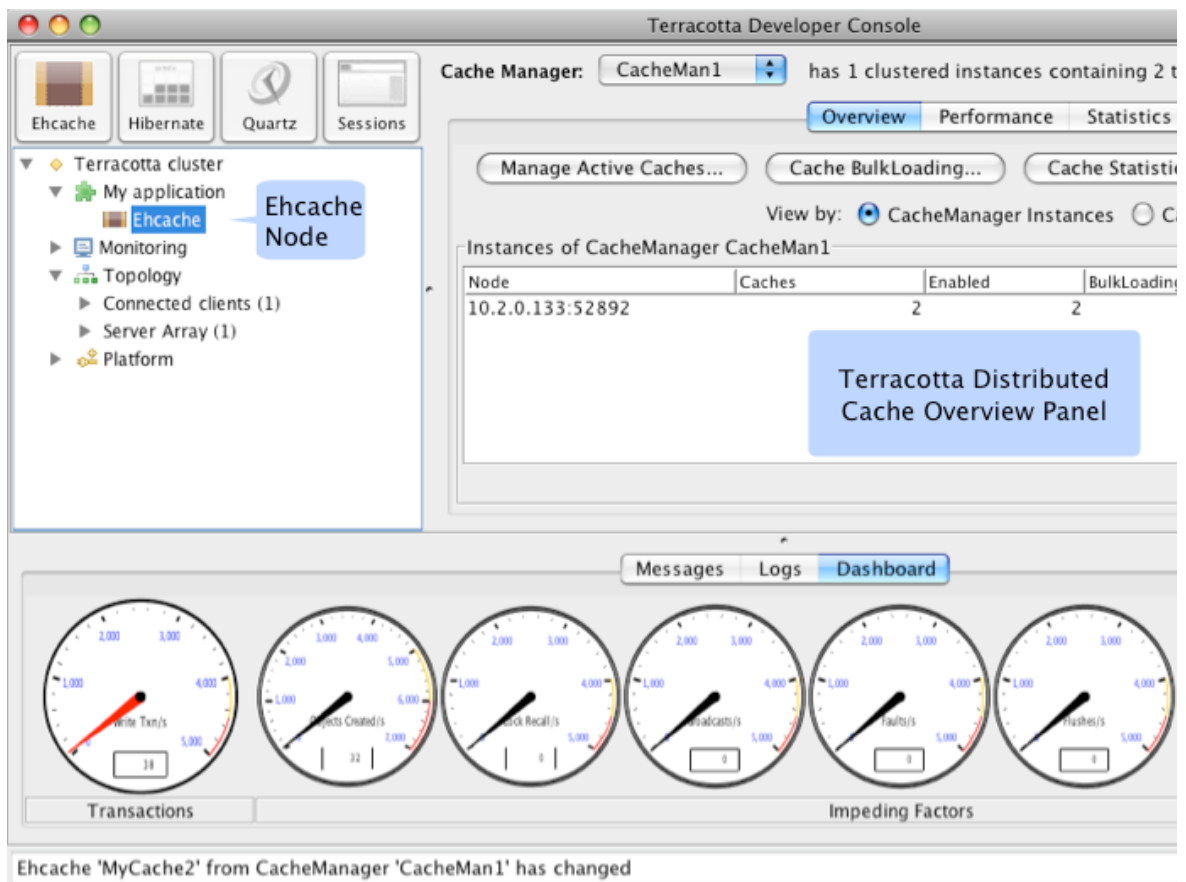
Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

4. Connect to the Terracotta cluster.
Click **Connect...** in the Terracotta Developer Console.



5. Click the **Ehcache** node in the cluster navigation window to see the caches in the Terracotta cluster. Your console should have a similar appearance to the following annotated figure.



2.1.5 Step 5: Edit the Terracotta Configuration

This step shows you how to run clients and servers on separate machines and add failover (High Availability). You will expand the Terracotta cluster and add High Availability by doing the following:

- Moving the Terracotta server to its own machine
- Creating a cluster with multiple Terracotta servers
- Creating multiple application nodes

These tasks bring your cluster closer to a production architecture.

2.1.5a Procedure:

1. Shut down the Terracotta cluster.
2. Create a Terracotta configuration file called `tc-config.xml` with contents similar to the following

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- All content copyright Terracotta, Inc., unless otherwise indicated. All rights reserved. -->
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd"
```

```

xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <servers>
    <!-- Sets where the Terracotta server can be found. Replace the value of host with the server's IP
    address. -->
    <server host="server.1.ip.address" name="Server1">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
    <!-- If using a standby Terracotta server, also referred to as an ACTIVE-PASSIVE configuration, add
    the second server here. -->
    <server host="server.2.ip.address" name="Server2">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
    <!-- If using more than one server, add an <ha> section. -->
    <ha>
      <mode>networked-active-passive</mode>
      <networked-active-passive>
        <election-time>5</election-time>
      </networked-active-passive>
    </ha>
  </servers>
  <!-- Sets where the generated client logs are saved on clients. Note that the exact location of
  Terracotta logs on client machines may vary based on the value of user.home and the local disk layout.
  -->
  <clients>
    <logs>%(user.home)/terracotta/client-logs</logs>
  </clients>
</tc:tc-config>

```

3. Install Terracotta 3.5.0 on a separate machine for each server you configure in `tc-config.xml`.
4. Copy the `tc-config.xml` to a location accessible to the Terracotta servers.
5. Perform [Step 2: Install the Distributed Cache](#) and [Step 3: Configure the Distributed Cache](#) on each application node you want to run in the cluster.
Be sure to install your application and any application servers on each node.
6. Add the following to the Ehcache configuration file, `ehcache.xml`:

```

<!-- Add the servers that are configured in tc-config.xml. -->
<terracottaConfig url="server.1.ip.address:9510,server.2.ip.address:9510" />

```

7. Copy `ehcache.xml` to each application node and ensure that it is on your application's classpath. If you are using a WAR file, add the Ehcache configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.
8. Start the Terracotta server in the following way, replacing "Server1" with the name you gave your server in `tc-config.xml`:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh -f <path/to/tc-config.xml> -n Server1 &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat -f <path/to/tc-config.xml> -n Server1 &
```

If you configured a second server, start that server in the same way on its machine, entering its name after the `-n` flag. The second server to start up becomes the "hot" standby, or PASSIVE. Any other servers you configured will also start up as standby servers.

9. Start all application servers.
10. Start the Terracotta Developer Console and view the cluster.

2.1.6 Step 6: Learn More

To learn more about using Terracotta Ehcache distributed cache, start with the following document:

- [Enterprise Ehcache Configuration Reference](#)
- [General Ehcache documentation](#)

To learn more about working with a Terracotta cluster, see the following documents:

- [6.1 Working with Terracotta Configuration Files](#) - Explains how `tc-config.xml` is propagated and

loaded in a Terracotta cluster in different environments.

- [6.3 Terracotta Server Arrays](#) - Shows how to design Terracotta clusters that are fault-tolerant, maintain data safety, and provide uninterrupted uptime.
- [6.2 Configuring Terracotta Clusters For High Availability](#) - Defines High Availability configuration properties and explains how to apply them.
- [8.1 Terracotta Developer Console](#) - Provides visibility into and control of caches.

2.2 Enterprise Ehcache API Guide

Enterprise Ehcache has a rich API for extending your application's capabilities.

2.2.1 Enterprise Ehcache Search API For Clustered Caches

Enterprise Ehcache Search is a powerful search API for querying clustered caches in a Terracotta cluster. Designed to be easy to integrate with existing projects, the Ehcache Search API can be implemented with configuration or programmatically. The following is an example from an Ehcache configuration file:

```
<cache name="myCache" maxElementsInMemory="0" eternal="true"
  overflowToDisk="false">
  <searchable>
    <searchAttribute name="age" />
    <searchAttribute name="first_name" expression="value.getFirstName()" />
    <searchAttribute name="last_name" expression="value.getLastName()" />
    <searchAttribute name="zip_code" expression="value.getZipCode()" />
  </searchable>
</cache>
```

By default, the storageStrategy used by myCache is "DCV2". Enterprise Ehcache Search cannot be used with caches that have the "classic" storageStrategy.

2.2.1a Sample Code

The following example assumes there is a Person class that serves as the value for elements in myCache. With the exception of "age" (which is bean style), each expression attribute in searchAttribute is set to use an accessor method on the cache element's value. The Person class must have accessor methods to match the configured expressions. In addition, assume that there is code that populates the cache. Here is an example of search code based on these assumptions:

```
// After CacheManager and Cache created, create a query for myCache:

Query query = myCache.createQuery();

// Create the Attribute objects.

Attribute<String> last_name = myCache.getSearchAttribute("last_name");
Attribute<Integer> zip_code = myCache.getSearchAttribute("zip_code");
Attribute<Integer> age = myCache.getSearchAttribute("age");

// Specify the type of content for the result set.
// Executing the query without specifying desired results
// returns no results even if there are hits.

query.includeKeys(); // Return the keys for values that are hits.

// Define the search criteria.
// This following uses Criteria.and() to set criteria to find adults
// with the last name "Marley" whose address has the zip code "94102".

query.addCriteria(last_name.eq("Marley").and(zip_code.eq(94102)));

// Execute the query, putting the result set
// (keys to element that meet the search criteria) in Results object.

Results results = query.execute();

// Find the number of results -- the number of hits.

int size = results.size();
// Discard the results when done to free up cache resources.
```



```

results.discard();

// Using an aggregator in a query to get an average age of adults:

Query averageAgeOfAdultsQuery = myCache.createQuery();
averageAgeOfAdultsQuery.addCriteria(age.ge(18));
averageAgeOfAdultsQuery.includeAggregator(age.average());
Results averageAgeOfAdults = averageAgeOfAdultsQuery.execute();

If (averageAgeOfAdults.size() > 0) {
    List aggregateResults = averageAgeOfAdults.all().iterator().next().getAggregatorResults();
    double averageAge = (Double) aggregateResults.get(0);
}

```

The following example shows how to programmatically create the cache configuration, with search attributes.

```

Configuration cacheManagerConfig = new Configuration();

CacheConfiguration cacheConfig = new CacheConfiguration("myCache", 0).eternal(true);

Searchable searchable = new Searchable();
cacheConfig.addSearchable(searchable);

// Create attributes to use in queries.
searchable.addSearchAttribute(new SearchAttribute().name("age"));

// Use an expression for accessing values.
searchable.addSearchAttribute(new SearchAttribute()
    .name("first_name")
    .expression("value.getFirstName()"));

searchable.addSearchAttribute(new SearchAttribute().name("last_name").expression("value.getLastName()"))
searchable.addSearchAttribute(new
SearchAttribute().name("zip_code").expression("value.getZipCode()"));

cacheManager = new CacheManager(cacheManagerConfig);
cacheManager.addCache(new Cache(cacheConfig));

Ehcache myCache = cacheManager.getEhcache("myCache");

// Now create the attributes and queries, then execute.
...

```

To learn more about the Ehcache Search API, see the `net.sf.ehcache.search.*` packages in this [Javadoc](#).

2.2.1b Stored Search Indexes

Searches occur on indexes held by the Terracotta server. By default, index files are stored in `/index` under the server's data directory. However, you can specify a different path using the `<index>` element:

```

...
<server>
  <data>%(user.home)/terracotta/server-data</data>
  <index>%(user.home)/terracotta/index</index>
  <logs>%(user.home)/terracotta/server-logs</logs>
  <statistics>%(user.home)/terracotta/server-statistics</statistics>
...
</server>
...

```

To enhance performance, it is recommended that you store server data and search indexes on different disks.

2.2.1c Troubleshooting Ehcache Search

This section contains information on avoiding potential issues with using the Ehcache Search API.

Large Result Sets

If your query returns a result set containing a very large amount of data, performance degradation or errors (such as OOME) could occur due to network and resource limitations. You can manage the size of result sets

by following these best practices:

- Limit the size of the results set with `Query.maxResults(int number_of_hits)`.
- Page the results set using `Results.range(int start_index, int number_of_hits)`.
- Use a built-in Aggregator function to return a summary statistic (see the `net.sf.ehcache.search.aggregator` package in this [Javadoc](#)).

Inconsistent Data

Ehcache Search guarantees that all local changes made before a query is executed are available to search results. However, to maintain a high level of performance, cluster locks are not checked. This *eventual* synchronization of clustered caches and search indexes means that remote changes may not be available to queries until the changes are applied cluster wide (or, for transactional caches, `commit()` is called). Thus it is possible to get results that include removed elements, inconsistent data from the same query executed at different times, and calculated values that are no longer accurate (such as those returned by aggregators).

You can take certain precautions to prevent these types of problems. For example, if your search uses aggregators, add all aggregators to the same query to get consistent data. If your code attempts to get value using keys returned by a query, use null guards.

Slow to Return

Search strings leading with the wildcard character asterisk ("*") or question mark ("?") can be very slow to return results.

Failures

Failure to return is handled by a `CacheException`. If an exception occurs when the value extractor (either expression-based or custom) executes, the affected attribute value is omitted from the search index.

2.2.2 Enterprise Ehcache Cluster Events

The Enterprise Ehcache cluster events API provides access to Terracotta cluster events and cluster topology.

2.2.2a Cluster Topology

The interface `net.sf.ehcache.cluster.CacheCluster` provides methods for obtaining topology information for a Terracotta cluster. The following table lists these methods.

Method	Definition
<code>String getScheme()</code>	Returns a scheme name for the cluster information. Currently TERRACOTTA is the only scheme supported. The scheme name is used by <code>CacheManager.getCluster()</code> to return cluster information (see Events API Example Code).
<code>Collection<ClusterNode> getNodes()</code>	Returns information on all the nodes in the cluster, including ID, hostname, and IP address.
<code>boolean addTopologyListener(ClusterTopologyListener listener)</code>	Adds a cluster-events listener. Returns true if the listener is already active.
<code>boolean removeTopologyListener(ClusterTopologyListener listener)</code>	Removes a cluster-events listener. Returns true if the listener is already inactive.

The interface `net.sf.ehcache.cluster.ClusterNode` provides methods for obtaining information on specific Terracotta nodes in the cluster. The following table lists these methods.

Method	Definition
<code>getId()</code>	Returns the unique ID assigned to the node.

<code>getHostname()</code>	Return the hostname on which the node is running.
<code>getIp()</code>	Return the IP address on which the node is running.

2.2.2b Cluster Events

The interface `net.sf.ehcache.cluster.ClusterTopologyListener` provides methods for detecting the following cluster events:

- `nodeJoined(ClusterNode)`
- `nodeLeft(ClusterNode)`
- `clusterOnline(ClusterNode)`
- `clusterOffline(ClusterNode)`

2.2.2c Events API Example Code

```
// Get cluster data
CacheManager mgr = new CacheManager(); // Local ehcache.xml exists, with at least one cache configured
with Terracotta clustering.
CacheCluster cluster = mgr.getCluster("TERRACOTTA");
```

NOTE: Programmatic Creation of CacheManager

If a `CacheManager` instance is created and configured programmatically (without an `ehcache.xml` or other external configuration resource), `getCluster("TERRACOTTA")` may return null even if a Terracotta cluster exists. To ensure that cluster information is returned in this case, get a cache that is clustered with Terracotta:

```
// mgr created and configured programmatically.
CacheManager mgr = new CacheManager();
// myCache has Terracotta clustering.
Cache cache = mgr.getEhcache("myCache");
// A Terracotta client has started, making available cluster information.
CacheCluster cluster = mgr.getCluster("TERRACOTTA");

// Get current nodes
Collection<ClusterNode> nodes = cluster.getNodes();
for(ClusterNode node : nodes) {
    System.out.println(node.getId() + " " + node.getHostname() + " " + node.getIp());
}

// Register listener
cluster.addTopologyListener(new ClusterTopologyListener() {
    public void nodeJoined(ClusterNode node) { System.out.println(node + " joined"); }
    public void nodeLeft(ClusterNode node) { System.out.println(node + " left"); }
    public void clusterOnline(ClusterNode node) { System.out.println(node + " enabled"); }
    public void clusterOffline(ClusterNode node) { System.out.println(node + " disabled"); }
});
```

2.2.3 Bulk-Load API

The Enterprise Ehcache bulk-load API can optimize bulk-loading of caches by removing the requirement for locks and adding transaction batching. The bulk-load API also allows applications to discover whether a cache is in bulk-load mode and to block based on that mode.

NOTE: The Bulk-Load API and the Configured Consistency Mode

The initial consistency mode of a cache is set by configuration and cannot be changed programmatically (see the attribute "consistency" in [<terracotta>](#)). The bulk-load API should be used for temporarily suspending the configured consistency mode to allow for bulk-load operations.

The following table lists the bulk-load API methods that are available in `org.terracotta.modules.ehcache.Cache`.

Method	Definition
<code>public boolean isClusterBulkLoadEnabled()</code>	Returns true if a cache is in bulk-load mode (<i>is not</i> consistent) throughout the cluster. Returns false if the cache is not in bulk-load mode (<i>is</i> consistent) anywhere in the cluster.

<pre>public boolean isNodeBulkLoadEnabled()</pre>	<p>Returns true if a cache is in bulk-load mode (is <i>not</i> consistent) on the current node. Returns false if the cache is not in bulk-load mode (<i>is</i> consistent) on the current node.</p>
<pre>public void setNodeBulkLoadEnabled(boolean)</pre>	<p>Sets a cache's consistency mode to the configured mode (false) or to bulk load (true) on the local node. There is no operation if the cache is already in the mode specified by <code>setNodeBulkLoadEnabled()</code> . When using this method on a nonstop cache , a multiple of the nonstop cache's timeout value applies. The bulk-load operation must complete within that timeout multiple to prevent the configured nonstop behavior from taking effect. For more information on tuning nonstop timeouts, see Tuning Nonstop Timeouts and Behaviors.</p>
<pre>public void waitUntilBulkLoadComplete()</pre>	<p>Waits until a cache is consistent before returning. Changes are automatically batched and the cache is updated throughout the cluster. Returns immediately if a cache is consistent throughout the cluster.</p>

Note the following about using bulk-load mode:

- Consistency cannot be guaranteed because `isClusterBulkLoadEnabled()` can return false in one node just before another node calls `setNodeBulkLoadEnabled(true)` on the same cache. Understanding exactly how your application uses the bulk-load API is crucial to effectively managing the integrity of cached data.
- If a cache is not consistent, any `ObjectNotFound` exceptions that may occur are logged.
- `get()` methods that fail with `ObjectNotFound` return null.
- Eviction is independent of consistency mode. Any configured or manually executed eviction proceeds unaffected by a cache's consistency mode.

2.2.3a Bulk-Load API Example Code

The following example code shows how a clustered application with Enterprise Ehcache can use the bulk-load API to optimize a bulk-load operation:

```
import net.sf.ehcache.Cache;

public class MyBulkLoader {
    CacheManager cacheManager = new CacheManager(); // Assumes local ehcache.xml.
    Cache cache = cacheManager.getEhcache("myCache"); // myCache defined in ehcache.xml.
    cache.setNodeBulkLoadEnabled(true); // myCache is now in bulk mode.

    // Load data into myCache.

    cache.setNodeBulkLoadEnabled(false); // Done, now set myCache back to its configured consistency mode
}
```

On another node, application code that intends to touch myCache can run or wait, based on whether myCache is consistent or not:

```
...
if (!cache.isClusterBulkLoadEnabled()) {

    // Do some work.
}

else {

    cache.waitUntilBulkLoadComplete()
    // Do the work when waitUntilBulkLoadComplete() returns.
}
...
```

Waiting may not be necessary if the code can handle potentially stale data:

```
...
    if (!cache.isClusterBulkLoadEnabled()) {

        // Do some work.
    }

    else {

        // Do some work knowing that data in myCache may be stale.

    }
...

```

2.2.4 Unlocked Reads for Consistent Caches (UnlockedReadsView)

Certain environments require consistent cached data while also needing to provide optimized reads of that data. For example, a financial application may need to display account data as a result of a large number o requests from web clients. The performance impact of these requests can be reduced by allowing unlocked reads of an otherwise locked cache.

In cases where there is tolerance for getting potentially stale data, an unlocked (inconsistent) reads view can be created for Cache types using the UnlockedReadsView decorator. UnlockedReadsView requires Ehcache 2.1 or higher. The underlying cache must have Terracotta clustering and use the strong consistency mode. For example, the following cache can be decorated with UnlockedReadsView:

```
<cache name="myCache"
      maxElementsInMemory="500"
      eternal="false"
      overflowToDisk="false"
      <terracotta clustered="true" consistency="strong" />
</cache>

```

You can create an unlocked view of myCache programmatically:

```
Cache cache = cacheManager.getEhcache("myCache");
UnlockedReadsView unlockedReadsView = new UnlockedReadsView(cache, "myUnlockedCache");

```

The following table lists the API methods available with the decorator

net.sf.ehcache.constructs.unlockedreadsvie.UnlockedReadsView .

Method	Definition
public String getName()	Returns the name of the unlocked cache view.
public Element get(final Object key) public Element get(final Serializable key)	Returns the data under the given key. Returns null if data has expired.
public Element getQuiet(final Object key) public Element getQuiet(final Serializable key)	Returns the data under the given key without updating cache statistics. Returns null if data has expired.

2.2.4a UnlockedReadsView and Data Freshness

By default, caches have the following attributes set as shown:

```
<cache ... copyOnRead="true" ... >
...
    <terracotta ... consistency="strong" storageStrategy="DCV2" ... />
...
</cache>

```

Default settings are designed to make distributed caches more efficient and consistent in most use cases.

2.2.5 Explicit Locking

The explicit locking methods for Enterprise Ehcache provide simple key-based locking that preserves

concurrency while also imposing cluster-wide consistency. If certain operations on cache elements must be locked, use the explicit locking methods available in the Cache type.

The explicit locking methods are listed in the following table:

public void acquireReadLockOnKey(Object key)	Set a read lock on the element specified by the argument (key).
public void acquireWriteLockOnKey(Object key)	Set a write lock on the element specified by the argument (key).
public void releaseReadLockOnKey(Object key)	Remove a read lock from the element specified by the argument (key).
public void releaseWriteLockOnKey(Object key)	Remove a write lock from the element specified by the argument (key).

The following example shows how to use explicit locking methods:

```
String key1 = "123";
Foo val1 = new Foo();
cache.acquireWriteLockOnKey(key1);
try {
    cache.put(new Element(key1, val1));
} finally {
    cache.releaseWriteLockOnKey(key1);
}

// Now safely read val1.
cache.acquireReadLockOnKey(key1);
try {
    Object cachedVal1 = cache.get(key1).getValue();
} finally {
    cache.releaseReadLockOnKey(key);
}
```

For locking available through the Terracotta Toolkit API, see [7.1.2e Locks](#).

2.2.6 Configuration Using the Fluent Interface

You can configure clustered CacheManagers and caches using the fluent interface as follows:

```
...
Configuration configuration =
new Configuration().terracotta(newTerracottaClientConfiguration()
    .url("localhost:9510")
        // == <terracottaConfig url="localhost:9510 />
    .defaultCache(new CacheConfiguration("defaultCache", 100))
        // == <defaultCache maxElementsInMemory="100" ... />
    .cache(new CacheConfiguration("example", 100)
        // == <cache name="example" maxElementsInMemory="100" ... />
    .timeToIdleSeconds(5)
    .timeToLiveSeconds(120)
        // added these TTI and TTL attributes to the cache "example"
    .terracotta(new TerracottaConfiguration()));
    // added <terracotta /> element in the cache "example"

// Pass the configuration to the CacheManager.
this.cacheManager = new CacheManager(configuration);
...
```

2.2.7 Write-Behind Queue in Enterprise Ehcache

If your application uses the write-behind API with Ehcache and you cluster Ehcache with Terracotta, the write-behind queue automatically becomes a clustered write-behind queue. The clustered write-behind queue features the following characteristics:

- Atomic - Put and remove operations are guaranteed to succeed or fail. Partial completion of transactions cannot occur.

- Distributable - Work is distributable among nodes in the cluster.
- Durable - Terracotta clustering guarantees that a lost node does not result in lost data. Terracotta servers automatically ensure that another node receives the queued data belonging to the lost node.
- Performance enhancement - Asynchronous writes reduce the load on databases.

The write-behind queue is enabled for a cache with the `<cacheWriter />` element. For example:

```
<cache name="myCache" eternal="false" maxElementsInMemory="1000" overflowToDisk="false">
  <cacheWriter writeMode="write_behind" maxWriteDelay="8" rateLimitPerSecond="5"
writeCoalescing="true" writeBatching="true" writeBatchSize="20" writeBehindMaxQueueSize="500"
retryAttempts="2" retryAttemptDelaySeconds="2">
    <cacheWriterFactory class="com.company.MyCacheWriterFactory"
        properties="just.some.property=test; another.property=test2"
propertySeparator=";" />
  </cacheWriter>
</cache>
```

Values for `<cacheWriter />` attributes can also be set programmatically. For example, the value for `writeBehindMaxQueueSize`, which sets the maximum number of pending writes (the maximum number of elements that can be waiting in the queue for processing), can be set with `net.sf.ehcache.config.CacheWriterConfiguration.setWriteBehindMaxQueueSize (`

See the [Ehcache documentation](#) for more information on the write-behind API and on using synchronous write-through caching.

2.3 Enterprise Ehcache Configuration Reference

Enterprise Ehcache uses the standard Ehcache configuration file to set clustering and consistency behavior, optimize cached data, integrate with JTA and OSGi, and more.

2.3.1 Ehcache Configuration File

The Enterprise Ehcache configuration file (`ehcache.xml` by default) contains the configuration for one instance of a `CacheManager` (the Ehcache class managing a set of defined caches). This configuration file must be in your application's classpath to be found. When using a WAR file, `ehcache.xml` should be copied to `WEB-INF/classes`.

TIP: Naming the CacheManager

If you employ multiple Ehcache configuration files, use the `name` attribute in the `<ehcache>` element to identify specific `CacheManagers` in the cluster. The Terracotta Developer Console provides a menu listing these names, allowing you to choose the `CacheManager` you want to view.

Note the following about `ehcache.xml` in a Terracotta cluster:

- The copy on disk is loaded into memory from the first Terracotta client (also called application server or node) to join the cluster.
- Once loaded, the configuration is persisted in memory by the Terracotta servers in the cluster and survives client restarts.
- In-memory configuration can be edited in the Terracotta Developer Console. Changes take effect immediately but are *not* written to the original on-disk copy of `ehcache.xml`.
- The in-memory cache configuration is removed with server restarts if the servers are in [non-persistent mode](#), which is the default. The original (on-disk) `ehcache.xml` is loaded.
- The in-memory cache configuration survives server restarts if the servers are in [persistent mode](#) (default is non-persistent). If you are using the Terracotta servers with persistence of shared data, and you want the cluster to load the original (on-disk) `ehcache.xml`, the servers' database must be wiped by removing the data files from the servers' `server-data` directory. This directory is specified in the Terracotta configuration file in effect (`tc-config.xml` by default). Wiping the database causes *all persisted shared data to be lost*.

2.3.1a Setting Cache Eviction

Cache eviction removes elements from the cache based on parameters with configurable values. Having an optimal eviction configuration is critical to maintaining cache performance.

To add eviction and control the size of the cache, edit the values of the following `<cache>` attributes and

tune these values based on results of performance tests:

- `timeToldleSeconds` - The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- `timeToLiveSeconds` - The maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from the cache. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
- `maxElementsInMemory` - The maximum number of elements allowed in a cache in any one Terracotta client (also called application server or node). If this target is exceeded, eviction occurs to bring the count within the allowed target. The default value is 0, which means no eviction takes place (infinite size is allowed).
- `maxElementsOnDisk` - The maximum sum total number of elements allowed for a cache in all Terracotta clients. If this target is exceeded, eviction occurs to bring the count within the allowed target. The default value is 0, which means no eviction takes place (infinite size is allowed). Note that this value reflects storage allocated on the Terracotta Server Array.
- `eternal` - If the cache's `eternal` flag is set, it overrides any finite TTI/TTL values that have been set.

Ensure that the edited `ehcache.xml` is in your application's classpath. If you are using a WAR file, `ehcache.xml` should be in `WEB-INF/classes`.

See [2.3.4 How Configuration Affects Element Eviction](#) for more information on how configuration can impact eviction. See [2.3.1d Terracotta Clustering Configuration Elements](#) for definitions of other available configuration properties.

2.3.1b Cache-Configuration File Properties

See [Terracotta Clustering Configuration Elements](#) for more information.

2.3.1c Exporting Configuration from the Developer Console

To create or edit a cache configuration in a live cluster, see [8.1.3d Editing Cache Configuration](#).

To persist custom cache configuration values, create a cache configuration file by exporting customized configuration from the Terracotta Developer Console or create a file that conforms to the required format. This file must take the place of any configuration file used when the cluster was last started.

2.3.1d Terracotta Clustering Configuration Elements

Certain elements in the `Ehcache` configuration control the clustering of caches with Terracotta.

<terracotta>

This element is an optional sub-element of `<cache>`. It can be set differently for each `<cache>` defined in `ehcache.xml`.

`<terracotta>` has one subelement, `<nonstop>` (see [2.3.3 Non-Blocking Disconnected \(Nonstop\) Cache](#) for more information).

The following `<terracotta>` attributes allow you to control the type of data consistency for the distributed cache:

- `consistency` - Enables strong consistency ("strong" DEFAULT) or eventual consistency ("eventual"). When strong, guarantees the consistency of the cache's data across the cluster at all times at the cost of performance. After any update is completed, no read can return a stale value. When eventual, improves performance while guaranteeing eventual cluster-wide cache consistency. Once set, this consistency mode cannot be changed except by reconfiguring the cache using a configuration file and reloading the file. *This setting cannot be changed programmatically.*

Except for special cases, the following `<terracotta>` attributes should operate with their default values:

- `clustered` - Enables ("true" DEFAULT) or disables ("false") Terracotta clustering of a specific cache. Clustering is enabled if this attribute is not specified. Disabling clustering also disables the effects of all of the other attributes.
- `synchronousWrites` - Enables ("true") or disables ("false" DEFAULT) synchronous writes from Terracotta clients (application servers) to Terracotta servers. Asynchronous writes (`synchronousWrites="false"`) maximize performance by allowing clients to proceed without waiting for a "transaction received" acknowledgement from the server. This acknowledgement is unnecessary in most use cases. Synchronous writes (`synchronousWrites="true"`) provide extreme data safety at a very high performance cost by requiring that a client receive server acknowledgement of a transaction before that client can proceed. *Enabling synchronous writes has a significant detrimental effect on cluster performance.* If the cache's consistency mode is eventual (`consistency="eventual"`), or while it is set to bulk load using

the bulk-load API, only asynchronous writes can occur (synchronousWrites="true" is ignored).

- **storageStrategy** - Sets the strategy for storing the cache's key set. Use "DCV2" (DEFAULT) to store the cache's key set on the Terracotta server array. DCV2 can be used only with serializable caches (the valueMode attribute must be set to "serialization"), whether using the standard installation or DSO. DCV2 takes advantage of performance optimization built into the Terracotta libraries. Use "classic" to store all keys on every Terracotta client, but note that the performance optimization techniques built into the Terracotta libraries *will not be in effect*. Identity caches (valueMode="identity") must use the classic mode. For more information on using storageStrategy, see [Offloading Large Caches](#).
- **concurrency** - Sets the number of segments for the map backing the underlying server store. The default (when this attribute is not set or set to 0) is 2048 segments with the DCV2 storageStrategy and 128 segments with the classic storageStrategy. See [2.3.2a Tuning Concurrency](#) for more information on how to tune this value for DCV2.
- **valueMode** - Sets the type of cache to `serialization` (DEFAULT, the standard Ehcache "copy" cache) or `identity` (Terracotta object identity). **Identity mode is not available with the standard (express) installation. Identity mode can be used only with a Terracotta DSO (custom) installation** (see [Standard Versus DSO Installations](#)).

TIP: Comparing Serialization and Identity Modes

In serialization mode, getting an element from the cache gets a copy of that element. Changes made to that copy do not affect any other copies of the same element or the value in the cache. Putting the element in the cache overwrites the existing value. This type of cache provides high performance with small, read-only data sets. Large data sets with high traffic, or caches with very large elements, can suffer performance degradation because this type of cache serializes clustered objects. This type of cache cannot guarantee a consistent view of object values in read-write data sets if the *consistency* attribute is disabled. Objects clustered in this mode *must be* serializable. Note that `getKeys()` methods return serialized versions of the keys.

In identity mode, getting an element from the cache gets a reference to that element. Changes made to the referenced element updates the element on every node on which it exists (or a reference to it exists) as well as updating the value in the cache. Putting the element in the cache does not overwrite the existing value. This mode guarantees data consistency. It can be used only with a custom Terracotta Distributed Cache installation. Objects clustered in this mode must be [portable](#) and must be locked when accessed. If you require identity mode, you must use DSO (see [Terracotta DSO Installation](#)).

- **copyOnRead** - DEPRECATED. Use the copyOnRead <cache> attribute. Enables ("true") or disables ("false" DEFAULT) "copy cache" mode. If disabled, cache values are not deserialized on every read. For example, repeated `get()` calls return a reference to the same object (references are ==). When enabled, cache values are deserialized (copied) on every read and the materialized values are *not* re-used between `get()` calls; each `get()` call returns a unique reference. When enabled, allows Ehcache to behave as a component of OSGI, allows a cache to be shared by callers with different classloaders, and prevents local drift if keys/values are mutated locally without being put back into the cache. **Enabling copyOnRead is relevant only for caches with valueMode set to serialization.**
- **coherentReads** - DEPRECATED. This attribute is superseded by the attribute *consistency*. Disallows ("true" DEFAULT) or allows ("false") "dirty" reads in the cluster. If set to "true", reads must be consistent on any node and returned data is guaranteed to be consistent. If set to false, local unlocked reads are allowed and returned data may be stale. Allowing dirty reads may boost the cluster's performance by reducing the overhead associated with locking. Read-only applications, applications where stale data is acceptable, and certain read-mostly applications may be suited to allowing dirty reads.

The following attributes are used with [Enterprise Ehcache for Hibernate](#):

- **localKeyCache** - Enables ("true") or disables ("false" DEFAULT) a local key cache. [Enterprise Ehcache for Hibernate](#) can cache a "hotset" of keys on clients to add locality-of-reference, a feature suitable for read-only cases. Note that the set of keys must be small enough for available memory.
- **localKeyCacheSize** - Defines the size of the local key cache in number (positive integer) of elements. In effect if localKeyCache is enabled. The default value, 300000, should be tuned to meet application requirements and environmental limitations.
- **orphanEviction** - Enables ("true" DEFAULT) or disables ("false") orphan eviction. *Orphans* are cache elements that are not resident in any Hibernate second-level cache but still present on the cluster's Terracotta server instances.
- **orphanEvictionPeriod** - The number of local eviction cycles (that occur on Hibernate) that must be completed before an orphan eviction can take place. The default number of cycles is 4. Raise this value for less aggressive orphan eviction that can reduce faulting on the Terracotta server, or raise it if garbage on the Terracotta server is a concern.

Default Behavior

By default, adding `<terracotta />` to a `<cache>` block is equivalent to adding the following:

```
<terracotta clustered="true" valueMode="serialization" consistency="strong" storageStrategy="DCV2" />
```

<terracottaConfig>

This element can *not* be used with a DSO installation (see [Standard Versus DSO Installations](#)). It enables the client to identify a source of Terracotta configuration. It also allows a client to rejoin a cluster after disconnecting from that cluster and being timed out by a Terracotta server. For more information on the rejoin feature, see [2.3.7a Using Rejoin to Automatically Reconnect Terracotta Clients](#).

The client must load the configuration from a file or a Terracotta server. The value of the `url` attribute should contain a path to the file or the address and DSO port (9510 by default) of a server. In the example value, "localhost:9510" means that the Terracotta server is on the local host.

TIP: Terracotta Clients and Servers

In a Terracotta cluster, the application server is also known as the client.

For more information on client configuration, see the *Clients Configuration Section* in the [Terracotta Configuration Guide and Reference](#).

Using an URL Attribute

Add the `url` attribute to the `<terracottaConfig>` element as follows:

```
<terracottaConfig url="<source>" />
```

where `<source>` must be one of the following:

- A path (for example, `url="/path/to/tc-config.xml"`)
- An URL (for example, `url="http://www.mydomain.com/path/to/tc-config.xml"`)
- A Terracotta host address in the form `<host>:<dso-port>` (for example, `url="host1:9510"`)

Note the following about using server addresses in the form `<host>:<dso-port>`:

- The default DSO port is 9510.
- In a multi-server cluster, you can specify a comma-delimited list (for example, `url="host1:9510,host2:9510,host3:9510"`).
- If the Terracotta configuration source changes at a later time, it must be updated in configuration.

Embedding Terracotta Configuration

You can embed the contents of a Terracotta configuration file in `ehcache.xml` as follows:

```
<terracottaConfig>
  <tc-config>
    <servers>
      <server host="server1" name="s1"/>
      <server host="server2" name="s2"/>
    </servers>
    <clients>
      <logs>app/logs-%i</logs>
    </clients>
  </tc-config>
</terracottaConfig>
```

Note that not all elements are supported. For example, the `<dso>` section of a Terracotta configuration file is ignored in an Ehcache configuration file.

2.3.1e Controlling Cache Size

Certain Ehcache cache configuration attributes affect caches clustered with Terracotta.

The following example shows a cache configuration with a number of attributes aimed at controlling the size of the cache:

```
<cache name="myCache" maxElementsInMemory="1000"
  maxElementsOnDisk="10000" eternal="false" timeToIdleSeconds="3600"
  timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
  <!-- Adding the element <terracotta /> turns on Terracotta clustering for the cache myCache. -->
  <terracotta />
</cache>
```

Note the following about the `myCache` configuration:

- Terracotta clients that load `myCache` will keep up to 1000 elements in heap (

`maxElementsInMemory`).

- If a client accesses an element in `myCache` that has been idle for more than an hour (`timeToIdleSeconds`), it evicts that element. The element is also evicted from the Terracotta Server Array.
- Elements in `myCache` can live forever if accessed at least once per 60 minutes (`timeToLiveSeconds`). However, unexpired elements may still be flushed based on memory limitations (`maxElementsInMemory`).
- Cluster-wide, `myCache` can store a maximum of 10000 elements (`maxElementsOnDisk`). This is the effective maximum number elements `myCache` is allowed on the Terracotta Server Array.

See [2.3.4 How Configuration Affects Element Eviction](#) for more information on how configuration affects eviction.

2.3.1f Cache Events Configuration

The `<cache>` subelement `<cacheEventListenerFactory>`, which registers listeners for cache events such as puts and updates, has a notification scope controlled by the attribute `listenFor` . This attribute can have one of the following values:

- `local` - Listen for events on the local node. No remote events are detected.
- `remote` - Listen for events on other nodes. No local events are detected.
- `all` - (DEFAULT) Listen for events on both the local node and on remote nodes.

In order for cache events to be detected by remote nodes in a Terracotta cluster, event listeners must have scope that includes remote events. For example, the following configuration allows listeners of type `MyCacheListener` to detect both local and remote events:

```
<cache name="myCache" maxElementsInMemory="1000"
      maxElementsOnDisk="10000" eternal="false" timeToIdleSeconds="3600"
      timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
  <!-- Not defining the listenFor attribute for <cacheEventListenerFactory> is by default equivalent to
  listenFor="all". -->
  <cacheEventListenerFactory class="net.sf.ehcache.event.TerracottaCacheEventReplicationFactory" />
</cache>
```

You must use `net.sf.ehcache.event.TerracottaCacheEventReplicationFactory` as the factory class to enable cluster-wide cache-event broadcasts in a Terracotta cluster.

See [2.3.6 Cache Events in a Terracotta Cluster](#) for more information on cache events in a Terracotta cluster.

2.3.1g Incompatible Configuration

For any clustered cache, you must delete, disable, or edit configuration elements in `ehcache.xml` that are incompatible when clustering with Terracotta. Clustered caches have a `<terracotta />` or `<terracotta clustered="true">` element.

The following Ehcache configuration attributes or elements should be deleted or disabled:

- DiskStore-related attributes `overflowToDisk` and `diskPersistent` .
The Terracotta server automatically provides a disk store.
- Replication-related attributes such as `replicateAsynchronously` and `replicatePuts` .
- The attribute `MemoryStoreEvictionPolicy` must be set to either `LFU` or `LRU`.
Setting `MemoryStoreEvictionPolicy` to `FIFO` causes the error `IllegalArgumentException` .

See the [Ehcache documentation](#) for more information on the elements in a standard Ehcache configuration file.

2.3.2 Offloading Large Caches

Storing a distributed cache's entire key set on each Terracotta client provides high locality-of-reference, reducing latency at the cost of using more client memory. It also allows for certain cache-management optimizations on each client that improve the overall performance of the cache. This works well for smaller key sets which can easily fit into the JVM.

However, for caches with elements numbering in the millions or greater, performance begins to deteriorate when every client must store the entire key set. Clusters with a large number of clients require even more overhead to manage those key sets. If the cache is also heavy on writes, that overhead can cause a considerable performance bottleneck.

In addition to making it more difficult to scale a cluster, larger caches can cause other serious performance

issues:

- Cache-loading slowdown - The cache's entire key set must be fully present in the client before the cache is available.
- Reduction in free client memory - Less available memory may cause more flushing and faulting.
- More garbage collection - Larger heaps (to accommodate larger key sets) and more objects in memory means more garbage created and more Java garbage collection cycles.

The DCV2 mode of managing Terracotta clustered caches avoids these issues by offloading cache entries to the Terracotta server array, allowing clients to fault in only required keys. Some of the advantages of the DCV2, which is used by default, include server-side eviction, automatic and flexible hot-set caching by clients, and cluster-wide consistency without cluster-wide delta broadcasts.

Note the following about the DCV2 mode:

- Under certain circumstances, unexpired elements evicted from Terracotta clients to meet the limit set by `maxElementsInMemory` or to free up memory may also be evicted from the Terracotta server array. The client cannot fault such elements back from the server.
See [2.3.4 How Configuration Affects Element Eviction](#) for more information on how DCV2, element expiration, and element eviction are related.
- `UnlockReadsView` and bulk-load mode is not optimized for DCV2 with strong consistency. Elements populated through bulk load expire according to a set timeout and may persist in the cache even after being evicted by the server array (see [2.3.4 How Configuration Affects Element Eviction](#) for more information). You can bypass this issue by using "eventual" consistency mode (see [2.3.5 Understanding Performance and Cache Consistency](#) for more information).
- The entire cache's key set must fit into the server array's aggregate heap. The server array's aggregate heap is equal to the sum of each active server's heap size. `BigMemory` allows you to bypass this restriction. See [6.4 Improving Server Performance With BigMemory](#) for more information.

Very large key sets can be offloaded effectively to a scaled-up Terracotta server array with a sufficient number of mirror groups. See [Scaling the Terracotta Server Array](#) for more information on mirror groups.

To configure a cache to *not* offload its key set, set the attribute `storageStrategy="classic"` in that cache's `<terracotta>` element.

2.3.2a Tuning Concurrency

The server map underlying the Terracotta Server Array contains the data used by clients in the cluster and is segmented to improve performance through added concurrency. The concurrency attribute in the `<terracotta>` element controls this segmentation.

With large data sets, a high concurrency value can improve performance by hashing the data into the segments, which reduces lock contention.

However, the default concurrency value may not be efficient for certain environments, such as those with very few cache elements or a low `maxElementsOnDisk` value. In this case, the default concurrency value of 2048 creates 2048 segments on the Terracotta Server Array, with each segment holding a few (or even one) element. `maxElementsOnDisk` may appear to have been exceeded, and the cluster may run low on memory as it loads all segments into RAM, even if they are nearly empty.

To set concurrency to an optimum value, follow these guidelines:

- If `maxElementsOnDisk` is not set, set to 0, or set to a value equal to or greater than 256, set concurrency equal to 256.
- If `maxElementsOnDisk` is set to a value less than 256, set concurrency to the highest power of 2 that is less than or equal to the value of `maxElementsOnDisk`.
For example, if `maxElementsOnDisk` is 130, set concurrency to 128.

To learn how to set concurrency for a cache, see the section on the [<terracotta>](#) element.

2.3.3 Non-Blocking Disconnected (Nonstop) Cache

A nonstop cache allows certain cache operations to proceed on clients that have become disconnected from the cluster. Clients go into nonstop mode if they receive a "cluster offline" event or if a cache operation cannot complete by the nonstop timeout value.

2.3.3a Configuring Nonstop

Nonstop is configured in a `<cache>` block under the `<terracotta>` subelement. In the following example, `myCache` has nonstop configuration:

```
<cache name="myCache" maxElementsInMemory="10000" eternal="false"
  overflowToDisk="false">
  <terracotta>
    <nonstop immediateTimeout="false" timeoutMillis="30000">
      <timeoutBehavior type="noop" />
    </nonstop>
  </terracotta>
</cache>
```

Nonstop is enabled by default or if `<nonstop>` appears in a cache's `<terracotta>` block.

2.3.3b Nonstop Timeouts and Behaviors

Nonstop caches can be configured with the following attributes:

- `enabled` - Enables ("true" DEFAULT) or disables ("false") the ability of a cache to execute certain action after a Terracotta client disconnects. This attribute is optional for enabling nonstop.
- `immediateTimeout` - Enables ("true") or disables ("false" DEFAULT) an immediate timeout response if the Terracotta client detects a network interruption (the node is disconnected from the cluster). If enabled, this parameter overrides `timeoutMillis`, so that the option set in `timeoutBehavior` is in effect immediately.
- `timeoutMillis` - Specifies the number of milliseconds an application waits for any cache operation to return before timing out. The default value is 30000 (thirty seconds). The behavior after the timeout occurs is determined by `timeoutBehavior`.

`<nonstop>` has one self-closing subelement, `<timeoutBehavior>`. This subelement determines the response after a timeout occurs (`timeoutMillis` expires or an immediate timeout occurs). The response can be set by the `<timeoutBehavior>` attribute `type`. This attribute can have one of the values listed in the following table:

Value	Behavior
<code>exception</code>	(DEFAULT) Throw <code>NonStopCacheException</code> . See When is NonStopCacheException Thrown? for more information on this exception.
<code>noop</code>	Return null for gets. Ignore all other cache operations. Hibernate users may want to use this option to allow their application to continue with an alternative data source.
<code>localReads</code>	For caches with Terracotta clustering, allow inconsistent reads of cache data. Ignore all other cache operations. For caches without Terracotta clustering, throw an exception.

Tuning Nonstop Timeouts and Behaviors

You can tune the default timeout values and behaviors of nonstop caches to fit your environment. For example, in an environment with regular network interruptions, consider disabling `immediateTimeout` and increasing `timeoutMillis` to prevent timeouts for most of the interruptions.

For a cluster that experiences regular but short network interruptions, and in which caches clustered with Terracotta carry read-mostly data or there is tolerance of potentially stale data, you may want to set `timeoutBehavior` to `localReads`.

In an environment where cache operations can be slow to return and data is required to always be in sync, increase `timeoutMillis` to prevent frequent timeouts. Set `timeoutBehavior` to `noop` to force the application to get data from another source or `exception` if the application should stop.

If a nonstop cache is bulk-loaded using the [Bulk-Load API](#), a multiplier is applied to the configured nonstop timeout whenever the method

`net.sf.ehcache.Ehcache.setNodeBulkLoadEnabled(boolean)` is used. The default value of the multiplier is 10. You can tune the multiplier using the `bulkOpsTimeoutMultiplyFactor` system property:

```
-DbulkOpsTimeoutMultiplyFactor=10
```

This multiplier also affects the methods `net.sf.ehcache.Ehcache.removeAll()`, `net.sf.ehcache.Ehcache.removeAll(boolean)`, and

```
net.sf.ehcache.Ehcache.setNodeCoherent(boolean) (DEPRECATED).
```

When is NonStopCacheException Thrown?

NonStopCacheException is usually thrown when it is the configured behavior for a nonstop cache in a client that disconnects from the cluster. In the following example, the exception would be thrown 30 seconds after the disconnection (or the "cluster offline" event is received):

```
<nonstop immediateTimeout="false" timeoutMillis="30000">
  <timeoutBehavior type="exception" />
</nonstop>
```

However, under certain circumstances the NonStopCache exception can be thrown even if a nonstop cache's timeout behavior is *not* set to throw the exception. This can happen when the cache goes into nonstop mode during an attempt to acquire or release a lock. These lock operations are associated with certain lock APIs and special cache types such as [Explicit Locking](#), BlockingCache, SelfPopulatingCache, and UpdatingSelfPopulatingCache.

A NonStopCacheException can also be thrown if the cache must fault in an element to satisfy a `get()` operation. If the Terracotta Server Array cannot respond within the configured nonstop timeout, the exception is thrown.

A related exception, InvalidLockAfterRejoinException, can be thrown during or after client rejoin (see [2.3.7 Using Rejoin to Automatically Reconnect Terracotta Clients](#)). This exception occurs when an unlock operation takes place on a lock obtained *before* the rejoin attempt completed.

TIP: Use try-finally Blocks

To ensure that locks are released properly, application code using Ehcache lock APIs should encapsulate lock-unlock operations with try-finally blocks:

```
myLock.acquireLock();
try {
    // Do some work.
} finally {
    myLock.unlock();
}
```

2.3.4 How Configuration Affects Element Eviction

Element eviction is a crucial part of keeping cluster resources operating efficiently. Element eviction and expiration are related, but an expired element is not necessarily evicted immediately and an evicted element is not necessarily an expired element. Cache elements may be evicted due to resource and configuration constraints, while expired elements are evicted from the Terracotta client when a *get* or *put* operation occurs on that element (sometimes called *inline* eviction).

If the cache's `storageStrategy` is set to "classic", then clients contain all of a cache's keys; once an element is expired, it can be evicted from any client and the Terracotta server array. In this case, eviction is a function of constraints on the client.

By default, the cache's `storageStrategy` is set to "DCV2". In this case, the Terracotta server array contains the full key set (as well as all values), while clients contain a subset of keys and values based on elements they've faulted in from the server array. The rest of this discussion assumes that the `storageStrategy` is set to DCV2.

TIP: Eviction Under DCV2 With UnlockedReadView and Bulk Loading

Under certain circumstances, DCV2 caches may evict elements based on a configured timeout. See [2.3.4a DCV2, Strict Consistency, UnlockedReadView, and Bulk Loading](#) for more information.

Typically, an expired cache element is evicted from a client when a `get()` or `put()` operation occurs on that element. However, a client may also evict expired, and then unexpired elements, whenever a cache's `maxElementsInMemory` is reached or it is under memory pressure. This type of eviction is intended to meet configured and real memory constraints.

Eviction from clients does not mean eviction from the server array. Elements can become candidates for eviction from the server array when disks run low on space. Servers with a disk-store limitation set by `maxElementsOnDisk` can come under disk-space pressure and will evict expired elements first. However, unexpired elements can also be evicted if they meet the following criteria:

- They are in a cache with infinite TTI/TTL (Time To Idle and Time To Live), or no explicit settings for TTI/TTL.
Enabling a cache's `eternal` flag overrides any finite TTI/TTL values that have been set.

- They are not resident on any Terracotta client. These elements can be said to have been "orphaned". Once evicted, they will have to be faulted back in from a system of record if requested by a client.
- Their per-element TTI/TTL settings indicate that they've expired and the server array is inspecting per-element TTI/TTL. Note that per-element TTI/TTL settings are, by default, *not* inspected by Terracotta servers.

TIP: Forcing Terracotta Servers to Inspect Per-Element TTI/TTL

To help maintain a high level of performance, per-element TTI/TTL settings are not inspected by Terracotta servers. To force servers to inspect and honor per-element TTI/TTL settings, enable the Terracotta property `ehcache.storageStrategy.dcv2.perElementTTITTL.enabled` by adding the following configuration to the top of the Terracotta configuration file (`tc-config.xml` by default) before starting the Terracotta server:

```
<tc-properties>
  <property name="ehcache.storageStrategy.dcv2.perElementTTITTL.enabled" value="true" />
</tc-properties>
```

While this setting may prevent unexpired elements (based on per-element TTI/TTL) from being evicted, it also degrades performance by incurring processing costs.

A server array will not evict unexpired elements if servers are configured to have infinite store (`maxElementsOnDisk` is not set or is set to 0). Under these conditions, **the expected data set must fit in the server array or the cluster may suffer from performance degradation and errors.**

2.3.4a DCV2, Strict Consistency, UnlockedReadsView, and Bulk Loading

When a cache that uses the DCV2 storage strategy and strict consistency is decorated with `UnlockedReadsView` (see [2.2.4 Unlocked Reads for Consistent Caches \(UnlockedReadsView\)](#)), unlocked reads may cause elements to be faulted in. These elements expire based on a cluster-wide timeout controlled by the Terracotta property

`ehcache.storageStrategy.dcv2.localcache.incoherentReadTimeout`. This timeout, which by default is set to five minutes, can be tuned in the Terracotta configuration file (`tc-config.xml`):

```
<tc-properties>
<!-- The following timeout is set in milliseconds. -->
  <property name="ehcache.storageStrategy.dcv2.localcache.incoherentReadTimeout" value="300000" />
</tc-properties>
```

If the same elements are changed on a remote node, the local elements under the effect of this timeout will *not* expire or become invalid until the timeout is reached.

This timeout also applies to elements that are put into the cache using the bulk-load API (see [2.2.3 Bulk-Load API](#)).

2.3.5 Understanding Performance and Cache Consistency

Cache consistency modes are configuration settings and API methods that control the behavior of clustered caches with respect to balancing data consistency and application performance. A cache can be in one of the following consistency modes:

- Strong - This mode ensures that data in the cache remains consistent across the cluster at all times. It can guarantee that a read gets an updated value only after all write operations to that value are completed. The use of locking and transaction acknowledgments maximizes consistency at the cost of performance. This mode is set using the Ehcache configuration file and cannot be changed programmatically (see the attribute "consistent" in [<terracotta>](#)).
- Eventual - This mode guarantees that data in the cache will eventually be consistent across the cluster. Read/write performance is boosted at the cost of potentially having an inconsistent cache for short periods of time. This mode is set using the Ehcache configuration file and cannot be changed programmatically (see the attribute "consistency" in [<terracotta>](#)).
- Bulk Load - This mode is optimized for bulk-loading data into the cache without the slowness introduced by locks or regular eviction. It is similar to the eventual mode, but has batching, higher write speeds, and weaker consistency guarantees. This mode is set using the bulk-load API only (see [2.2.3 Bulk-Load API](#)). When turned off, allows the configured consistency mode (either strong or eventual) to take effect again.

Use configuration to set the permanent consistency mode for a cache as required for your application, and the bulk-load mode only during the time when populating (warming) or refreshing the cache.

The following APIs and settings also affect consistency:

- **Explicit Locking** - This API provides methods for locking specific elements in a cache. There is guaranteed consistency across the cluster at all times for operations on elements covered by a lock. While explicit locking of elements provides fine-grained locking, there is still the potential for contention, blocked threads, and increased performance overhead from managing clustered locks. See [2.2.5 Explicit Locking](#) for more information.
- **UnlockedReadsView** - A cache decorator that allows dirty reads of the cache. This decorator can be used only with caches in the strong consistency mode. UnlockedReadsView raises performance for this mode by bypassing the requirement for a read lock. See [2.2.4 Unlocked Reads for Consistent Caches \(UnlockedReadsView\)](#) for more information.
- **Atomic methods** - You can guarantee write consistency at all times, avoiding potential race conditions for put operations, by using the following atomic methods `Cache.putIfAbsent(Element element)` and `Cache.replace(Element oldOne, Element newOne)`. However, there is no guarantee that these methods' return value is not stale because another operation may change the element after the atomic method completes but before the return value is read. To guarantee the return value, use locks (see [2.2.5 Explicit Locking](#)). Note that using locks may impact performance.

2.3.6 Cache Events in a Terracotta Cluster

Cache events are fired for certain cache operations:

- **Evictions** - An eviction on a client generates an eviction event on that client. An eviction on a Terracotta server fires an event on a random client.
- **Puts** - A `put()` on a client generates a put event on that client.
- **Updates** - If a cache uses default storage strategy (`<terracotta ... storageStrategy="DCV2" ... >`), then an update on a client generates a put event on that client.
- **orphan eviction** - An orphan is an element that exists only on the Terracotta Server Array. If an orphan is evicted, an eviction event is fired on a random client.

See [2.3.1f Cache Events Configuration](#) for more information on configuring the scope of cache events.

2.3.6a Handling Cache Update Events With DCV2

Caches that use the DCV2 storage strategy will generate put events whenever elements are put or updated. If it is important for your application to distinguish between puts and updates, check for the existence of the element during `put()` operations:

```
if (cache.containsKey(key)) {
    cache.put(element);
    // Action in the event handler on replace.
} else {
    cache.put(element);
    // Action in the event handler on new puts.
}
```

To protect against races, wrap the if block with explicit locks (see [2.2.5 Explicit Locking](#)). You can also use the atomic cache methods `putIfAbsent()` or to check for the existence of an element:

```
if((olde = cache.putIfAbsent(element)) == null) { // Returns null if successful or returns the existing
    // Action in the event handler on new puts.
} else {
    cache.replace(olde, newElement); // Returns true if successful.
    // Action in the event handler on replace.
}
```

If your code cannot use these approaches (or a similar workaround), you can force update events for cache updates by setting the Terracotta property `ehcache.clusteredStore.checkContainsKeyOnPut` at the top of the Terracotta configuration file (`tc-config.xml` by default) before starting the Terracotta Server Array:

```
<tc-properties>
  <property name="ehcache.clusteredStore.checkContainsKeyOnPut" value="true" />
</tc-properties>
```

Enabling this property can substantially degrade performance.

2.3.7 Configuring Caches for High Availability

Enterprise Ehcache caches provide the following High Availability (HA) settings:

- Non-blocking cache - Also called nonstop cache. When enabled, this attribute gives the cache the ability to take a configurable action after the Terracotta client receives a cluster-offline event. See [2.3.3 Non-Blocking Disconnected \(Nonstop\) Cache](#) for more information.
- Rejoin - The rejoin attribute allows a Terracotta client to reconnect to the cluster after it receives a cluster-online event. See [2.3.7a Using Rejoin to Automatically Reconnect Terracotta Clients](#) for more information.

To learn about configuring HA in a Terracotta cluster, see [6.2 Configuring Terracotta Clusters For High Availability](#).

2.3.7a Using Rejoin to Automatically Reconnect Terracotta Clients

A Terracotta client running Enterprise Ehcache may disconnect and be timed out (ejected) from the cluster. Typically, this occurs because of network communication interruptions lasting longer than the configured HA settings for the cluster. Other causes include long GC pauses and slowdowns introduced by other processes running on the client hardware.

You can configure clients to automatically rejoin a cluster after they are ejected. If the ejected client continues to run under nonstop cache settings, and then senses that it has reconnected to the cluster (receives a clusterOnline event), it can begin the rejoin process.

Note the following about using the rejoin feature:

- Rejoin is for CacheManagers with only nonstop caches. If one or more of a CacheManager's caches is not set to be nonstop, and rejoin is enabled, an exception is thrown at initialization. An exception is also thrown in this case if a cache is created programmatically without nonstop.
- Clients rejoin as new members and will wipe all cached data to ensure that no pauses or inconsistencies are introduced into the cluster.
- Any nonstop-related operations that begin (and do not complete) before the rejoin operation completes may be unsuccessful and may generate a NonStopCacheException.
- Enterprise Ehcache with rejoin enabled should not share a Terracotta client or a JVM with products that do not support rejoin. For example, if Enterprise Ehcache is running with Terracotta Quartz Scheduler in a single Terracotta client, and a rejoin occurs, a second client running only Ehcache will be spawned to share the JVM with the original client. To ensure that rejoin succeeds, run Ehcache in its own JVM.
- Once a client rejoins, the clusterRejoined event is fired on that client only.

Configuring Rejoin

The rejoin feature is disabled by default. To enable the rejoin feature in an Enterprise Ehcache client, follow these steps:

1. Ensure that all of the caches in the Ehcache configuration file where rejoin is enabled have nonstop enabled.
2. Ensure that your application does not create caches on the client without nonstop enabled.
3. Enable the rejoin attribute in the client's <terracottaConfig> element:

```
<terracottaConfig url="myHost:9510" rejoin="true" />
```

Avoiding OOME From Multiple Rejoins

Each time a client rejoins a cluster, it reloads all class definitions into the heap's Permanent Generation (PermGen) space. If a number of rejoins happen before Java garbage collection (GC) is able to free up enough PermGen, an OutOfMemory error (OOME) can occur. PermGen holds other data, as well, and allocating too large of a PermGen space can make an OOME more likely under these conditions.

The default value of PermGen on Oracle Sun JVM is 64MB. You can tune this value using the Java options -XX:PermSize (starting value) and -XX:MaxPermSize (maximum allowed value). For example:

```
-XX:PermSize=<value>m -XX:MaxPermSize=<value>m
```

If your cluster experiences regular node disconnections that trigger many rejoins, and OOMs are occurring, investigate your application's usage of the PermGen space and how well GC is keeping up with reclaiming that space. Then test lower and higher values for PermGen with the aim of eliminating the OOMs.

Exception During Rejoin

Under certain circumstances, if one of the Ehcache locking APIs is being used by your application, an InvalidLockAfterRejoinException could be thrown. See [When is NonStopCacheException Thrown?](#) for more information.

2.3.8 Working With Transactional Caches

Transactional caches add a level of safety to cached data and ensure that the cached data and external data stores are in sync. Enterprise Ehcache caches can participate in Java Transaction API (JTA) transactions as a fully compliant XA resource. This is useful in JTA applications requiring caching, or where cached data is critical and must be persisted and remain consistent with System of Record data.

However, transactional caches are slower than non-transactional caches due to the overhead from having to write transactionally. Transactional caches also have the following restrictions:

- Data can be accessed only transactionally, even for read-only purposes.
You must encapsulate data access with `begin()` and `commit()` statements.
- `copyOnRead` and `copyOnWrite` must be enabled.
These `<cache>` attributes are "false" by default and must set to "true".
- Caches must be strongly consistent.
A transactional cache's `consistency` attribute must be set to "strong".
- Nonstop caches cannot be made transactional.
Transactional caches must *not* contain the `<nonstop>` subelement.
- Decorating a transactional cache with `UnlockedReadsView` can return inconsistent results for data obtained through `UnlockedReadsView`.
Puts, and gets not through `UnlockedReadsView`, are not affected.
- Objects stored in a transactional cache must override `equals()` and `hashCode()`.
If overriding `equals()` and `hashCode()` is not possible, see [2.3.8e Implementing an Element Comparator](#).

You can choose one of three different modes for transactional caches:

- **Strict XA** - Has full JTA support for XA transactions. May not be compatible with transaction managers that do not fully support JTA.
- **XA** - Has support for the most common JTA components, so likely to be compatible with most transaction managers. But unlike strict XA, may fall out of sync with a database after a failure (has no recovery). Integrity of cache data, however, is preserved.
- **Local** - Local transactions written to a local store and likely to be faster than the other transaction modes. This mode does not require a transaction manager and does not synchronize with remote data sources. Integrity of cache data is preserved in case of failure.

NOTE: Deadlocks

Both the XA and local mode write to the underlying store synchronously and using pessimistic locking. Under certain circumstances, this can result in a deadlock, which generates a `DeadLockException` after a transaction times out and a commit fails. Your application should catch `DeadLockException` (or `TransactionException`) and call `rollback()`.

Deadlocks can have a severe impact on performance. A high number of deadlocks indicates a need to refactor application code to prevent races between concurrent threads attempting to update the same data.

These modes are explained in the following sections.

2.3.8a Strict XA (Full JTA Support)

Note that Ehcache as an XA resource:

- Has an isolation level of `ReadCommitted`.
- Updates the underlying store asynchronously, potentially creating update conflicts.
With this optimistic locking approach, Ehcache may force the transaction manager to roll back the entire transaction if a `commit()` generates a `RollbackException` (indicating a conflict).
- Can work alongside other resources such as JDBC or JMS resources.
- Guarantees that its data is always synchronized with other XA resources.
- Can be configured on a per-cache basis (transactional and non-transactional caches can exist in the same configuration).
- Automatically performs enlistment.
- Can be used standalone or integrated with frameworks such as Hibernate.
- Is tested with the most common transaction managers by Atomikos, Bitronix, JBoss, WebLogic, and others.

For more information on working with transactional caches in Enterprise Ehcache for Hibernate, see [Setting Up Transactional Caches](#).

Configuration

To configure Enterprise Ehcache as an XA resource able to participate in JTA transactions, the following

<cache> attributes must be set as shown:

- transactionalMode="xa_strict"
- copyOnRead="true"
- copyOnWrite="true"

In addition, the <cache> subelement <terracotta> must have the following attributes set as shown:

- valueMode="serialization"
- clustered="true"

For example, the following cache is configured for JTA transactions with strict XA:

```
<cache name="com.my.package.Foo"
  maxElementsInMemory="500"
  eternal="false"
  overflowToDisk="false"
  copyOnRead="true"
  copyOnWrite="true"
  consistency="strong"
  transactionalMode="xa_strict">
  <terracotta clustered="true" valueMode="serialization" />
</cache>
```

Any other XA resource that could be involved in the transaction, such as a database, must also be configured to be XA compliant.

Usage

Your application can directly use a transactional cache in transactions. This usage must occur after the transaction manager has been set to start a new transaction and before it has ended the transaction.

For example:

```
...
myTransactionMan.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
myTransactionMan.commit();
...
```

If more than one transaction writes to a cache, it is possible for an XA transaction to fail. See [2.3.8d Avoiding XA Commit Failures With Atomic Methods](#) for more information.

2.3.8b XA (Basic JTA Compliance)

Transactional caches set to "xa" provide support for basic JTA operations. Configuring and using XA does not differ from using local transactions (see [2.3.8c Local Transactions](#)), except that "xa" mode requires a transaction manager and allows the cache to participate in JTA transactions.

NOTE: Atomikos Transaction Manager

When using XA with an Atomikos transaction Manager, be sure to set `com.atomikos.icatch.threaded_2pc=false` in the Atomikos configuration. This helps prevent unintended rollbacks due to a bug in the way Atomikos behaves under certain conditions.

For example, the following cache is configured for JTA transactions with XA:

```
<cache name="com.my.package.Foo"
  maxElementsInMemory="500"
  eternal="false"
  overflowToDisk="false"
  copyOnRead="true"
  copyOnWrite="true"
  consistency="strong"
  transactionalMode="xa">
  <terracotta clustered="true" valueMode="serialization" />
</cache>
```

Any other XA resource that could be involved in the transaction, such as a database, must also be configured to be XA compliant.

2.3.8c Local Transactions

Local transactional caches (with the `transactionalMode` attribute set to "local") write to a local store using an API that is part of the Enterprise Ehcache core application. Local transactions have the following

characteristics:

- Recovery occurs at the time an element is accessed.
- Updates are written to the underlying store immediately.
- Get operations on the underlying store may block during commit operations.

To use local transactions, instantiate a `TransactionController` instance instead of a transaction manager instance:

```
TransactionController txCtrl = myCacheManager.getTransactionController();
...
txCtrl.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
txCtrl.commit();
...
```

You can use `rollback()` to roll back the transaction bound to the current thread.

TIP: Finding the Status of a Transaction on the Current Thread

You can find out if a transaction is in process on the current thread by calling `TransactionController.getCurrentTransactionContext()` and checking its return value. If the value isn't null, a transaction has started on the current thread.

Commit Failures and Timeouts

Commit operations can fail if the transaction times out. If the default timeout requires tuning, you can get and set its current value:

```
int currentDefaultTransactionTimeout = txCtrl.getDefaultTransactionTimeout();
...
txCtrl.setDefaultTransactionTimeout(30); // in seconds -- must be greater than zero.
```

You can also bypass the commit timeout using the following version of `commit()` :

```
txCtrl.commit(true); // "true" forces the commit to ignore the timeout.
```

2.3.8d Avoiding XA Commit Failures With Atomic Methods

If more than one transaction writes to a cache, it is possible for an XA transaction to fail. In the following example, if a second transaction writes to the same key ("1") and completes its commit first, the commit in the example may fail:

```
...
myTransactionMan.begin();
Cache fooCache = cacheManager.getCache("Foo");
fooCache.put("1", "Bar");
myTransactionMan.commit();
...
```

One approach to prevent this type of commit failure is to use one of the atomic put methods, such as `Cache.replace()` :

```
myTransactionMan.begin();
int val = cache.get(key).getValue(); // "cache" is configured to be transactional.
Element olde = new Element(key, val);
if (cache.replace(olde, new Element(key, val + 1)) { // True only if the element was successfully
    replaced.
    myTransactionMan.commit();
}
else { myTransactionMan.rollback(); }
```

Another useful atomic put method is `Cache.putIfAbsent(Element element)`, which returns null on success (no previous element exists with the new element's key) or returns the existing element (the put is not executed). Atomic methods cannot be used with null elements, or elements with null keys.

2.3.8e Implementing an Element Comparator

For all transactional caches, the atomic methods `Cache.removeElement(Element element)` and `Cache.replace(Element old, Element element)` must compare elements for the atomic operation to complete. This requires all objects stored in the cache to override `equals()` and `hashCode()`.

If overriding these methods is not desirable for your application, a default comparator is used (

`net.sf.ehcache.store.DefaultElementValueComparator`). You can also implement a custom comparator and specify it in the cache configuration with `<elementValueComparator>`:

```
<cache name="com.my.package.Foo"
      maxElementsInMemory="500"
      eternal="false"
      overflowToDisk="false"
      copyOnRead="true"
      copyOnWrite="true"
      consistency="strong"
      transactionalMode="xa">
  <elementValueComparator class="com.company.xyz.MyElementComparator" />
  <terracotta clustered="true" valueMode="serialization" />
</cache>
```

Custom comparators must implement `net.sf.ehcache.store.ElementValueComparator`.

A comparator can also be specified programmatically.

2.3.9 Working With OSGi

To allow Enterprise Ehcache to behave as an OSGi component, the following attributes should be set as shown:

```
<cache ... copyOnRead="true" ... >
...
  <terracotta ... clustered="true" valueMode="serialization" ... />
...
</cache>
```

3 Enterprise Ehcache for Hibernate

Enterprise Ehcache for Hibernate provides a flexible and powerful second-level cache solution for boosting the performance of Hibernate applications.

This document has the following sections:

- [Enterprise Ehcache for Hibernate Express Installation](#)
Begin with this simple installation procedure.
- [Testing and Tuning Enterprise Ehcache for Hibernate](#)
Use this document to locate trouble spots find ideas for improving performance.
- [Enterprise Ehcache for Hibernate Reference](#)
Covers configuration and other topics.

3.1 Enterprise Ehcache for Hibernate Express Installation

3.1.1 Step 1: Requirements

- JDK 1.5 or greater
- Hibernate 3.2.5, 3.2.6, 3.2.7, 3.3.1, or 3.3.2
Use the same version of Hibernate throughout the cluster. Sharing of Hibernate regions between different versions of Hibernate versions is not supported.
- Terracotta 3.5.0 package

3.1.2 Step 2: Install and Update the JAR files

For guaranteed compatibility, use the JAR files included with the Terracotta kit you are installing. Mixing with older components may cause errors or unexpected behavior.

To install the distributed cache in your application, add the following JAR files to your application's classpath:

- `${TERRACOTTA_HOME}/ehcache/lib/ehcache-terracotta-ee-<version>.jar`
<version> is the current version of the Ehcache-Terracotta JAR.
- `${TERRACOTTA_HOME}/ehcache/lib/ehcache-core-ee-<ehcache-version>.jar`
The Ehcache core libraries, where <ehcache-version> is the current version of Ehcache (2.4.1 or higher).
- `${TERRACOTTA_HOME}/ehcache/lib/slf4j-<slf4j-version>.jar`

The SLF4J logging facade allows Ehcache to bind to any supported logger used by your application. Binding JARs for popular logging options are available from the [SLF4J project](#). For convenience, the binding JAR for `java.util.logging` is provided in `${TERRACOTTA_HOME}/ehcache` (see below).

- `${TERRACOTTA_HOME}/ehcache/lib/slf4j-jdk14-<slf4j-version>.jar`
An SLF4J binding JAR for use with the standard `java.util.logging`, also known as JDK 1.4 logging.
- `${TERRACOTTA_HOME}/common/terracotta-toolkit-<API-version>-runtime-ee-<version>.jar`
The Terracotta Toolkit JAR contains the Terracotta client libraries. `<API-version>` refers to the Terracotta Toolkit API version. `<version>` is the current version of the Terracotta Toolkit JAR.

If you are using the open-source edition of the Terracotta kit, no JAR files will have "-ee-" as part of their name.

If you are using a WAR file, add these JAR files to the `WEB-INF/lib` directory.

NOTE: Application Servers

Most application servers (or web containers) should work with this installation of the Terracotta Distributed Cache. However, note the following:

- GlassFish - You must add the following to `domains.xml`:

```
<jvm-options>-Dcom.sun.enterprise.server.ss.ASQuickStartup=false</jvm-options>
```

- WebLogic - You must use the [supported version](#) of WebLogic. If using version 10.3, you must remove the `xml-apis` from `WEB-INF/lib` and add the following to `WEB-INF/weblogic.xml`:

```
<weblogic-web-app>
  <container-descriptor>
    <prefer-web-inf-classes>true</prefer-web-inf-classes>
  </container-descriptor>
</weblogic-web-app>
```

- JBoss 5.x - PermGen memory must be at least 128MB and can be set using the switch `-XX:MaxPermSize=128m`.

3.1.3 Step 3: Prepare Your Application for Caching

Hibernate entities that should be cached must be marked in one of the following ways:

- Using the `@Cache` annotation.
- Using the `<cache>` element of a class or collection mapping file (`hbm.xml` file).
- Using the `<class-cache>` (or `<collection-cache>`) element in the Hibernate XML configuration file (`hibernate.cfg.xml` by default).

For more information on configuring Hibernate, including configuring collections for caching, see the [Hibernate documentation](#).

In addition, you must specify a concurrency strategy for each cached entity. The following cache concurrency strategies are supported:

- `READ_ONLY`
- `READ_WRITE`
- `NONSTRICT_READ_WRITE`
- `TRANSACTIONAL`

Transactional caches are supported with Ehcache 2.0 or later. See [Setting Up Transactional Caches](#) for more information on configuring a transactional cache.

See [Cache Concurrency Strategies](#) for more information on selecting a cache concurrency strategy.

3.1.3a Using @Cache

Add the `@Cache` annotation to all entities in your application code that should be cached:

```
@Cache(usage=CacheConcurrencyStrategy.READ_WRITE)
public class Foo {...}
```

`@Cache` must set the cache concurrency strategy for the entity, which in the example above is `READ_WRITE`.

3.1.3b Using the <cache> Element

In the Hibernate mapping file (`hbm.xml` file) for the target entity, set caching for the entity using the

<cache> element:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.my.package">
  <class name="Foo" table="BAR">
    <cache usage="read-write"/>
    <id name="id" column="BAR_ID">
      <generator class="native"/>
    </id>
    <!-- Some properties go here. -->
  </class>
</hibernate-mapping>
```

Use the `usage` attribute to specify the concurrency strategy.

3.1.3c Using the <class-cache> Element

In `hibernate.cfg.xml`, set caching for an entity by using `<class-cache>`, a subelement of the `<session-factory>` element:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

  <session-factory name="java:some/name">
    <!-- Properties go here. -->

    <!-- mapping files -->
    <mapping resource="com/my/package/Foo.hbm.xml"/>

    <!-- cache settings -->
    <class-cache class="com.my.package.Foo" usage="read-write"/>
  </session-factory>
</hibernate-configuration>
```

Use the `usage` attribute to specify the concurrency strategy.

3.1.4 Step 4: Edit Configuration Files

You must edit the Hibernate configuration file to enable and specify the second-level cache provider. You must also edit the Enterprise Ehcache for Hibernate configuration file to configure caching for the Hibernate entities that will be cached and to enable Terracotta clustering.

3.1.4a Hibernate Configuration File

For Hibernate 3.3, you can improve performance by substituting a factory class for the provider class used in previous versions of Hibernate. Add the following to your `hibernate.cfg.xml` file:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property
  name="hibernate.cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheRegionFactory</property>
```

For Hibernate 3.2, which cannot use the factory class, add the following to your `hibernate.cfg.xml` file:

```
<property name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.provider_class">net.sf.ehcache.hibernate.EhCacheProvider</property>
```


TIP: Singletons

To use a singleton version of the provider or factory class, substitute `net.sf.ehcache.hibernate.SingletonEhCacheProvider` or `net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory`.

Singleton CacheManagers are simpler to access and use, and can be helpful in less complex setups where only one configuration is required. Note that a singleton CacheManager should not be used in setups requiring multiple configuration resources or involving multiple instances of Hibernate.

TIP: Spring Users

If you are configuring Hibernate using a Spring context file, you can enable and set the second-level cache provider using values in the `hibernateProperties` property in the bean definition for the session factory:

```
<bean id="mySessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="myDataSource"/>
</property>
<!-- Other properties, such as "mappingResources" listing Hibernate mapping files. -->
  <property name="hibernateProperties">
    <value>
      hibernate.cache.use_second_level_cache=true
      hibernate.cache.region.factory_class=
        net.sf.ehcache.hibernate.EhCacheRegionFactory
    </value>
  </property>
</bean>
```

3.1.4b Enterprise Ehcache Configuration File

Create a basic Ehcache configuration file, `ehcache.xml` by default:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache name="Foo"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="ehcache.xsd">
  <defaultCache
    maxElementsInMemory="0"
    eternal="false"
    timeToIdleSeconds="1200"
    timeToLiveSeconds="1200">
    <terracotta />
  </defaultCache>
  <terracottaConfig url="localhost:9510" />
</ehcache>
```

This defaultCache configuration includes Terracotta clustering. The Terracotta client must load the configuration from a file or a Terracotta server. The value of the `<terracottaConfig />` element's `url` attribute should contain a path to the file or the address and DSO port (9510 by default) of a server. In the example value, "localhost:9510" means that the Terracotta server is on the local host. If the Terracotta configuration source changes at a later time, it must be updated in configuration.

TIP: Terracotta Clients and Servers

In a Terracotta cluster, the application server is also known as the client.

`ehcache.xml` must be on your application's classpath. If you are using a WAR file, add the Ehcache configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.

Specifying Caches for Hibernate Entities

Using an Ehcache configuration file with only a defaultCache configuration means that every cached Hibernate entity is cached with the settings of that defaultCache. You can create specific cache configurations for Hibernate entities using `<cache>` elements.

For example, add the following `<cache>` block to `ehcache.xml` to cache a Hibernate entity that has been configured for caching (see [Step 3: Prepare Your Application for Caching](#)):

```
<cache name="com.my.package.Foo" maxElementsInMemory="1000"
  maxElementsOnDisk="10000" eternal="false" timeToIdleSeconds="3600"
```



```

    timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
<!-- Adding the element <terracotta /> turns on Terracotta clustering for the cache Foo. -->
<terracotta />
</cache>

```

Eviction Settings

You can edit the eviction settings in the defaultCache and any other caches that you configure in `ehcache.xml` to better fit your application's requirements. See [Eviction Parameters](#) for more information

3.1.5 Step 5: Start Your Application with the Cache

You must start both your application and a Terracotta server.

1. Start the Terracotta server.
Start the Terracotta server with the following command:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat
```

2. Start your application.
Your application should now be running with the Terracotta second-level cache.
3. Start the Terracotta Developer Console.
To view the cluster along with the cache, run the following command to start the Terracotta Developer Console:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

4. On the console's initial panel, click **Connect...**
5. In the cluster navigation tree, navigate to **Terracotta cluster > My application > Hibernate**.
Hibernate and second-level cache statistics, as well as other visibility and control panels should be available.

3.1.6 Step 6: Edit the Terracotta Configuration

This step shows you how to run clients and servers on separate machines and add failover (High Availability). You will expand the Terracotta cluster and add High Availability by doing the following:

- Moving the Terracotta server to its own machine
- Creating a cluster with multiple Terracotta servers
- Creating multiple application nodes

These tasks bring your cluster closer to a production architecture.

3.1.6a Procedure:

1. Shut down the Terracotta cluster.
2. Create a Terracotta configuration file called `tc-config.xml` with contents similar to the following

```

<?xml version="1.0" encoding="UTF-8"?>
<!-- All content copyright Terracotta, Inc., unless otherwise indicated. All rights reserved. -->
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-4.xsd"
xmlns:tc="http://www.terracotta.org/config" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <servers>
    <!-- Sets where the Terracotta server can be found. Replace the value of
         host with the server's IP address. -->
    <server host="server.1.ip.address" name="Server1">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
    <!-- If using a standby Terracotta server, also referred to as an ACTIVE-
         PASSIVE configuration, add the second server here. -->
    <server host="server.2.ip.address" name="Server2">
      <data>%(user.home)/terracotta/server-data</data>

```

```

    <logs>%(user.home)/terracotta/server-logs</logs>
  </server>
  <ha>
    <mode>networked-active-passive</mode>
    <networked-active-passive>
      <election-time>5</election-time>
    </networked-active-passive>
  </ha>
</servers>
<!-- Sets where the generated client logs are saved on clients. -->
<clients>
  <logs>%(user.home)/terracotta/client-logs</logs>
</clients>
</tc:tc-config>

```

3. Install Terracotta 3.5.0 on a separate machine for each server you configure in `tc-config.xml`.
4. Copy the `tc-config.xml` to a location accessible to the Terracotta servers.
5. Perform [Step 2: Install and Update the JAR files](#) and [Step 4: Edit Configuration Files](#) steps on each application node you want to run in the cluster.
Be sure to install your application and any application servers on each node.
6. Edit the `<terracottaConfig>` element in Terracotta Distributed Ehcache for Hibernate configuration file, `ehcache.xml`, that you created above:

```

<!-- Add the servers that are configured in tc-config.xml. -->
<terracottaConfig url="server.1.ip.address:9510,server.2.ip.address:9510" />

```

Later in this procedure, you will see where to get more information on editing the settings in the configuration file.

7. Copy `ehcache.xml` to each application node and ensure that it is on your application's classpath (or in `WEB-INF/classes` for web applications).
8. Start the Terracotta server in the following way, replacing "Server1" with the name you gave your server in `tc-config.xml`:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh -f <path/to/tc-config.xml> -n Server1 &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat -f <path\to\tc-config.xml> -n Server1 &
```

If you configured a second server, start that server in the same way on its machine, entering its name after the `-n` flag. The second server to start up becomes the "hot" standby, or PASSIVE. Any other servers you configured will also start up as standby servers.

9. Start all application servers.
10. Start the Terracotta Developer Console and view the cluster.

3.1.7 Step 7: Learn More

To learn more about working with a Terracotta cluster, see the following documents:

- [Working with Terracotta Configuration Files](#) -- Explains how `tc-config.xml` is propagated and loaded in a Terracotta cluster in different environments.
- [Terracotta Server Arrays](#) -- Shows how to design Terracotta clusters that are fault-tolerant, maintain data safety, and provide uninterrupted uptime.
- [Configuring Terracotta Clusters For High Availability](#) -- Defines High Availability configuration properties and explains how to apply them.
- [Terracotta Developers Console Guide](#)

3.2 Testing and Tuning Enterprise Ehcache for Hibernate

This document shows you how to test and tune Enterprise Ehcache for Hibernate.

TIP: Top Tuning Tips

- Set [Eviction Parameters](#)
- Turn Off [Query Cache](#)
- Prevent Unnecessary Database Connections (see [Reducing Unnecessary Database Connections](#))
- Configure Database Connection Pool (see [Connection Pools](#))
- Turn off Unnecessary [Statistics Gathering](#)

3.2.1 Testing the Cache

The main benefit of a Hibernate second-level cache is raising performance by decreasing the number of times an application accesses the database. To gauge the level of database offloading provided by the Enterprise Ehcache for Hibernate second-level cache, look for these benefits:

- Server CPU offload - The CPU load on the database server should decrease.
- Lower latency - The latency for returning data should decrease.
- Higher Transactions per second (TPS) - The TPS rate should increase.
- More concurrency - The number of threads that can access data should increase.

The number of threads that can simultaneously access the distributed second-level cache can be scaled up more easily and efficiently than database connections, which generally are limited by the size of the connection pool.

You should record measurements for all of these factors before enabling the Enterprise Ehcache for Hibernate second-level cache to create a benchmark against which you can assess the impact of using the cache. You should also record measurements for all of these factors before tuning the cache to gauge the impact of any tuning changes you make.

Another important test in addition to performance testing is verifying that the expected data is being loaded. For example, loading one entity can result in multiple cache entries. One approach to tracking cache operations is to set Hibernate cache logging to "debug" in `log4j.properties`:

```
log4j.logger.org.hibernate.cache=debug
```

This level of logging should not be used during performance testing.

NOTE: Optimizing Cache Performance

Before doing performance testing, you should read through the rest of this document to learn about optimizing cache performance. Some performance optimization can be done ahead of time, while some may require testing to reveal its applicability.

When using a testing framework, ensure that the framework does not cause a performance bottleneck and skew results.

3.2.2 Optimizing the Cache Size

Caches that get too large may become inefficient and suffer from performance degradation. A growing rate of flushing and faulting is an indication of a cache that's become too large and should be pruned.

3.2.2a Eviction Parameters

The most important parameters for tuning cache size and cache performance in general are the following:

- Time to Idle (TTI) - This parameter controls how long an entity can remain in the cache without being accessed at least once. TTI is reset each time an entity is accessed. Use TTI to evict little-used entities to shrink the cache or make room for more frequently used entities. Adjust the TTI up if the faulting rate (data faulted in from the database) seems too high, and lower it if flushing (data cleared from the cache) seems too high.
- Time to Live (TTL) - This parameter controls how long an entity can remain in the cache, regardless of how often it is used (it is `_never_` overridden by TTI). Use TTL to prevent the cache from holding stale data. As entities are evicted by TTL, fresh versions are cached the next time they are accessed.

TTI and TTL are set in seconds. See [2.3.4 How Configuration Affects Element Eviction](#) for more information on how configuration affects eviction.

You can also control Hibernate region sizes using the following parameters:

- Target Max In-Memory Count - The maximum number of elements allowed in a region in any one client (any one application server). If this target is exceeded, the client flushes elements to the Terracotta Server Array until the count is within the allowed target. 0 means no eviction takes place (infinite size is allowed).
- Target Max Total Count - The maximum sum total number of elements allowed for a region in all clients (all application servers). If this target is exceeded, eviction occurs to bring the count within the allowed target. 0 means no eviction takes place (infinite size is allowed).

TIP: Default Eviction Parameters

When you first install Enterprise Ehcache for Hibernate, eviction is turned off by default because all eviction parameters are set to 0. You should set these parameters to non-zero values to turn eviction on, then tune them based on how your application requirements and performance characteristics.

How to Set Eviction Parameters

You can set eviction parameters in two different ways:

- In `ehcache.xml` - Configuration file for Enterprise Ehcache for Hibernate with properties for controlling eviction on a per-cache basis. See [Setting Cache Eviction](#) for more information.
- In the Terracotta Developer Console - The GUI for Hibernate second-level cache allows you to apply real-time values to eviction parameters and export a configuration file. For more information, see [8.1.4 Enterprise Ehcache for Hibernate Applications](#).

After setting eviction parameters, be sure to test the effect on performance (see [Testing the Cache](#)) .

3.2.2b Reducing the Cache Miss Rate

The *cache miss rate* is a measure of requests that the cache could not meet. Each miss can lead to a fault which requires a database query. (However, misses and faults are not one-to-one since a query can return results that satisfy more than one miss.) A high or growing cache miss rate indicates the cache should be optimized.

To lower the miss rate, adjust for regions containing entities with high access rates to evict less frequently. This keeps popular entities in the cache for longer periods of time. You should adjust eviction parameter values incrementally and carefully observe the effect on the cache miss rate. For example, TTI and TTL that are set too high can introduce other drawbacks, such as stale data or overly large caches.

3.2.2c Examiner Example

[Examiner](#) , the Terracotta reference application that uses Enterprise Ehcache for Hibernate to implement the second-level cache, supports thousands of concurrent user sessions. This web-based test-taking application caches exams and must have TTI and TTL properly tuned to prevent unnecessarily large data caches and stale exam pages.

The following sections detail how certain cached Examiner data is configured for second-level caching. Included are snippets from the Enterprise Ehcache for Hibernate configuration file (see [Cache Configuration File](#)).

User Roles

The data defining user roles has the following characteristics:

- Never changes - User roles are fixed (read only).
- Accessed frequently - Each user session must have a user role.

Therefore, user roles are cached and never evicted (TTI=0, TTL=0). In general, read-only data that is used frequently and never grows stale should be cached continuously.

```

<cache name="org.terracotta.reference.exam.domain.UserRole"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="0"
  timeToLiveSeconds="0"
  overflowToDisk="false">
  <terracotta/>
</cache>

```

User Data

User data, which includes the user entity and its role, is useful only while the user is active. This data has the following characteristics:

- Access is unpredictable - User interaction with the application is unpredictable and can be sporadic.
- Lifetime is unpredictable - The data is useful as long as the user session has activity. Only when the user becomes inactive are the associated entities idle.

Therefore, these entities should have a short idle time of two minutes (TTI=120) to allow data associated with inactive user sessions to be evicted. However, there should never be eviction based on a hard lifetime (TTL=0), thus allowing the associated entities to be cached indefinitely as long as TTI is reset by activity.

```

<cache name="org.terracotta.reference.exam.domain.User"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="120"
  timeToLiveSeconds="0"
  overflowToDisk="false">
  <terracotta/>
</cache>

```

```

<cache name="org.terracotta.reference.exam.domain.User.roles"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="120"
  timeToLiveSeconds="0"
  overflowToDisk="false">
  <terracotta/>
</cache>

```

Exam Data

Exam data includes the actual exams being taken by users. It has the following characteristics:

- Rarely changes - There is the potential for exam questions to be changed in the database, but this happens infrequently.
- Data set is large - There can be any number of exams, and not all of them can be cached due to limitations on the size of the cache.

Since there can be many different exams, and the potential exists for a cached exam to become stale, cached exams should be periodically evicted based on lack of access (TTI=3600) and to ensure they are up-to-date (TTL=86400).

```

<cache name="org.terracotta.reference.exam.domain.Exam"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="3600"
  timeToLiveSeconds="86400"
  overflowToDisk="false">
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Section"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="3600"
  timeToLiveSeconds="86400"
  overflowToDisk="false">
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Section.questions"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="3600"
  timeToLiveSeconds="86400"
  overflowToDisk="false">
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Section.sections"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="3600"
  timeToLiveSeconds="86400"
  overflowToDisk="false">
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Question"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="3600"
  timeToLiveSeconds="86400"
  overflowToDisk="false">
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Question.choices"
  maxElementsInMemory="1000"
  eternal="false"
  timeToIdleSeconds="3600"
  timeToLiveSeconds="86400"
  overflowToDisk="false">
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Choice"

```

```

maxElementsInMemory="1000"
eternal="false"
timeToIdleSeconds="3600"
timeToLiveSeconds="86400"
overflowToDisk="false">
  <terracotta/>
</cache>

```

3.2.3 Optimizing for Read-Only Data

If your application caches read-only data, the following may improve performance:

- Turn off eviction for often-used, long-lived data (see [Eviction Parameters](#)).
- Turn on key caching (see [Local Key Cache](#)).
- Eliminate "empty" database connections (see [Reducing Unnecessary Database Connections](#)).

3.2.4 Reducing Unnecessary Database Connections

The JDBC mode Autocommit automatically writes changes to the database, making it unnecessary for an application to do so explicitly. However, unnecessary database connections can result from Autocommit because of the way JDBC drivers are designed. For example, transactional read-only operations in Hibernate even those that are resolved in the second-level cache, still generate "empty" database connections. This situation, which can be tracked in database logs, can quickly have a detrimental effect on performance.

Turning off Autocommit should prevent empty database connections, but may not work in all cases. Lazily fetching JDBC connections resolves the issue by preventing JDBC calls until a connection to the database actually needed.

NOTE: Autocommit

While Autocommit should be turned off to reduce unnecessary database connections for applications that create their own transaction boundaries, it may be useful for applications with on-demand (lazy) loading of data. You should investigate Autocommit with your application to discover its effect.

Two options are provided for implementing lazy fetching of database connections:

- [Lazy Fetching with Spring-Managed Transactions](#)
- [Lazy Fetching for Non Spring Applications](#)

3.2.4a Lazy Fetching with Spring-Managed Transactions

If your application is based on the Spring framework, turning off Autocommit may not be enough to reduce unnecessary database connections for transactional read operations. You can prevent these empty database connections from occurring by using the Spring `LazyConnectionDataSourceProxy` proxy definition. The proxy holds unnecessary JDBC calls until a connection to the database is actually required, at which time the held calls are applied.

To implement the proxy, create a target `DataSource` definition (or rename your existing target `DataSource`) and a `LazyConnectionDataSourceProxy` proxy definition in the Spring application context file:

```

<!-- Renamed the existing target DataSource to 'dataSourceTarget' which will be used by the proxy. -->
<bean id="dataSourceTarget"
class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
  <property name="driverClassName"><value>com.mysql.jdbc.Driver</value></property>
  <property name="url"><value>jdbc:mysql://localhost:3306/imagedb</value></property>
  <property name="username"><value>admin</value></property>
  <property name="password"><value></value></property>
  <!-- other datasource configuration properties -->
</bean>
<!-- This is the lazy DataSource proxy that interacts with the target DataSource once a real statement
is sent to the database. Users use this DataSource to set up their Hibernate session factory, which in
turn forces the Hibernate second-level cache and also everything that interacts with that Hibernate
session factory to use it. -->
<bean id="dataSource"
class="org.springframework.jdbc.datasource.LazyConnectionDataSourceProxy">
  <property name="targetDataSource"><ref local="dataSourceTarget"/></property>
</bean>

```

Your application's `SessionFactory`, transaction manager, and all DAOs should access the proxy. Since the proxy implements the `DataSource` interface too, it can simply be passed in instead of the target `DataSource`.

See the [Spring documentation](#) for more information.

3.2.4b Lazy Fetching for Non Spring Applications

By implementing a custom Hibernate connection provider, you can use the `LazyConnectionDataSourceProxy` in a non-Spring based application:

```
public class LazyDBCPConnectionProvider implements ConnectionProvider {
    private DataSource ds;
    private BasicDataSource basicDs;
    public void configure(Properties props) throws HibernateException {
        // DBCP properties used to create the BasicDataSource
        Properties dbcpProperties = new Properties();
        // set some DBCP properties or implement logic to get them from the Hibernate config
        try {
            // Let the factory create the pool
            basicDs = (BasicDataSource)BasicDataSourceFactory.createDataSource(dbcpProperties);
            ds = new LazyConnectionDataSourceProxy(basicDs);
            // The BasicDataSource has lazy initialization
            // borrowing a connection will start the DataSource
            // and make sure it is configured correctly.
            Connection conn = ds.getConnection();
            conn.close();
        } catch (Exception e) {
            String message = "Could not create a DBCP pool";
            if (basicDs != null) {
                try {
                    basicDs.close();
                } catch (Exception e2) {
                    // ignore
                }
                ds = null;
                basicDs = null;
            }
            throw new HibernateException(message, e);
        }
    }
    public Connection getConnection() throws SQLException {
        return ds.getConnection();
    }
    public void closeConnection(Connection conn) throws SQLException {
        conn.close();
    }
    public void close() throws HibernateException {
        try {
            if (basicDs != null) {
                basicDs.close();
                ds = null;
                basicDs = null;
            }
        } catch (Exception e) {
            throw new HibernateException("Could not close DBCP pool", e);
        }
    }
    public boolean supportsAggressiveRelease() {
        return false;
    }
}
```

To use the custom connection provider, update `hibernate.cfg.xml` with the following property:

```
<property name="connection.provider_class">LazyDBCPConnectionProvider</property>
```

3.2.5 Reducing Memory Usage with Batch Processing

If your application must perform a large number of insertions or updates with Hibernate, a potential antipattern can emerge from the fact that all transactional insertions or updates in a session are stored in the first-level cache until flushed. Therefore, waiting to flush until the transaction is committed can result in an `OutOfMemoryException` (OOM) during large operations of this type.

You can prevent OOMs in this case by processing the insertions or updates in batches, flushing after each batch. The Hibernate core documentation gives the following example for inserts:

```
Session session = sessionFactory.openSession();
```



```

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();

```

TIP: session.clear()

The performance of `session.clear()` has been improved in Hibernate 3.3.2.

Updates can be batched similarly. The JDBC batch size referred to in the comment above is set in the Hibernate configuration property `hibernate.jdbc.batch_size`. For more information, see "Batch processing" in the [Hibernate core documentation](#).

3.2.6 Other Important Tuning Factors

The following factors could affect the performance of your second-level cache.

3.2.6a Query Cache

This Hibernate feature creates overhead regardless of how many queries are actually cached. For example, it records timestamps for entities even if not caching the related queries. Query cache is *on* if the following element is set in `hibernate.cfg.xml`:

```
<property name="hibernate.cache.use_query_cache">true</property>
```

If query cache is turned on, two specially-named cache regions appear in the Terracotta Developer Console cache-regions list. The two regions are the query cache and the timestamp cache.

Unless you are certain that the query cache benefits your application, it is recommended that you turn it off (set `hibernate.cache.use_query_cache` to "false").

3.2.6b Connection Pools

If your installation of Hibernate uses JDBC directly, you use a connection pool to create and manage the JDBC connections to a database. Hibernate provides a default connection pool and supports a number of different connection pools. The low-performance default connection pool is inadequate for more than just initial development and testing. Use one of the supported connection pools, such as C3P0 or DBCP, and be sure to set the number of connections to an optimal amount for your application.

3.2.6c Local Key Cache

Enterprise Ehcache for Hibernate can cache a "hotset" of keys on clients to add locality-of-reference, a feature suitable for read-only cases. Note that the set of keys must be small enough for available memory.

See [Terracotta Clustering Configuration Elements](#) for more information on configuring a local key cache.

3.2.6d Hibernate CacheMode

CacheMode is the Hibernate class that controls how a session interacts with second-level and query caches.

If your application explicitly warms the cache (reloads entities), *CacheMode* should be set to `REFRESH` to prevent unnecessary reads and null checks.

3.2.6e Cache Concurrency Strategy

If your application can tolerate somewhat inconsistent views of data, and the data does not change frequently, consider changing the cache concurrency strategy from `READ_WRITE` to `NONSTRICT_READ_WRITE` to boost performance. See [Cache Concurrency Strategies](#) for more information on cache concurrency strategies.

3.2.6f Terracotta Server Optimization

You can optimize the Terracotta servers in your cluster to improve cluster performance with a second-level

cache. Some server optimization requires editing the Terracotta configuration file. For more information on Terracotta configuration file, see:

- [Configuring DSO](#)
- [Working with Terracotta Configuration Files](#)
- [Configuration Guide and Reference](#)

Test the following recommendations to gauge their impact on performance.

Less Aggressive Memory Management

By default, Terracotta servers clear a certain amount of heap memory based on the percentage of memory used. You can configure a Terracotta server to be less aggressive in clearing heap memory by raising the threshold that triggers this action. Allowing more data to remain in memory makes larger caches more efficient by reducing the server's swap-to-disk dependence. Be sure to test any changes to the threshold to confirm that the server doesn't suffer an OOME by failing to effectively manage memory at the new threshold level.

The default threshold is 70 (70 percent of heap memory used). Raise the threshold by setting a higher value for the Terracotta property `l2.cachemanager.threshold` in one of the following ways.

Create a Java Property

To set the threshold at 90, add the following option to `$JAVA_OPTS` before starting the Terracotta server:

```
-Dcom.tc.l2.cachemanager.threshold=90
```

Be sure to export `JAVA_OPTS`. If you adjust the threshold value after the server is running, you must restart the Terracotta server for the new value to take effect.

Add to Terracotta Configuration

Add the following configuration to the top of the Terracotta configuration file (`tc-config.xml` by default) before starting the Terracotta server:

```
<tc-properties>
  <property name="l2.cachemanager.threshold" value="90" />
</tc-properties>
```

You must start the Terracotta server with the configuration file you've updated:

```
start-tc-server.sh -f <path_to_configuration_file>
```

Use `start-tc-server.bat` in Microsoft Windows.

Run in Non-Persistent Mode

If your data is backed by a database, and no critical data exists only in memory, you can run the Terracotta server in non-persistent mode (*temporary-swap-only* mode). By default, Terracotta servers are set to non-persistent mode. For more information on persistence, see the [Terracotta Configuration Guide and Reference](#) .

Reduce the Berkeley DB Memory Footprint

Terracotta allots a certain percentage of memory to Berkeley DB, the database application used to manage the disk store. The default is 25 percent. Under the following circumstances, this percentage can be reduced:

- Running in temporary-swap-only mode (see [Run in Non-Persistent Mode](#)) requires less memory for Berkeley DB since it is managing less data.
- Running with a large heap size may require a smaller percentage of memory for Berkeley DB.

For example, if Berkeley DB has a fixed requirement of 300- 400MB of memory, and the heap size is set to 6GB, Berkeley DB can be allotted eight percent. You can set the percentage using the Terracotta property `l2.berkeleydb.je.maxMemoryPercent` in one of the following ways.

Create a Java Property

To set the percentage at 8, add the following option to `$JAVA_OPTS` (or `$JAVA_OPTIONS`) before starting the Terracotta server:

```
-Dcom.tc.l2.berkeleydb.je.maxMemoryPercent=8
```

Be sure to export `JAVA_OPTS` (or `JAVA_OPTIONS`). If you adjust the percentage value after the server is running, you must restart the Terracotta server for the new value to take effect.

Add to Terracotta Configuration

Add the following configuration to the top of the Terracotta configuration file (`tc-config.xml` by default) before starting the Terracotta server:

```
<tc-properties>
  <property name="l2.berkeleydb.je.maxMemoryPercent" value="8" />
</tc-properties>
```

You must start the Terracotta server with the configuration file you've updated:

```
start-tc-server.sh -f <path_to_configuration_file>
```

Use `start-tc-server.bat` in Microsoft Windows.

If you lower the value of `l2.berkeleydb.je.maxMemoryPercent`, be sure to test the new value's effectiveness by noting the amount of flushing to disk that occurs in the Terracotta server. If flushing rises to a level that impacts performance, increase the value of `l2.berkeleydb.je.maxMemoryPercent` incrementally until an optimal level is observed.

3.2.6g JDK Version

While both JDK 1.5 and 1.6 are supported, JDK 1.6 may deliver better performance.

3.2.6h Statistics Gathering

Each time you connect to the Terracotta cluster with the Developer Console and go to the second-level cache node, Hibernate and cache statistics gathering is automatically started. Since this may have a negative impact on performance, consider disabling statistics gathering during performance tests and in production if you continue to use the Developer Console. To disable statistics gathering, navigate to the **Overview** panel in the **Hibernate** view, then click **Disable Statistics**.

3.2.6i Logging

There is a negative impact on performance if logging is set. Consider disabling statistics logging during performance tests and in production.

To disable statistics gathering in the Terracotta Developer Console, navigate to the **Configuration** panel in the **Hibernate** view, then select the target regions in the list and clear **Logging enabled** if it is set.

To disable debug logging for Enterprise Ehcache, set the logging level for the clustered store to be less granular than FINE.

3.2.6j Java Garbage Collection

Garbage Collection (GC) should be aggressive. Consider using the `-server` Java option on all application servers to force a "server" GC strategy.

3.2.6k Database Tuning

A well-tuned database reduces latency and improves performance:

- Indexes should be optimized for your application.
Databases should be indexed to load data quickly, based on the types of queries your application performs (type of key used, for example).
- Database tables should be of a format that is optimized for your application. In MySQL, for example, the InnoDB format provides better performance than the default MyISAM (or the older ISAM) format if your application performs many transactions and uses foreign keys.
- Ensure that the database is set to accept at least as many connections as the connection pool can open. See [Connection Pools](#) for more information.

The following are issues that could affect the functioning of Enterprise Ehcache for Hibernate.

3.2.6l Unwanted Synchronization with Hibernate Direct Field Access

When direct field access is used, Hibernate uses reflection to access fields, triggering unwanted synchronization that can degrade performance across a cluster. See [this JIRA issue](#) for more information.

3.2.6m Hibernate Exception Thrown With Cascade Option

Under certain circumstances, using a `cascade="all-delete-orphan"` can throw a Hibernate exception. This will happen if you load an object with a `cascade="all-delete-orphan"` collection and then remove the reference to the collection. Don't replace this collection, use `clear()` so the orphan-deletion algorithm can detect your change. See the [Hibernate troubleshooting issue](#) s for more information.

3.2.6n Cacheable Entities and Collections Not Cached

Certain data that should be in the second-level cache may not have been configured for caching (or may have not been configured correctly). This oversight may not cause an error, but may impact performance. See [Finding Cacheable Entities and Collections](#) for more information.

3.3 Enterprise Ehcache for Hibernate Reference

This document contains technical reference information for Enterprise Ehcache for Hibernate.

3.3.1 Cache Configuration File

Note the following about `ehcache.xml` in a Terracotta cluster:

- The copy on disk is loaded into memory from the first Terracotta client (also called application server or node) to join the cluster.
- Once loaded, the configuration is persisted in memory by the Terracotta servers in the cluster and survives client restarts.
- In-memory configuration can be edited in the Terracotta Developer Console. Changes take effect immediately but are *not* written to the original on-disk copy of `ehcache.xml`.
- The in-memory cache configuration is removed with server restarts if the servers are in [non-persistent mode](#), which is the default. The original (on-disk) `ehcache.xml` is loaded.
- The in-memory cache configuration survives server restarts if the servers are in [persistent mode](#) (default is non-persistent). If you are using the Terracotta servers with persistence of shared data, and you want the cluster to load the original (on-disk) `ehcache.xml`, the servers' database must be wiped by removing the data files from the servers' `server-data` directory. This directory is specified in the Terracotta configuration file in effect (`tc-config.xml` by default). Wiping the database causes *all persisted shared data to be lost*.

3.3.1a Setting Cache Eviction

Cache eviction removes elements from the cache based on parameters with configurable values. Having an optimal eviction configuration is critical to maintaining cache performance. For more information on cache eviction, see [2.3.1a Setting Cache Eviction](#).

See [2.3.4 How Configuration Affects Element Eviction](#) for more information on how configuration can impact eviction. See [2.3.1d Terracotta Clustering Configuration Elements](#) for definitions of other available configuration properties.

3.3.1b Cache-Configuration File Properties

See [Terracotta Clustering Configuration Elements](#) for more information.

3.3.1c Exporting Configuration from the Developer Console

To create or edit a cache configuration in a live cluster, see [8.1.3d Editing Cache Configuration](#).

To persist custom cache configuration values, create a cache configuration file by exporting customized configuration from the Terracotta Developer Console or create a file that conforms to the required format. This file must take the place of any configuration file used when the cluster was last started.

3.3.2 Migrating From an Existing Second-Level Cache

If you are migrating from another second-level cache provider, recreate the structure and values of your cache configuration in `ehcache.xml`. Then simply follow the directions for installing and configuring Enterprise Ehcache for Hibernate in [Enterprise Ehcache for Hibernate Express Installation](#).

3.3.3 Cache Concurrency Strategies

A cache concurrency strategy controls how the second-level cache is updated based on how often data is likely to change. Cache concurrency is set using the `usage` attribute in one of the following ways:

- With the `@Cache` annotation:

```
@Cache(usage=CacheConcurrencyStrategy.READ_WRITE)
```

- In the cache-mapping configuration entry in the Hibernate configuration file `hibernate.cfg.xml`.
- In the `<cache>` property of a class or collection mapping file (hbm file).

Supported cache concurrency strategies are described in the following sections.

3.3.3a READ_ONLY

The READ_ONLY strategy works well for unchanging reference data. It can also work in use cases where the cache is periodically invalidated by an external event. That event can flush the cache, then allow it to repopulate.

3.3.3b READ_WRITE

The READ_WRITE strategy works well for data that changes and must be committed. READ_WRITE guarantees correct data at all times by using locks to ensure that transactions are not open to more than one thread.

If a cached element is created or changed in the database, READ_WRITE updates the cache after the transaction completes. A check is done for the element's existence and (if the element exists) for an existing lock. The cached element is guaranteed to be the same version as the one in database.

Note, however, that Hibernate may lock elements before a transaction (update or delete) completes to the database. In this case, other transactions attempting to access those elements will miss and be forced to retrieve the data from the database.

Cache loading is done with checks for existence and version (existing elements that are newer are not replaced).

Enterprise Ehcache for Hibernate is designed to maximize performance with READ_WRITE strategies when the data involved is partitioned by your application (using sticky sessions, for example). However, caching needs are application-dependent and should be investigated on a case-by-case basis.

3.3.3c NONSTRICT_READ_WRITE

The NONSTRICT_READ_WRITE strategy is similar to READ_WRITE, but may provide better performance. NONSTRICT_READ_WRITE works well for data that changes and must be committed, but it does not guarantee exclusivity or consistency (and so avoids the associated performance costs). This strategy allows more than one transaction to simultaneously write to the same entity, and is intended for applications able to tolerate caches that may at times be out of sync with the database.

Because it does not guarantee the stability of data as it is changed in the database, NONSTRICT_READ_WRITE *does not* update the cache when an element is created or changed in the database. However, elements that are updated in the database, *whether or not the transaction completes*, are removed from the cache.

Cache loading is done with no checks, and `get()` operations return null for nonexistent elements.

3.3.3d TRANSACTIONAL

The TRANSACTIONAL strategy is intended for use in an environment utilizing the Java Transaction API (JTA) to manage transactions across a number of XA resources. This strategy guarantees that a cache remains in sync with other resources, such as databases and queues. Hibernate does not use locking for any type of access, but relies instead on a properly configured transactional cache to handle transaction isolation and data integrity.

The TRANSACTIONAL strategy is supported in Ehcache 2.0 and higher. For more information on how to set up a second-level cache with transactional caches, see [Setting Up Transactional Caches](#).

3.3.3e How Entitymanagers Choose the Data Source

Entitymanagers can read data from the cache, or from the database. Which source the entitymanager selects depends on the cache concurrency strategy chosen.

3.3.4 Setting Up Transactional Caches

To set up transactional caches in a second-level cache with Enterprise Ehcache for Hibernate, ensure the following:

Ehcache

- You are using Ehcache 2.1.0 or higher.
- The attribute `transactionalMode` is set to "xa".
- The cache is clustered (the `<cache>` element has the subelement `<terracotta clustered="true">`).
For example, the following cache is configured to be transactional:

```
<cache name="com.my.package.Foo"
  maxElementsInMemory="500"
  eternal="false"
  overflowToDisk="false"
```

```

    transactionalMode="xa">
    <terracotta clustered="true"/>
</cache>

```

- The cache `UpdateTimestampsCache` is not configured to be transactional. Hibernate updates `org.hibernate.cache.UpdateTimestampsCache` that prevents it from being able to participate in XA transactions.

Hibernate

- You are using Hibernate 3.3.
- The factory class used for the second-level cache is `net.sf.ehcache.hibernate.EhCacheRegionFactory`.
- Query cache is turned off.
- The value of `current_session_context_class` is `jta`.
- The value of `transaction.manager_lookup_class` is the name of a `TransactionManagerLookup` class (see your Transaction Manager).
- The value of `transaction.factory_class` is the name of a `TransactionFactory` class to use with the Hibernate Transaction API.
- The cache concurrency strategy is set to `TRANSACTIONAL`.
For example, to set the cache concurrency strategy for `com.my.package.Foo` in `hibernate.cfg.xml`:

```
<class-cache class="com.my.package.Foo" usage="transactional"/>
```

Or in a Hibernate mapping file (hbm file):

```
<cache usage="transactional"/>
```

Or using annotations:

```
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class Foo {...}
```

For more on cache concurrency strategies, see [Cache Concurrency Strategies](#).

3.3.5 Configuring Multiple Hibernate Applications

If you are using more than one Hibernate web application with the Terracotta second-level cache, additional configuration is needed to allow for multiple classloaders. See the section on configuring an application group (*app-groups*) in the [Configuration Guide and Reference](#) for more information on configuring application groups.

3.3.6 Finding Cacheable Entities and Collections

Certain data that should be in the second-level cache may not have been configured for caching. This oversight may not cause an error, but may impact performance.

Using the Terracotta Developer Console, you can compare the set of cached regions with the set of all Hibernate entities and collections. Note any items, such as collections containing fixed or slow-changing data, that appear as Hibernate entities but do not have corresponding cache regions.

3.3.7 Cache Regions in the Object Browser

If the Enterprise Ehcache for Hibernate second-level cache is being clustered correctly, a [Terracotta root](#) representing the second-level cache appears in the Terracotta Developer Console's object browser. Under this root, which exists in every client (application server), are the cached regions and their children.

You can use this root to verify that the second-level cache is running and is clustered with Terracotta:

1. Start the Terracotta server:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh -f <path_to_tc-config.xml> &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat -f <path_to_tc-config.xml>
```

2. Start your application.
You can start more than one instance of your application.
3. Start the Terracotta Developer Console:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

Using the [Terracotta Developer Console](#), verify that there is a root named `default:terracottaHibernateCaches`. For each Terracotta client (application server), the cache should appear as MapEntry objects under this root, one per cache region. The data itself is found inside these cache-region entries.

3.3.8 Hibernate Statistics Sampling Rate

The second-level cache runtime statistics are pulled from Hibernate statistics, which have a fixed sampling rate of one second (sample once per second). The Terracotta Developer Console's sampling rate for display purposes, however, is adjustable.

To display all of the Hibernate statistical counts, set the Terracotta Developer Console's sampling rate to one second. To set the sampling rate, choose **Options...** from the Developer Console's **Tools** menu, then set **Poll period seconds** to "1".

For example, if the sampled Hibernate statistics record the Cache Miss Count values "15, 25, 62, 10, 12, 43," and the Terracotta Developer Console's sampling rate is set to one second, then all of these values are graphed. However, if the Terracotta Developer Console's sampling rate is set to three seconds, then only the values "15, 62, 43" are graphed (assuming that the first poll period coincides with the first value recorded).

3.3.9 Is a Cache Appropriate for Your Use Case?

Some use cases may present hurdles to realizing benefits from a second-level Hibernate cache implementation.

3.3.9a Frequent Updates of Database

Volatile data requires frequent cache invalidation, which increases the overhead of maintaining the cache. At some point this overhead impacts performance at a cost too high to make the cache favorable. Identifying "hotsets" of data can mitigate this situation by limiting the amount of data that requires reloading. Another solution is scaling your cluster to keep more data in memory (see [Terracotta Server Arrays](#)).

3.3.9b Very Large Data Sets

Huge data sets that are queried randomly (or across the set with no clear pattern or "hotsets") are difficult to cache because of the impact on memory of attempting to load that set or having to evict and load elements at a very high rate. Solutions include scaling the cluster to allow more data into memory (see [Terracotta Server Arrays](#)), adding storage to allow Terracotta to spill more data to disk, and using partitioning strategies to prevent any one node from loading too much data.

3.3.9c Frequent Updates of In-Memory Data

As the rate of updating cached data goes up, application performance goes down as Hibernate attempts to manage and persist the changes. Write-behind or some asynchronous approach to writing the data may be a good solution for this issue (see [DSO Async Processing](#)).

3.3.9d Low Frequency of Cached Data Queries

The benefits of caching are maximized when cached data is queried multiple times before expiring. If cached data is infrequently accessed, or often expires before it is used, the benefits of caching may be lost. Solutions to this situation include invalidating data in the cache more often to force updates. Also, refactoring your application to cache more frequently queried data and avoid caching data that tends to expire unused.

3.3.9e Requirements of Critical Data

Cached data cannot be guaranteed to be consistent at all times with the data in a database. In situations where this must be guaranteed, such as when an application requires auditing, access to the data must be through the System of Record (SoR). Financial applications, for example, require auditing, and for this the database must be accessed directly. If critical data is changed in a cache, however, the data obtained from the database could be erroneous.

3.3.9f Database Modified by Other Applications

If data in the database can be modified by applications outside of your application with Hibernate, and that same data is eligible for the second-level cache, unpredictable results could occur. One solution is a redesign

to prevent data that can end up in the cache from being modified by applications outside of the scope of your Hibernate application.

4 Quartz Scheduler

Quartz Scheduler is a full-featured job scheduling service for Java applications. The TerracottaJobStore for Quartz Scheduler clusters that service. Clustering Quartz Scheduler using TerracottaJobStore provides a number of advantages:

- Adds High-Availability - Use "hot" standbys to immediately replace failed servers with no downtime, no lost data.
- Includes a Locality API - Route jobs to specific node groups or base routing decisions on system characteristics and available resources.
- Improves Performance - Offload traditional databases and automatically distribute load.
- Provides a Clear Scale-Out Path - Add capacity without requiring additional database resources.
- Ensures Persistence - Automatically back up shared data without impacting cluster performance.

To install the TerracottaJobStore for Quartz Scheduler, see [Clustering Quartz Scheduler](#).

4.1 Clustering Quartz Scheduler

This document shows you how to add Terracotta clustering to an application that is using Quartz Scheduler. Use this express installation if you have been running your application:

- on a single JVM, or
- on a cluster using JDBC-Jobstore.

To set up the cluster with Terracotta, you will add a Terracotta JAR to each application and run a Terracotta server array. Except as noted below, you can continue to use Quartz in your application as specified in the [Quartz documentation](#).

To add Terracotta clustering to an application that is using Quartz, follow these steps:

4.1.1 Step 1: Requirements

- JDK 1.5 or higher.
- [Terracotta 3.5.0 or higher](#)
Download the kit and run the installer on the machine that will host the Terracotta server.
- All clustered Quartz objects must be serializable.
If you create Quartz objects such as Trigger types, they must be serializable.

4.1.2 Step 2: Install Quartz Scheduler

For guaranteed compatibility, use the JAR files included with the Terracotta kit you are installing. Mixing with older components may cause errors or unexpected behavior.

To install the Quartz Scheduler in your application, add the following JAR files to your application's classpath:

- `${TERRACOTTA_HOME}/quartz/quartz-terracotta-ee-<version>.jar`
<version> is the current version of the Quartz-Terracotta JAR.
- `${TERRACOTTA_HOME}/quartz/quartz-<quartz-version>.jar`
- <quartz-version> is the current version of Quartz (1.7.0 or higher).
- `${TERRACOTTA_HOME}/common/terracotta-toolkit-<API-version>-runtime-ee-<version>.jar`
The Terracotta Toolkit JAR contains the Terracotta client libraries. <API-version> refers to the Terracotta Toolkit API version. <version> is the current version of the Terracotta Toolkit JAR.

If you are using the open-source edition of the Terracotta kit, no JAR files will have "-ee-" as part of their name.

If you are using a WAR file, add these JAR files to its `WEB-INF/lib` directory.

NOTE: Application Servers

Most application servers (or web containers) should work with this installation of the Quartz Scheduler. However, note the following:

- GlassFish application server - You must add the following to `domains.xml` :

```
<jvm-options>-Dcom.sun.enterprise.server.ss.ASQuickStartup=false</jvm-  
options>
```

- WebLogic application server - You must use the [supported version](#) of WebLogic.

4.1.3 Step 3: Configure Quartz Scheduler

The Quartz configuration file, `quartz.properties` by default, should be on your application's classpath. If you are using a WAR file, add the Quartz configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.

4.1.3a Add Terracotta Configuration

To be clustered by Terracotta, the following properties in `quartz.properties` must be set as follows:

```
# Do not use the jobStore class TerracottaJobStore unless you are using the open-source version of the
Terracotta Server Array.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl = <path/to/Terracotta/configuration>
```

The property `org.quartz.jobStore.tcConfigUrl` must point the client (or application server) at the location of the Terracotta configuration.

TIP: Terracotta Clients and Servers

In a Terracotta cluster, the application server is also known as the client.

The client must load the configuration from a file or a Terracotta server. If loading from a server, give the server's hostname and its Terracotta DSO port (9510 by default). The following example shows a configuration that is loaded from the Terracotta server on the local host:

```
# Do not use the jobStore class TerracottaJobStore unless you are using the open-source version of the
Terracotta Server Array.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl = localhost:9510
```

To load Terracotta configuration from a Terracotta configuration file (`tc-config.xml` by default), use path. For example, if the Terracotta configuration file is located on `myHost.myNet.net` at `/usr/local/TerracottaHome`, use the full URL along with the configuration file's name:

```
# Do not use the jobStore class TerracottaJobStore unless you are using the open-source version of
Terracotta Server Array.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl = file://myHost.myNet.net/usr/local/TerracottaHome/tc-config.xml
```

If the Terracotta configuration source changes at a later time, it must be updated in configuration.

4.1.3b Scheduler Instance Name

A Quartz scheduler has a default name configured by the following `quartz.properties` property:

```
org.quartz.scheduler.instanceName = QuartzScheduler
```

Setting this property is not required. However, you can use this property to instantiate and differentiate between two or more instances of the scheduler, each of which then receives a separate store in the Terracotta cluster.

Using different scheduler names allows you to isolate different job stores within the Terracotta cluster (logically unique scheduler instances). Using the same scheduler name allows different scheduler instances to share the same job store in the cluster.

4.1.4 Step 4: Start the Cluster

1. Start the Terracotta server:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat
```

2. Start the application servers.
3. Start the Terracotta Developer Console:

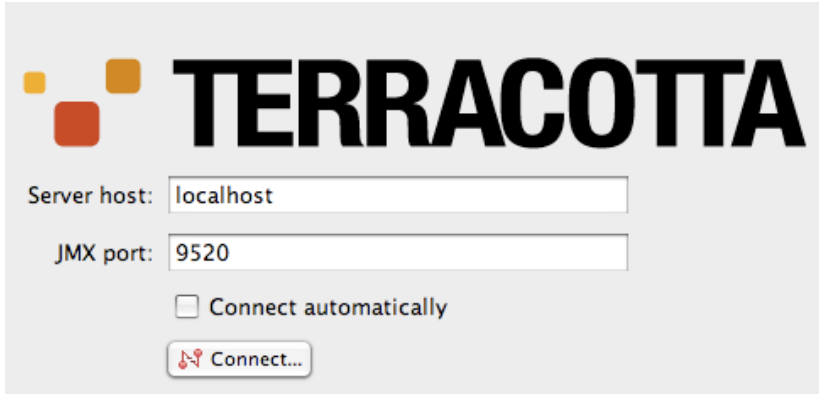
UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

Microsoft Windows

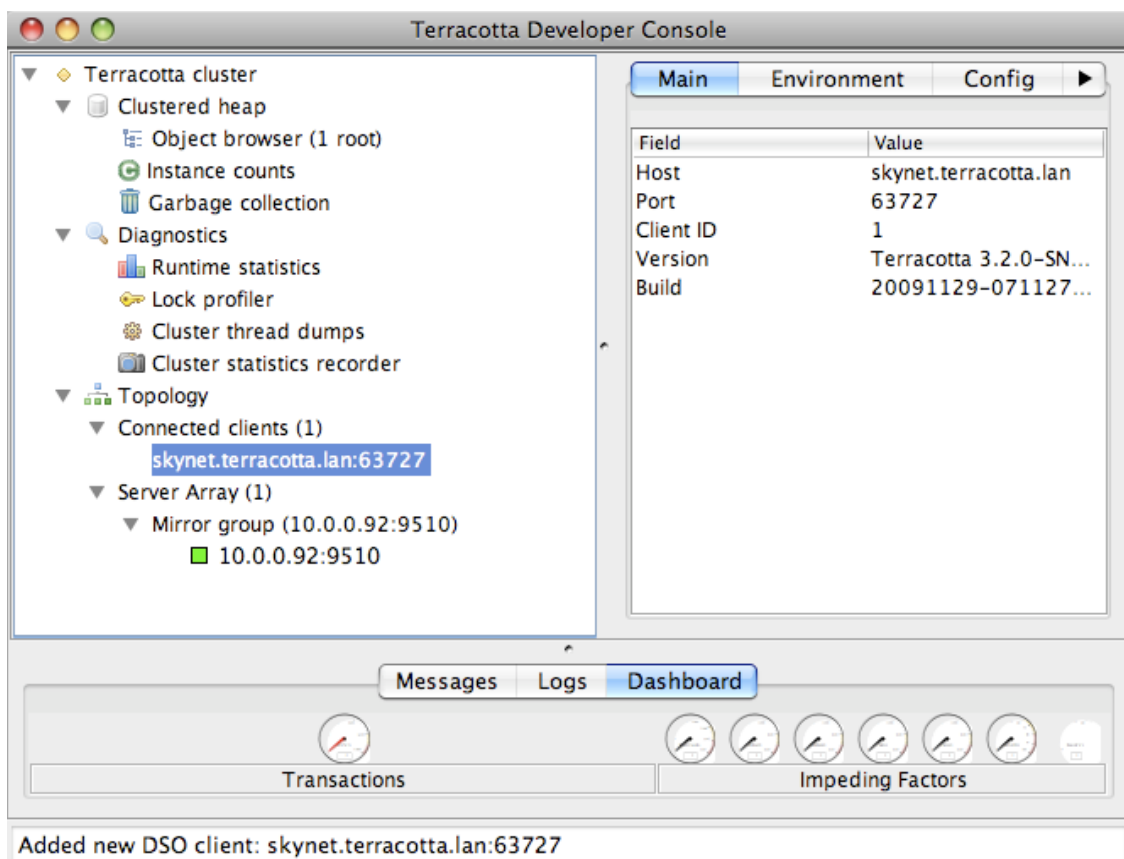
```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

4. Connect to the Terracotta cluster.
Click **Connect...** in the Terracotta Developer Console.



The image shows the Terracotta Developer Console connection interface. It features the Terracotta logo at the top left. Below it, there are two input fields: 'Server host:' with 'localhost' entered, and 'JMX port:' with '9520' entered. There is a checkbox labeled 'Connect automatically' which is currently unchecked. At the bottom, there is a 'Connect...' button with a red network icon.

5. Click the **Topology** node in the cluster navigation window to see the Terracotta servers and clients (application servers) in the Terracotta cluster.
Your console should have a similar appearance to the following figure.



4.1.5 Step 5: Edit the Terracotta Configuration

This step shows you how to run clients and servers on separate machines and add failover (High Availability). You will expand the Terracotta cluster and add High Availability by doing the following:

- Moving the Terracotta server to its own machine
- Creating a cluster with multiple Terracotta servers
- Creating multiple application nodes

These tasks bring your cluster closer to a production architecture.

4.1.5a Procedure:

1. Shut down the Terracotta cluster.
2. Create a Terracotta configuration file called `tc-config.xml` with contents similar to the following

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- All content copyright Terracotta, Inc., unless otherwise indicated. All rights reserved. -->
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd"
xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <servers>
    <!-- Sets where the Terracotta server can be found. Replace the value of      host with the
server's IP address. -->
    <server host="server.1.ip.address" name="Server1">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
    <!-- If using a standby Terracotta server, also referred to as an ACTIVE-PASSIVE configuration, add
the second server here. -->
    <server host="server.2.ip.address" name="Server2">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
    <!-- If using more than one server, add an <ha> section. -->
    <ha>
      <mode>networked-active-passive</mode>
      <networked-active-passive>
        <election-time>5</election-time>
      </networked-active-passive>
    </ha>
  </servers>
  <!-- Sets where the generated client logs are saved on clients. -->
  <clients>
    <logs>%(user.home)/terracotta/client-logs</logs>
  </clients>
</tc:tc-config>
```

3. Install Terracotta 3.5.0 on a separate machine for each server you configure in `tc-config.xml`.
4. Copy the `tc-config.xml` to a location accessible to the Terracotta servers.
5. Perform [Step 2: Install Quartz Scheduler](#) and [Step 3: Configure Quartz Scheduler](#) on each application node you want to run in the cluster.
Be sure to install your application and any application servers on each node.
6. Edit the `org.quartz.jobStore.tcConfigUrl` property in `quartz.properties` to list both Terracotta servers:

```
org.quartz.jobStore.tcConfigUrl = server.1.ip.address:9510,server.2.ip.address:9510
```

7. Copy `quartz.properties` to each application node and ensure that it is on your application's classpath. If you are using a WAR file, add the Quartz configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.
8. Start the Terracotta server in the following way, replacing "Server1" with the name you gave your server in `tc-config.xml`:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh -f <path/to/tc-config.xml> -n Server1 &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat -f <path\to\tc-config.xml> -n Server1 &
```

If you configured a second server, start that server in the same way on its machine, entering its name after the `-n` flag. The second server to start up becomes the "hot" standby, or PASSIVE. Any other servers you configured will also start up as standby servers.

9. Start all application servers.
10. Start the Terracotta Developer Console and view the cluster.

4.1.6 Step 6: Learn More

To learn more about working with a Terracotta cluster, see the following documents:

- [6.1 Working with Terracotta Configuration Files](#) - Explains how `tc-config.xml` is propagated and loaded in a Terracotta cluster in different environments.
- [6.3 Terracotta Server Arrays](#) - Shows how to design Terracotta clusters that are fault-tolerant, maintain data safety, and provide uninterrupted uptime.

- [6.2 Configuring Terracotta Clusters For High Availability](#) - Defines High Availability configuration properties and explains how to apply them.
- [8.1 Terracotta Developer Console](#) - Provides visibility into and control of caches.

4.2 Quartz Scheduler Reference

This section contains information on functional aspects of Terracotta Quartz Scheduler and optimizing your use of TerracottaJobstore for Quartz Scheduler.

4.2.1 Quartz Scheduler Where (Locality API)

Terracotta Quartz Scheduler Where is an Enterprise feature that allows jobs and triggers to run on specified Terracotta clients instead of randomly chosen ones. Quartz Scheduler Where provides a locality API that has a more readable fluent interface for creating and scheduling jobs and triggers. This locality API, together with configuration, can be used to route jobs to nodes based on defined criteria:

- Specific resources constraints such as free memory.
- Specific system characteristics such as type of operating system.
- A member of a specified group of nodes.

This section shows you how to install, configure, and use the locality API.

4.2.1a Installing Quartz Scheduler Where

To access the Quartz Scheduler Locality API in a standard installation, include the file `quartz-terracotta-ee-<version>.jar` in your classpath, where `<version>` is Quartz version 2.0.0 or higher. This jar is found under the `${TERRACOTTA_HOME}/quartz` directory.

For DSO installation, see [9.1.6 Quartz Scheduler DSO Installation](#).

4.2.1b Configuring Quartz Scheduler Where

To configure Quartz Scheduler Where, follow these steps:

1. Edit `quartz.properties` to cluster with Terracotta.
See [4.1 Clustering Quartz Scheduler](#) for more information.
2. If you intend to use node groups, configure an implementation of `org.quartz.spi.InstanceIdGenerator` to generate instance IDs to be used in the locality configuration.
See [4.2.1c Understanding Generated Node IDs](#) for more information about generating instance IDs.
3. Configure the node and trigger groups in `quartzLocality.properties`. For example:

Set up node groups that can be referenced from application code. The values shown are instance IDs:

```
org.quartz.locality.nodeGroup.slowJobs = node0, node3
org.quartz.locality.nodeGroup.fastJobs = node1, node2
org.quartz.locality.nodeGroup.allNodes = node0, node1, node2, node3
```

Set up trigger groups whose triggers fire only on nodes in the specified node groups. For example, a trigger in the trigger group `slowTriggers` will fire only on `node0` and `node3`:

```
org.quartz.locality.nodeGroup.slowJobs.triggerGroups = slowTriggers
org.quartz.locality.nodeGroup.fastJobs.triggerGroups = fastTriggers
```

4. Ensure that `quartzLocality.properties` is on the classpath, the same as `quartz.properties`.

See [4.2.1e Quartz Scheduler Where Code Sample](#) for an example of how to use Quartz Scheduler Where.

4.2.1c Understanding Generated Node IDs

Terracotta clients each run an instance of a clustered Quartz Scheduler scheduler. Every instance of this clustered scheduler must use the same scheduler name, specified in `quartz.properties`. For example:

```
# Name the clustered scheduler.
org.quartz.scheduler.instanceName = myScheduler
```

`myScheduler`'s data is shared across the cluster by each of its instances. However, every instance of `myScheduler` must also be identified uniquely, and this unique ID is specified in `quartz.properties` by the property `org.quartz.scheduler.instanceId`. This property should have one of the following values:

- `<string>` - A string value that identifies the scheduler instance running on the Terracotta client that

- loaded the containing `quartz.properties`. Each scheduler instance must have a unique ID value
- **AUTO** - Delegates the generation of unique instance IDs to the class specified by the property `org.quartz.scheduler.instanceIdGenerator.class`.

For example, you can set `org.quartz.scheduler.instanceId` to equal "node1" on one node, "node2" on another node, and so on.

If you set `org.quartz.scheduler.instanceId` equal to "AUTO", then you should specify a generator class in `quartz.properties` using the property `org.quartz.scheduler.instanceIdGenerator.class`. This property can have one of the values listed in the following table.

Value	Notes
<code>org.quartz.simpl.HostnameInstanceIdGenerator</code>	Returns the hostname as the instance ID.
<code>org.quartz.simpl.SystemPropertyInstanceIdGenerator</code>	Returns the value of the <code>org.quartz.scheduler.instanceId</code> system property. Available with Quartz 2.0 or higher.
<code>org.quartz.simpl.SimpleInstanceIdGenerator</code>	Returns an instance ID composed of the local hostname with the current timestamp appended. Ensures a unique name. If you do not specify a generator class, this generator class is used by default. However, this class is not suitable for use with Quartz Scheduler Where because the IDs it generates are not predictable.
Custom	Specify your own implementation of the interface <code>org.quartz.spi.InstanceIdGenerator</code> .

Using SystemPropertyInstanceIdGenerator

`org.quartz.simpl.SystemPropertyInstanceIdGenerator` is useful in environments that use initialization scripts or configuration files. For example, you could add the `instanceId` property to an application server's startup script in the form `-Dorg.quartz.scheduler.instanceId=node1`, where "node1" is the instance ID assigned to the local Quartz Scheduler scheduler. Or it could also be added to a configuration resource such as an XML file that is used to set up your environment.

The `instanceId` property values configured for each scheduler instance can be used in `quartzLocality.properties` node groups. For example, if you configured instance IDs `node1`, `node2` and `node3`, you can use these IDs in node groups:

```
org.quartz.locality.nodeGroup.group1 = node1, node2
org.quartz.locality.nodeGroup.allNodes = node1, node2, node3
```

4.2.1d Available Constraints

Quartz Scheduler Where offers the following constraints:

- CPU - Minimum number of cores and available threads, maximum amount of load.
- Resident keys - Use a node with a specified Enterprise Ehcache distributed cache that has the best match for the specified keys.
- Memory - Minimum amount of memory available.
- Node group - A node in the specified node group, as defined in `quartzLocality.properties`.
- OS - A node running the specified operating system.

4.2.1e Quartz Scheduler Where Code Sample

A cluster has Terracotta clients running Quartz Scheduler running on the following hosts: `node0`, `node1`, `node2`, `node3`. These hostnames are used as the instance IDs for the Quartz Scheduler scheduler instances because the following `quartz.properties` properties are set as shown:

```
org.quartz.scheduler.instanceId = AUTO
```

```
#This sets the hostnames as instance IDs:
```

```
org.quartz.scheduler.instanceIdGenerator.class = org.quartz.simpl.HostnameInstanceIdGenerator
```

`quartzLocality.properties` has the following configuration:


```

org.quartz.locality.nodeGroup.slowJobs = node0, node3
org.quartz.locality.nodeGroup.fastJobs = node1, node2
org.quartz.locality.nodeGroup.allNodes = node0, node1, node2, node3

```

```

org.quartz.locality.nodeGroup.slowJobs.triggerGroups = slowTriggers
org.quartz.locality.nodeGroup.fastJobs.triggerGroups = fastTriggers

```

The following code snippet uses Quartz Scheduler Where to create locality-aware jobs and triggers.

```

// Note the static imports of builder classes that define a Domain Specific Language (DSL).
import static org.quartz.JobBuilder.newJob;
import static org.quartz.TriggerBuilder.newTrigger;
import static org.quartz.locality.LocalityTriggerBuilder.localTrigger;
import static org.quartz.locality.NodeSpecBuilder.node;
import static org.quartz.locality.constraint.NodeGroupConstraint.partOfNodeGroup;

import org.quartz.JobDetail;
import org.quartz.locality.LocalityTrigger;
// Other required imports...

// Using the Quartz Scheduler fluent interface, or the DSL.

// Create a locality-aware job that can be run on any node from nodeGroup "group1" that runs a Linux OS:
LocalityJobDetail jobDetail1 =
    localJob(
        newJob(myJob1.class)
            .withIdentity("myJob1")
            .storeDurably(true)
            .build()
        .where(
            node()
                .is(partOfNodeGroup("group1"))
                .is(Constraint.LINUX))
        .build();

// Create a trigger for myJob1:
Trigger trigger1 = newTrigger()
    .forJob("myJob1")
    .withIdentity("myTrigger1")
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(10)
        .withRepeatCount(2))
    .build();

// Create a second job:
JobDetail jobDetail2 = newJob(myJob2.class)
    .withIdentity("myJob2")
    .storeDurably(true)
    .build();

// Create a locality-aware trigger for myJob2.
LocalityTrigger trigger2 =
    localTrigger(newTrigger()
        .forJob("myJob2")
        .withIdentity("myTrigger2")
        .where(
            node()
                .is(partOfNodeGroup("allNodes"))) // fire on any node in allNodes
                .has(atLeastAvailable(100, MemoryConstraint.Unit.MB)) // with at least 100MB in
free memory.
        .build());

// The following trigger will fire myJob1 on any node in the allNodes group that's running Linux.
LocalityTrigger trigger3 =
    localTrigger(newTrigger()
        .forJob("myJob1")
        .withIdentity("myTrigger3")
        .where(
            node()
                .is(partOfNodeGroup("allNodes")))
        .build());

```

```
// The following job detail sets up a job (cacheJob) that will be fired on the node where myCache has,
// locally, the most keys specified in the collection myKeys.

// After the best match is found, missing elements will be faulted in. If these types of jobs are fired
// frequently and a large amount of data must often be faulted in, performance could degrade. To maintain
// performance, ensure that most of the targeted data is already cached.

Cache myCache = cacheManager.getEhcache("myCache"); // myCache is already configured, populated, and
distributed.

// A Collection is needed to hold the keys for the elements to be targeted by cacheJob.
// The following assumes String keys.

Set<String> myKeys = new HashSet<String>();

... // Populate myKeys with the keys for the target elements in myCache.

// Create the job that will do work on the target elements.

LocalityJobDetail cacheJobDetail =
    localJob(
        newJob(cacheJob.class)
            .withIdentity("cacheJob")
            .storeDurably(true)
            .build()
            .where(
                node()
                    .has(elements(myCache, myKeys)))
            .build());
```

Notice that trigger3, the third trigger defined, overrode the partOfNodeGroup constraint of myJob1. Where triggers and jobs have conflicting constraints, the triggers take priority. However, since trigger3 did not provide an OS constraint, it did *not* override the OS constraint in myJob1.

NOTE: Unmet Constraints Cause Errors

If a trigger cannot fire because it has constraints that cannot be met by any node, that trigger will go into an error state. Applications using Quartz Scheduler Where with constraints should be tested under conditions that simulate those constraints in the cluster.

This example showed how memory and node-group constraints are used to route locality-aware triggers and jobs. trigger2, for example, is set to fire myJob2 on a node in a specific group ("allNodes") with a specified minimum amount of free memory. A constraint based on operating system (Linux, Microsoft Windows, Apple OSX, and Oracle Solaris) is also available.

4.2.1f Locality With the Standard Quartz Scheduler API

It is also possible to add locality to jobs and triggers created with the standard Quartz Scheduler API by assigning the triggers to a trigger group specified in `quartzLocality.properties`.

4.2.2 Execution of Jobs

In the general case, exactly one Quartz Scheduler node, or Terracotta client, executes a clustered job when that job's trigger fires. This can be any of the nodes that have the job. If a job repeats, it may be executed by any of the nodes that have it exactly once per the interval configured. It is not possible to predict which node will execute the job.

With Quartz Scheduler Where, a job can be assigned to a specific node based on certain criteria.

4.2.3 Working With JobDataMaps

JobDataMaps contain data that may be useful to jobs at execution time. A JobDataMap is stored at the time its associated job is added to a scheduler.

4.2.3a Updating a JobDataMap

If the stored job is stateful (implements the StatefulJob interface), and the contents of its JobDataMap is updated (from within the job) during execution, then a new copy of the JobDataMap is stored when the job completes.

If the job not stateful, then it must be explicitly stored again with the changed JobDataMap to update the

stored copy of the job's JobDataMap. This is because TerracottaJobStore contains deep copies of JobDataMap objects and does not reflect updates made after a JobDataMap is stored.

4.2.3b Best Practices for Storing Objects in a JobDataMap

Because TerracottaJobStore contains deep copies of JobDataMap objects, application code should not have references to mutable JobDataMap objects. If an application does rely on these references, there is risk of getting stale data as the mutable objects in a deep copy do not reflect changes made to the JobDataMap after it is stored.

To maximize performance and ensure long-term compatibility, place only Strings and primitives in JobDataMap. JobDataMap objects are serialized and prone to class-versioning issues. Putting complex object into a clustered JobDataMap could also introduce other errors that are avoided with Strings and primitives.

4.2.4 Cluster Data Safety

By default, Terracotta clients (application servers) do not block to wait for a "transaction received" acknowledgement from a Terracotta server when writing data transactions to the cluster. This asynchronous write mode translates into better performance in a Terracotta cluster.

However, the option to maximize data safety by requiring that acknowledgement is available using the following Quartz configuration property:

```
org.quartz.jobStore.synchronousWrite = true
```

When synchronousWrite is set to "true", a client blocks with each transaction written to the cluster until an acknowledgement is received from a Terracotta server. This ensures that the transaction is committed in the cluster before the client continues work.

4.2.5 Effective Scaling Strategies

Clustering Quartz schedulers is an effective approach to distributing load over a number of nodes if jobs are long-running or are CPU intensive (or both). Distributing the jobs lessens the burden on system resources. In this case, and with a small set of jobs, lock contention is usually infrequent.

However, using a single scheduler forces the use of a cluster-wide lock, a pattern that degrades performance as you add more clients. The cost of this cluster-wide lock becomes more evident if a large number of short-lived jobs are being fired by a single scheduler. In this case, consider partitioning the set of jobs across more than one scheduler.

If you do employ multiple schedulers, they can be run on every node, striping the cluster-wide locks. This is an effective way to reduce lock contention while adding scale.

If you intend to scale, measure your cluster's throughput in a test environment to discover the optimal number of schedulers and nodes.

5 Clustering Web Applications with Terracotta Web Sessions

This document shows how to cluster web applications with Terracotta Web Sessions.

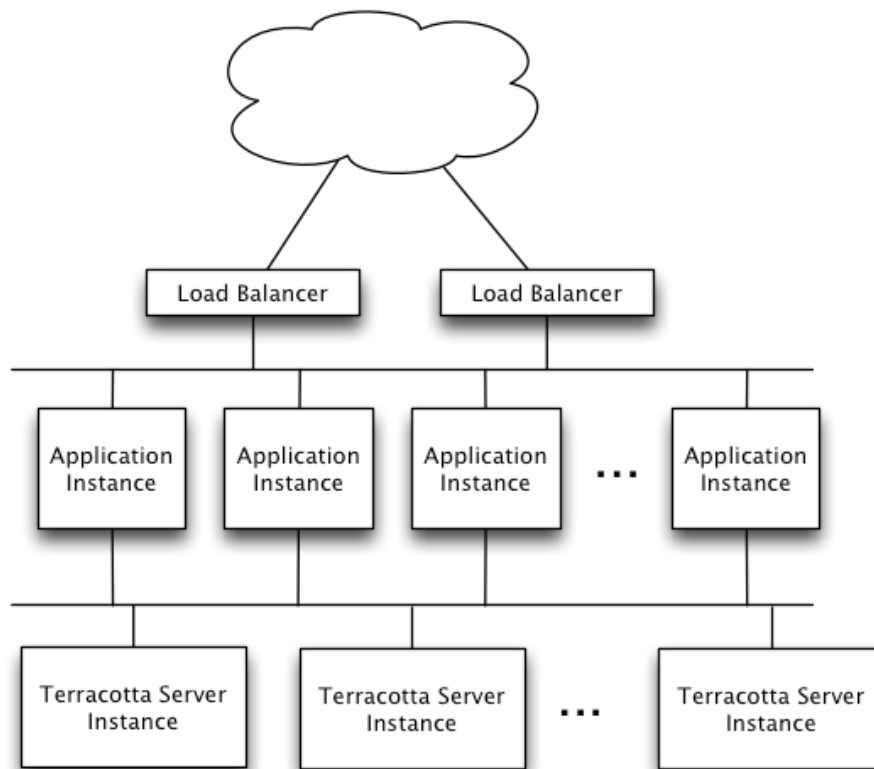
Terracotta clusters web applications based on a number of popular web containers (or application servers). See [Platform Support](#) for certified containers and supported versions. Terracotta Web Sessions provides a number of advantages over container session-replication schemes, including the following:

- No Loss of Data - Robust session-data persistence backed by the Terracotta Server Array, including failover with no data loss.
- Consistent Data - Consistency guarantees across the entire cluster.
- No Size Limit - Unlimited scale-out based on the Terracotta Server Array.
- Memory Efficient - No consumption of heap on clients for the purpose of backing up session data.

To install Web Sessions, see [Web Sessions Installation](#).

5.0.1 Architecture of a Terracotta Cluster

The following diagram shows the architecture of a typical Terracotta-enabled web application.



The load balancer parcels out HTTP requests from the Internet to each application server. To maximize the locality of reference of the clustered HTTP session data, the load balancer uses HTTP session affinity so all requests corresponding to the same HTTP session are routed to the same application server. However, with a Terracotta-enabled web application, any application server can process any request. Terracotta Web Sessions clusters the sessions, allowing sessions to survive node hops and failures.

The application servers run both your web application *and* the Terracotta client software, and are called "clients" in a Terracotta cluster. As many application servers may be deployed as needed to handle your site load.

For more information on sizing and deployment concerns, see the [Deployment Guide](#) and the [Operations Guide](#).

A Terracotta cluster can be deployed with one or more Terracotta servers, which act as the data store for HTTP session data and coordinate access by the application servers to that session data. For more information on setting up a Terracotta cluster, see [Configuring Terracotta Clusters For High Availability](#) and [Terracotta Server Arrays](#).

5.1 Web Sessions Installation

This document shows you how to cluster web sessions *without* the requirements imposed by Terracotta DSO, such as cluster-wide locking, class instrumentation, and class portability. If you must use DSO, see [Terracotta DSO Installation](#).

5.1.1 Step 1: Requirements

- JDK 1.5 or higher.
- [Terracotta 3.5.0 or higher](#)
Download the kit and run the installer on the machine that will host the Terracotta server. For use with WebSphere, you must have kit version 3.2.1_2 or higher.
- All clustered objects must be serializable.
If you cannot use Serializable classes, you must use the custom web-sessions installation (see [Terracotta DSO Installation](#)). Clustering non-serializable classes is not supported with the express installation.
- An application server listed in [Step 2: Install the Terracotta Sessions JAR](#).

5.1.2 Step 2: Install the Terracotta Sessions JAR

For guaranteed compatibility, use the JAR files included with the Terracotta kit you are installing. Mixing

with older components may cause errors or unexpected behavior.

To cluster your application's web sessions, add the following JAR file to your application's classpath:

- `${TERRACOTTA_HOME}/sessions/terracotta-session-<version>.jar`
 <version> is the current version of the Terracotta Sessions JAR. For use with WebSphere, you must have Terracotta Sessions JAR version 1.0.2 or higher.
- `${TERRACOTTA_HOME}/common/terracotta-toolkit-<API-version>-runtime-ee-<version>.jar`
 The Terracotta Toolkit JAR contains the Terracotta client libraries. <API-version> refers to the Terracotta Toolkit API version. <version> is the current version of the Terracotta Toolkit JAR.

If you are using the open-source edition of the Terracotta kit, no JAR files will have "-ee-" as part of their name.

See the following table on suggestions on where to add the Terracotta Sessions JAR based on application server.

Application Server	Suggested Location for Terracotta Sessions JAR File
JBoss AS (earlier than 6.0)	<jboss install dir>/lib
JBoss AS 6.0	<jboss install dir>/common/lib
Jetty	WEB-INF/lib
Tomcat 5.0 and 5.5	\$CATALINA_HOME/server/lib
Tomcat 6.0	\$CATALINA_HOME/lib
WebLogic	WEB-INF/lib
WebSphere	WEB-INF/lib

NOTE: Supported Application Servers

The table above lists the only application servers supported by the express installation. See [Platform Support](#) to obtain the latest versions for the listed application servers.

5.1.3 Step 3: Configure Web-Session Clustering

Terracotta servers, and Terracotta clients running on the application servers in the cluster, are configured with a Terracotta configuration file, `tc-config.xml` by default. Servers not started with a specified configuration use a default configuration.

To add Terracotta clustering to your application, you must specify how Terracotta clients get their configuration by including the source in `web.xml` or `context.xml`.

Find the configuration to use for your application server in the sections below.

5.1.3a Jetty, WebLogic, and WebSphere

Add the following configuration snippet to `web.xml`:

```
<filter>
  <filter-name>terracotta</filter-name>
  <!-- The filter class is specific to the application server. -->
  <filter-class>org.terracotta.session.{container-specific-class}</filter-class>
  <init-param>
    <param-name>tcConfigUrl</param-name>
    <!-- <init-param> of type tcConfigUrl has a <param-value> element containing the URL or filepath
    (for example, /lib/tc-config.xml) to tc-config.xml. If the Terracotta configuration source changes at a
    later time, it must be updated in configuration. -->
    <param-value>localhost:9510</param-value>
  </init-param>
</filter>
<filter-mapping>
  <!-- Must match filter name from above. -->
  <filter-name>terracotta</filter-name>
```

```

<url-pattern>*/</url-pattern>
<!-- Enable all available dispatchers. -->
<dispatcher>ERROR</dispatcher>
<dispatcher>INCLUDE</dispatcher>
<dispatcher>FORWARD</dispatcher>
<dispatcher>REQUEST</dispatcher>
</filter-mapping>

```

<filter-name> can contain a string of your choice. However, the value of <filter>/<filter-name> must match <filter-mapping>/<filter-name>.

Choose the appropriate value for <filter-class> from the following table.

Container	Value of <filter-class>
Jetty 6.1	org.terracotta.session.TerracottaJet
WebLogic 9	org.terracotta.session.TerracottaWeb
WebLogic 10	org.terracotta.session.TerracottaWeb
WebSphere 6.1	org.terracotta.session.TerracottaWebsphere61xSess

If you have customized a Terracotta configuration file and want to include its contents rather than providing an URL, use an <init-param> of type tcConfig :

```

<init-param>
  <param-name>tcConfig</param-name>
  <param-value>&lt;tc:tc-config ... &lt;/tc:tc-config&gt;</param-value>
</init-param>

```

Use URL-safe codes (also known as "URL escaping") or HTML names for all special characters such as angle brackets ("<" and ">").

Ensure that the Terracotta filter is the first <filter> element listed in web.xml . Filters processed ahead of the Terracotta valve may disrupt its processing.

web.xml should be in /WEB-INF if you are using a WAR file.

5.1.3b JBoss AS and Tomcat

Add the following to context.xml :

```
<Valve className="org.terracotta.session.{container-specific-class}" tcConfigUrl="localhost:9510">
```

where tcConfigUrl contains an URL or file path (for example, tcConfigURL="/lib/tc-config.xml") to tc-config.xml . If the Terracotta configuration source changes at a later time, it must be updated in configuration.

If you have customized a Terracotta configuration file and want to include its contents rather than providing an URL, replace tcConfigUrl with tcConfig:

```
<Valve className="org.terracotta.session.{container-specific-class}" tcConfig="&lt;tc:tc-config ...
&lt;/tc:tc-config&gt;";>
```

Use URL-safe codes (also known as "URL escaping") or HTML names for all special characters such as angle brackets ("<" and ">"). Choose the appropriate value of className from the following table.

Container	Value of className
JBoss Application Server 4.0	org.terracotta.session.TerracottaJbc
JBoss Application Server 4.2	org.terracotta.session.TerracottaJbc
JBoss Application Server 5.1	org.terracotta.session.TerracottaJbc
JBoss Application Server 6.0	org.terracotta.session.TerracottaJbc
Tomcat 5.0	org.terracotta.session.TerracottaTom
Tomcat 5.5	org.terracotta.session.TerracottaTom

Tomcat 6.0

org.terracotta.session.TerracottaTom

For example, to use Tomcat 6.0, the content of `context.xml` should be similar to the following:

```
<Context>
<Valve className="org.terracotta.session.TerracottaTomcat60xSessionValve" tcConfigUrl="localhost:9510"/>
</Context>
```

Ensure that the Terracotta valve is the first `<Valve>` element listed in `context.xml`. Valves processed ahead of the Terracotta valve may disrupt its processing.

NOTE: Using Tomcat's Built-In Authentication

If you use one of the authenticator valves available with Tomcat, you may encounter an `UnsupportedOperationException` when running with Terracotta clustering. With Tomcat 5.5 and above, users can prevent this error by disabling `changeSessionIdOnAuthentication`. For example:

```
<Valve changeSessionIdOnAuthentication="false"
className="org.apache.catalina.authenticator.BasicAuthenticator"/>
```

If you are using a WAR file, `context.xml` should be in `/META-INF` for Tomcat and in `/WEB-INF` for JBoss Application Server.

5.1.4 Step 4: Start the Cluster

1. Start the Terracotta server:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat
```

2. Start the application servers.
3. Start the Terracotta Developer Console:

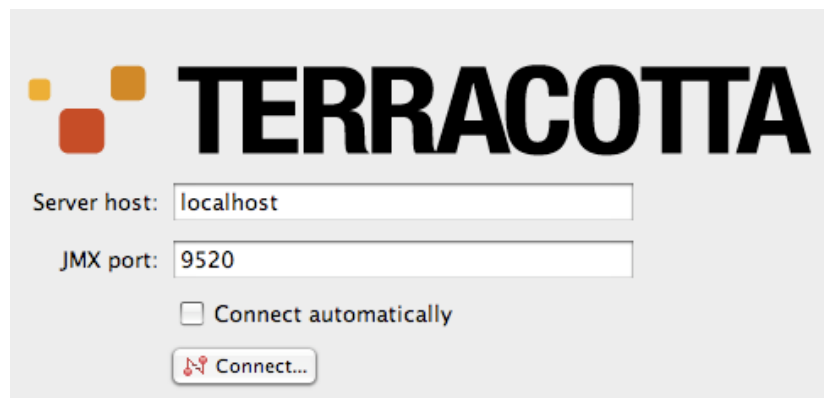
UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

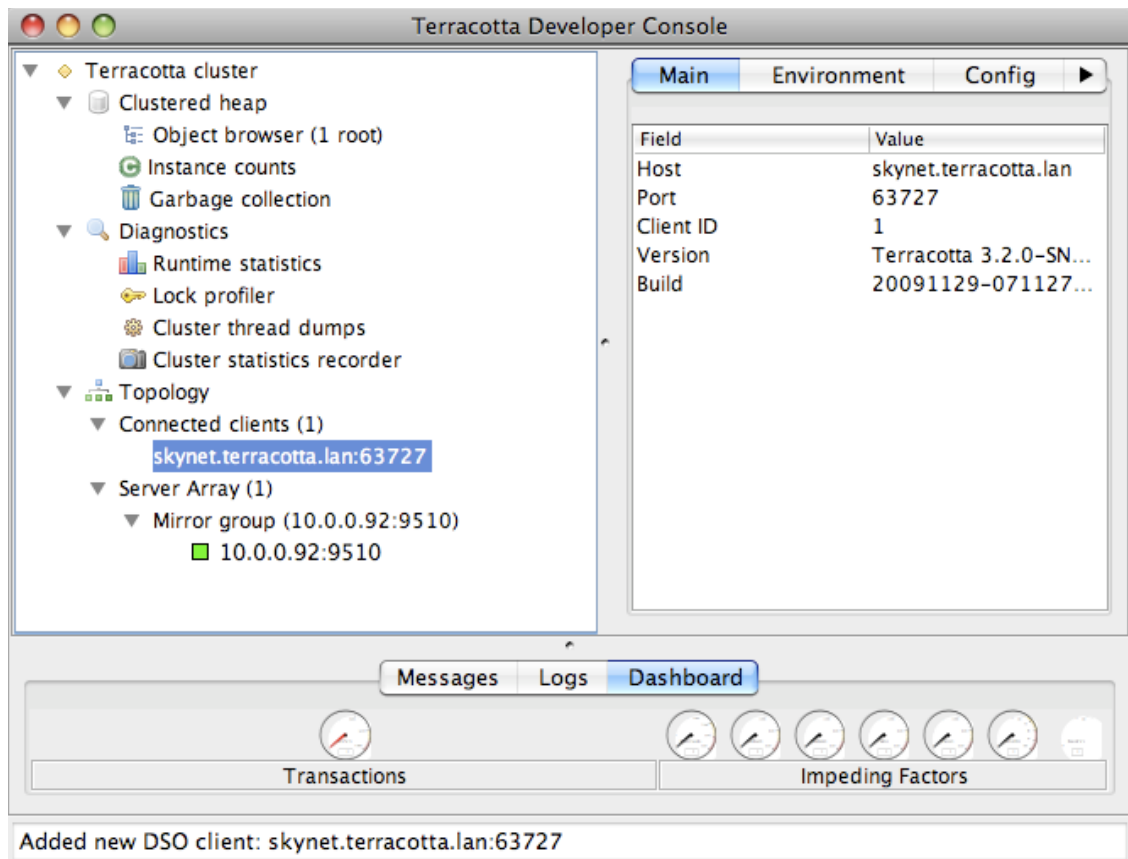
Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

4. Connect to the Terracotta cluster.
Click **Connect...** in the Terracotta Developer Console.



5. Click the **Topology** node in the cluster navigation window to see the Terracotta servers and clients (application servers) in the Terracotta cluster.
Your console should have a similar appearance to the following figure.



5.1.5 Step 5: Edit the Terracotta Configuration

This step shows you how to run clients and servers on separate machines and add failover (High Availability). You will expand the Terracotta cluster and add High Availability by doing the following:

- Moving the Terracotta server to its own machine
- Creating a cluster with multiple Terracotta servers
- Creating multiple application nodes

These tasks bring your cluster closer to a production architecture.

5.1.5a Procedure:

1. Shut down the Terracotta cluster.
2. Create a Terracotta configuration file called `tc-config.xml` with contents similar to the following

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- All content copyright Terracotta, Inc., unless otherwise indicated.      All rights reserved. -->
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd"
xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <servers>
    <!-- Sets where the Terracotta server can be found. Replace the value of      host with the
server's IP address. -->
    <server host="server.1.ip.address" name="Server1">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
    <!-- If using a standby Terracotta server, also referred to as an ACTIVE-PASSIVE configuration, add
the second server here. -->
    <server host="server.2.ip.address" name="Server2">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
  </servers>
  <!-- If using more than one server, add an <ha> section. -->
  <ha>
    <mode>networked-active-passive</mode>
  </ha>
</tc:tc-config>
```

```

    <networked-active-passive>
      <election-time>5</election-time>
    </networked-active-passive>
  </ha>
</servers>
<!-- Sets where the generated client logs are saved on clients. -->
<clients>
  <logs>%(user.home)/terracotta/client-logs</logs>
</clients>
</tc:tc-config>

```

3. Install Terracotta 3.5.0 on a separate machine for each server you configure in `tc-config.xml`.
4. Copy the `tc-config.xml` to a location accessible to the Terracotta servers.
5. Perform [Step 2: Install the Terracotta Sessions JAR](#) on each application node you want to run in the cluster.
Be sure to install your application and any application servers on each node.
6. Edit `web.xml` or `context.xml` on each application server to list both Terracotta servers:

```
<param-value>server.1.ip.address:9510,server.2.ip.address:9510</param-value>
```

or

```
tcConfigUrl="server.1.ip.address:9510,server.2.ip.address:9510"
```

7. Start the Terracotta server in the following way, replacing "Server1" with the name you gave your server in `tc-config.xml`:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh -f <path/to/tc-config.xml> -n Server1 &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat -f <path\to\tc-config.xml> -n Server1 &
```

If you configured a second server, start that server in the same way on its machine, entering its name after the `-n` flag. The second server to start up becomes the "hot" standby, or PASSIVE. Any other servers you configured will also start up as standby servers.

8. Start all application servers.
9. Start the Terracotta Developer Console and view the cluster.

5.1.6 Step 6: Learn More

To learn more about working with a Terracotta cluster, see the following documents:

- [6.1 Working with Terracotta Configuration Files](#) - Explains how `tc-config.xml` is propagated and loaded in a Terracotta cluster in different environments.
- [6.3 Terracotta Server Arrays](#) - Shows how to design Terracotta clusters that are fault-tolerant, maintain data safety, and provide uninterrupted uptime.
- [6.2 Configuring Terracotta Clusters For High Availability](#) - Defines High Availability configuration properties and explains how to apply them.
- [8.1 Terracotta Developer Console](#) - Provides visibility into and control of caches.

5.2 Web Sessions Reference

This section contains further information on configuring and troubleshooting Terracotta Web Sessions.

5.2.1 Additional Configuration Options

While Terracotta Web Sessions is designed for optimum performance with the configuration you set at installation, in some cases it may be necessary to use the configuration options described in the following sections.

5.2.1a Session Locking

By default, session locking is off in standard (non-DSO) Terracotta Web Sessions. If your application requires disabling concurrent requests in sessions, you can enable session locking.

To enable session locking in filter-based configuration, add an `<init-param>` block as follows:

```

<filter>
  <filter-name>terracotta-filter</filter-name>
  <filter-class>org.terracotta.session.TerracottaContainerSpecificSessionFilter</filter-class>

```

```

<init-param>
  <param-name>tcConfigUrl</param-name>
  <param-value>localhost:9510</param-value>
</init-param>
<init-param>
  <param-name>sessionLocking</param-name>
  <param-value>true</param-value>
</init-param>
</filter>

```

To enable session locking in Valve-based configuration, add a `sessionLocking` attribute as follows:

```

<Valve className="com.terracotta.TerracottaContainerSpecificSessionValve" tcConfigUrl="localhost:9510"
sessionLocking="true"/>

```

If you enable session locking, see [5.2.2d Deadlocks When Session Locking Is Enabled](#).

5.2.1b Synchronous Writes

Synchronous write locks provide an extra layer of data protection by having a client node wait until it receives acknowledgement from the Terracotta Server Array that the changes have been committed. The client releases the write lock after receiving the acknowledgement. Enabling synchronous write locks *can substantially raise latency rates, thus degrading cluster performance*.

To enable synchronous writes in filter-based configuration, add an `<init-param>` block as follows:

```

<filter>
  <filter-name>terracotta-filter</filter-name>
  <filter-class>org.terracotta.session.TerracottaContainerSpecificSessionFilter</filter-class>
  <init-param>
    <param-name>tcConfigUrl</param-name>
    <param-value>localhost:9510</param-value>
  </init-param>
  <init-param>
    <param-name>synchronousWrite</param-name>
    <param-value>true</param-value>
  </init-param>
</filter>

```

To enable synchronous writes in Valve-based configuration, add a `synchronousWrite` attribute as follows:

```

<Valve className="com.terracotta.TerracottaContainerSpecificSessionValve" tcConfigUrl="localhost:9510"
synchronousWrite="true"/>

```

5.2.2 Troubleshooting

The following sections summarize common issues than can be encountered when clustering web sessions.

5.2.2a Sessions Time Out Unexpectedly

Sessions that are set to expire after a certain time instead seem to expire at unexpected times, and sooner than expected. This problem can occur when sessions hop between nodes that do not have the same system time. A node that receives a request for a session that originated on a different node still checks local time to validate the session, not the time on the original node. Adding the Network Time Protocol (NTP) to all nodes can help avoid system-time drift. However, note that having nodes set to different time zones can cause this problem, even with NTP.

This problem can also cause sessions to time out later than expected, although this variation can have many other causes.

5.2.2b Changes Not Replicated

Terracotta Web Sessions must run in serialization mode. Serialization mode is not an option as it is in the DSI version of Web Sessions. In serialization mode, sessions are still clustered, but your application must now follow the standard servlet convention on using `setAttribute()` for mutable objects in replicated sessions.

5.2.2c Tomcat 5.5 Messages Appear With Tomcat 6 Installation

If you are running Tomcat 6, you may see references to Tomcat 5.5 in the Terracotta log. This occurs because Terracotta Web Sessions run with Tomcat 6 reuses some classes from the Tomcat 5.5 Terracotta Integration Module.

5.2.2d Deadlocks When Session Locking Is Enabled

In some containers or frameworks, it is possible to see deadlocks when session locking is in effect. This happens when an *external* request is made from inside the locked session to access that same session. This type of request fails because the session is locked.

5.2.2e Events Not Received on Node

Most Servlet spec-defined events will work with Terracotta clustering, but the events are generated on the node where they occur. For example, if a session is created on one node and destroyed on a second node, the event is received on the second node, not the first node.

6 The Terracotta Server Array

The Terracotta Server Array provides the platform for Terracotta products and the backbone for Terracotta clusters. Terracotta Server Array documentation describes how to:

- Use Terracotta configuration files
- Set up High Availability
- Work with different cluster architectures to meet failover, persistence, and scaling needs
- Improve performance using BigMemory
- Add cluster security
- Manage cluster topology

Start with learning about Terracotta configuration files and setting up High Availability, then see the architecture section to find a cluster setup that meets your needs. Once you have a test cluster running, add BigMemory and measure performance improvement.

6.1 Working with Terracotta Configuration Files

Terracotta XML configuration files set the characteristics and behavior of Terracotta server instances and Terracotta clients. The easiest way to create your own Terracotta configuration file is by editing a copy of one of the sample configuration files available with the Terracotta kit.

Where you locate the Terracotta configuration file, or how your Terracotta server and client configurations are loaded, depends on the stage your project is at and on its architecture. This document covers the following cases:

- Development stage, 1 Terracotta server
- Development stage, 2 Terracotta servers
- Deployment stage

This document discusses cluster configuration in the Terracotta Server Array. To learn more about the Terracotta server instances, see [Terracotta Server Arrays](#).

For a comprehensive and fully annotated configuration file, see `config-samples/tc-config-express-reference.xml` in the Terracotta kit.

6.1.1 How Terracotta Servers Get Configured

At startup, Terracotta servers load their configuration from one of the following sources:

- A default configuration included with the Terracotta kit
- A local or remote XML file

These sources are explored below.

6.1.1a Default Configuration

If no configuration file is specified *and* no `tc-config.xml` exists in the directory in which the Terracotta instance is started, then default configuration values are used.

6.1.1b Local XML File (Default)

The file `tc-config.xml` is used by default if it is located in the directory in which a Terracotta instance is started *and* no configuration file is explicitly specified.

6.1.1c Local or Remote Configuration File

You can explicitly specify a configuration file by passing the `-f` option to the script used to start a Terracotta server. For example, to start a Terracotta server on UNIX/Linux using the provided script, enter:

```
start-tc-server.sh -f <path_to_configuration_file>
```

where `<path_to_configuration_file>` can be a URL or a relative directory path. In Microsoft Windows, use `start-tc-server.bat`.

6.1.2 How Terracotta Clients Get Configured

At startup, Terracotta clients load their configuration from one of the following sources:

- [Local or Remote XML File](#)
- [Terracotta Server](#)
- An Ehcache configuration file (using the `<terracottaConfig>` element) used with Enterprise Ehcache and Enterprise Ehcache for Hibernate.
- A Quartz properties file (using the `org.quartz.jobStore.tcConfigUrl` property) used with Quartz Scheduler.
- A Filter (in `web.xml`) or Valve (in `context.xml`) elements used with containers and Terracotta Sessions.
- The client constructor (`TerracottaClient()`) used when a client is instantiated programmatically using the Terracotta Toolkit.

Terracotta clients can load customized configuration files to specify `<client>` and `<application>` configuration. However, the `<servers>` block of every client in a cluster must match the `<servers>` block of the servers in the cluster. If there is a mismatch, the client will emit an error and fail to complete its startup

NOTE: Error with Matching Configuration Files

On startup, a Terracotta client may emit a configuration-mismatch error if its `<servers>` block does not match that of the server it connects to. However, under certain circumstances, this error may occur even if the `<servers>` blocks appear to match.

The following suggestions may help prevent this error:

- Use `-Djava.net.preferIPv4Stack` consistently. If it is explicitly set on the client, be sure to explicitly set it on the server.
- Ensure `etc/hosts` file does not contain multiple entries for hosts running Terracotta servers.
- Ensure that DNS always returns the same address for hosts running Terracotta servers.

6.1.2a Local or Remote XML File

See the discussion for local XML file (default) in [How Terracotta Servers Get Configured](#).

To specify a configuration file for a Terracotta client, see [Clients in Development](#).

NOTE: Fetching Configuration from the Server

On startup, Terracotta clients must fetch certain configuration properties from a Terracotta server. A client loading its own configuration will attempt to connect to the Terracotta servers named in that configuration. If none of the servers named in that configuration are available, the client cannot complete its startup.

6.1.2b Terracotta Server

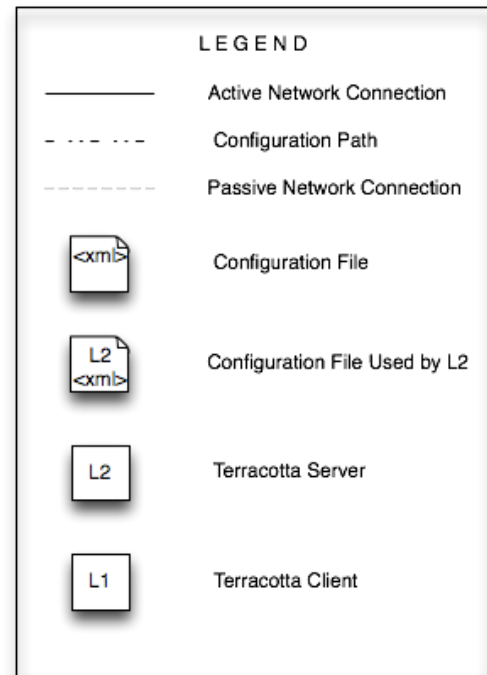
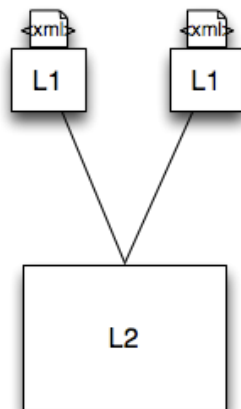
Terracotta clients can load configuration from a running Terracotta server by specifying its hostname and DSO port (see [Clients in Production](#)).

6.1.3 Configuration in a Development Environment

In a development environment, using a different configuration file for each Terracotta client facilitates the testing and tuning of configuration options. This is an efficient and effective way to gain valuable insight on best practices for clustering your application with Terracotta DSO.

6.1.3a One-Server Setup in Development

For one Terracotta server, the default configuration is adequate.

Development Environment

To use the default configuration settings, start your Terracotta server using the `start-tc-server.sh` (or `start-tc-server.bat`) script in a directory that does *not* contain the file `tc-config.xml` :

[PROMPT] `${TERRACOTTA_HOME}\bin\start-tc-server.sh`

To specify a configuration file, use one of the approaches discussed in [How Terracotta Servers Get Configured](#).

6.1.3b Two-Server Setup in Development

A two-server setup, sometimes referred to as an active-passive setup, has one active server instance and one "hot standby" (the passive, or backup) that should load the same configuration file.

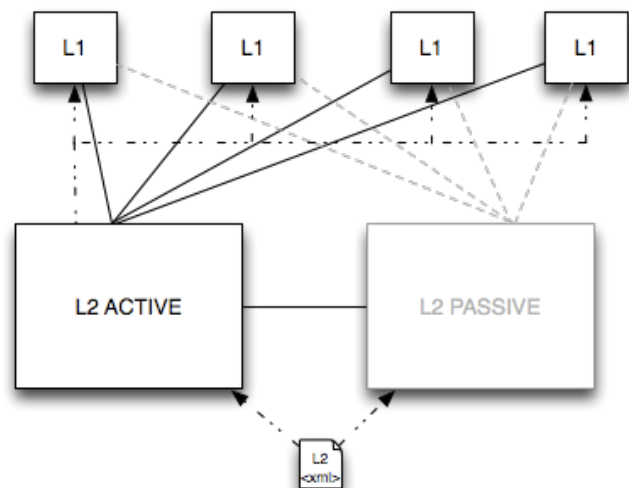
The configuration file loaded by the Terracotta servers must define each server separately using `<server>` elements. For example:

```
<tc:tc-config
xsi:schemaLocation="http://www.terracotta.org
/schema/terracotta-5.xsd"
xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
...
<!-- Use an IP address or a resolvable host
name for the host attribute. -->
  <server host="123.456.7.890" name="Server1">
...
  <server host="myResolvableHostName"
name="Server2">
...
</tc:tc-config>
```

Assuming Server1 is the active server, using the same configuration allows Server2 to be the hot standby and maintain the environment in case of failover. If you are running both Terracotta servers on the same host, the only port that has to be specified in configuration is the `<dso-port>`; the values for `<jmx-port>` and `<l2-group-port>` are filled in automatically.

NOTE: Running Two Servers on the Same Host

If you are running the servers on the same machine, some elements in the `<server>` section, such as `<dso-port>` and `<server-logs>`, must have different values for each server.

Production Environment

Server Names for Startup

With multiple <server> elements, the name attribute may be required to avoid ambiguity when starting a server:

```
start-tc-server.sh -n Server1 -f <path_to_configuration_file>
```

In Microsoft Windows, use `start-tc-server.bat`.

For example, if you are running Terracotta server instances on the same host, you must specify the name attribute to set an unambiguous target for the script.

However, if you are starting Terracotta server instances in an unambiguous setup, specifying the server name is optional. For example, if the Terracotta configuration file specifies different IP addresses for each server, the script assumes that the server with the IP address corresponding to the local IP address is the target.

6.1.3c Clients in Development

You can explicitly specify a client's Terracotta configuration file by passing `-Dtc.config=path/to/my-tc-config.xml` when you start your application with the Terracotta client.

DSO users can use the `dso-java.sh` script (or `dso-java.bat` for Windows):

```
dso-java.sh -Dtc.config=path/to/my-tc-config.xml -cp classes myApp.class.Main
```

where `myApp.class.Main` is the class used to launch the application you want to cluster with Terracotta. In Microsoft Windows, use `dso-java.bat`.

If `tc-config.xml` exists in the directory in which you run `dso-java`, it can be loaded without `-Dtc.config`.

TIP: Avoiding the dso-java Script

If you do not require DSO, do not use the `dso-java` script. Terracotta products provide simple configuration setups.

If you are using DSO, you may want to avoid using the `dso-java` script and start your application with the Terracotta client using your own scripts. See [Setting Up the Terracotta Environment](#).

6.1.4 Configuration in a Production Environment

For an efficient production environment, it's recommended that you maintain one Terracotta configuration file. That file can be loaded by the Terracotta server (or servers) and pushed out to clients. While this is an optional approach, it's an effective way to centralize and decrease maintenance.

If your Terracotta configuration file uses `%i` for the hostname attribute in its server element, change it to the actual hostname in production. For example, if in development you used the following:

```
<server host="%i" name="Server1">
```

and the production host's hostname is `myHostName`, then change the host attribute to the `myHostName`:

```
<server host="myHostName" name="Server1">
```

6.1.4a Clients in Production

For clients in production, you can set up the Terracotta environment before launching your application. DSO users can use the `dso-java` script.

Setting Up the Terracotta Environment

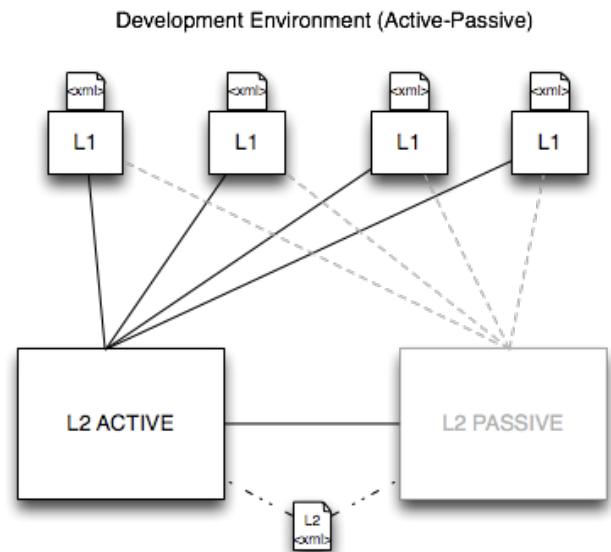
To start your application with the Terracotta client using your own scripts, first set the following environment variables:

```
TC_INSTALL_DIR=<path_to_local_Terracotta_home>
```

```
TC_CONFIG_PATH=<path/to/tc-config.xml>
```

or

```
TC_CONFIG_PATH=<server_host>:<dso-port>
```



where `<server_host>`:`<dso-port>` points to the running Terracotta server. The specified Terracotta server will push its configuration to the Terracotta client.

If more than one Terracotta server is available, enter them in a comma-separated list:

`TC_CONFIG_PATH=<server_host1>:<dso-port>,<server_host2>:<dso-port>`

If `<server_host1>` is unavailable, `<server_host2>` is used.

If using DSO, complete setting up the Terracotta client's environment by running the following:

UNIX/Linux

```
[PROMPT] ${TC_INSTALL_DIR}/platform/bin/dso-env.sh -q
```

```
[PROMPT] export JAVA_OPTS="$TC_JAVA_OPTS $JAVA_OPTS"
```

Microsoft Windows

```
[PROMPT] %TC_INSTALL_DIR%\bin\dso-env.bat -q
```

```
[PROMPT] set JAVA_OPTS=%TC_JAVA_OPTS;%JAVA_OPTS%
```

Before starting up your application, confirm that the value of `JAVA_OPTS` is correct.

Terracotta Products

Terracotta products without DSO (also called the "express" installation) can set a configuration path using their own configuration files.

For Enterprise Ehcache and Enterprise Ehcache for Hibernate, use the `<terracottaConfig>` element in the Ehcache configuration file (`ehcache.xml` by default):

```
<terracottaConfig url="localhost:9510" />
```

For Quartz, use the `org.quartz.jobStore.tcConfigUrl` property in the Quartz properties file (`quartz.properties` by default):

```
org.quartz.jobStore.tcConfigUrl = /myPath/to/tc-config.xml
```

For Terracotta Web Sessions, use the appropriate elements in `web.xml` or `context.xml` (see [Web Sessions Installation](#)).

6.1.5 Binding Ports to Interfaces

Normally, the ports you specify for a server in the Terracotta configuration are bound to the interface associated with the host specified for that server. For example, if the server is configured with the IP address "12.345.678.8" (or a hostname with that address), the server's ports are bound to that same interface:

```
<server host="12.345.678.8" name="Server1">
  ...
  <dso-port>9510</dso-port>
  <jmx-port>9520</jmx-port>
  <l2-group-port>9530</l2-group-port>
</server>
```

However, in certain situations it may be necessary to specify a different interface for one or more of a server's ports. This is done using the `bind` attribute, which allows you bind a port to a different interface. For example, a JMX client may only be able connect to a certain interface on a host. The following configuration shows a JMX port bound to an interface different than the host's:

```
<server host="12.345.678.8" name="Server1">
  ...
  <dso-port>9510</dso-port>
  <jmx-port bind="12.345.678.9">9520</jmx-port>
  <l2-group-port>9530</l2-group-port>
</server>
```

6.1.6 terracotta.xml (DSO only)

The file `terracotta.xml`, which contains a fragment of Terracotta configuration, can be part of a Terracotta integration module (TIM). When the TIM is loaded at runtime, `terracotta.xml` is inserted into the client configuration. For more information on TIMs and `terracotta.xml`, see the [Terracotta Integration Modules Manual](#).

NOTE: Element Hierarchy

If the use of `terracotta.xml` introduces duplicate elements into the client configuration, the value of the last element parsed is used. The last element parsed appears last in order in the configuration file.

6.1.7 Which Configuration?

Each server and client must maintain separate log directories. By default, server logs are written to `%(user.home)/terracotta/server-logs` and client logs to `%(user.home)/terracotta/client-logs`.

To find out which configuration a server or client is using, search its logs for an INFO message containing the text "Configuration loaded from".

6.2 Configuring Terracotta Clusters For High Availability

High Availability (HA) is an implementation designed to maintain uptime and access to services even during component overloads and failures. Terracotta clusters offer simple and scalable HA implementations based on the Terracotta Server Array (see [Terracotta Server Arrays](#) for more information).

The main features of a Terracotta HA architecture include:

- Instant failover using a hot standby or multiple active servers - provides continuous uptime and service
- Configurable automatic internode monitoring - Terracotta [HealthChecker](#)
- Automatic permanent storage of all current shared (in-memory) data - available to all server instance (no loss of application state)
- Automatic reconnection of temporarily disconnected server instances and clients - restores hot standbys without operator intervention, allows "lost" clients to reconnect

TIP: Nomenclature

This document may refer to a Terracotta server instance as L2, and a Terracotta client (the node running your application) as L1. These are the shorthand references used in Terracotta configuration files.

To learn about reconnecting Enterprise Ehcache clients that have been disconnected from their cluster, see [2.3.7a Using Rejoin to Automatically Reconnect Terracotta Clients](#).

It is important to thoroughly test any High Availability setup before going to production. Suggestions for testing High Availability configurations are provided in [this document](#).

6.2.1 Common Causes of Failures in a Cluster

Failures in a cluster include L1s being ejected, standby L2s becoming active and attempting to take over the cluster while the original active L2 is still functional, long pauses in cluster operations, and even complete cluster failure.

The most common causes of failures in a cluster are interruptions in the network and long Java GC cycles on particular nodes. Tuning the HealthChecker and reconnect features can reduce or eliminate these two problems. However, additional actions should also be considered.

Sporadic disruptions in network connections between L2s and between L2s and L1s can be difficult to track down. Be sure to thoroughly test all network segments connecting the nodes in a cluster, and also test network hardware. Check for speed, noise, reliability, and other applications that grab bandwidth.

Long GC cycles can also be helped by the following:

- Tuning Java GC to work more efficiently with the clustered application.
- Refactoring clustered applications that unnecessarily create too much garbage can also help.
- Ensuring that the problem node has enough memory allocated to the heap.

TIP: Using BigMemory to Alleviate GC Slowdowns

Terracotta BigMemory allows L2s to use memory outside of the Java object heap. See [Improving Server Performance With BigMemory](#).

Two other sources of failures in a cluster are disks that are nearly full or are running slowly, and other applications that compete for a node's resources.

6.2.2 Basic High-Availability Configuration

A basic high-availability configuration has the following components:

- Two or More Terracotta Server Instances
See [Terracotta Server Arrays](#) on how to set up a cluster with multiple Terracotta server instances.
- Active-Passive Mode
The `<ha>` section in the Terracotta configuration file should indicate the mode as networked-active-passive to allow for an active server instance and one or more "hot standby" (backup) server instances

The <networked-active-passive> subsection has a configurable parameter called <election-time> whose value is given in seconds. <election-time>, which sets the duration for elections to elect an active server, is a factor in network latency and server load. The default value is 5 seconds:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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-5.xsd">
  <servers>
    ...
    <ha>
      <mode>networked-active-passive</mode>
      <networked-active-passive>
        <election-time>5</election-time>
      </networked-active-passive>
    </ha>
  </servers>
  ...
</tc:tc-config>
```

NOTE: Servers Should Not Share Data Directories

When using networked-active-passive mode, Terracotta server instances must not share data directories. Each server's <data> element should point to a different and preferably local data directory.

- **Server-Server Reconnection**
A reconnection mechanism can be enabled to restore lost connections between active and passive Terracotta server instances. See [Automatic Server Instance Reconnect](#) for more information.
- **Server-Client Reconnection**
A reconnection mechanism can be enabled to restore lost connections between Terracotta clients and server instances. See [Automatic Client Reconnect](#) for more information.

For more information on Terracotta configuration files, see:

- [Working with Terracotta Configuration Files](#)
- [Configuration Guide and Reference](#) (*Servers Configuration Section*)

6.2.3 High-Availability Features

The following high-availability features can be used to extend the reliability of a Terracotta cluster. These features are controlled using properties set with the <tc-properties> section at the *beginning* of the Terracotta configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- All content copyright Terracotta, Inc., unless otherwise indicated. All rights reserved. -->
<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-5.xsd">

  <tc-properties>
    <property name="some.property.name" value="true"/>
    <property name="some.other.property.name" value="true"/>
    <property name="still.another.property.name" value="1024"/>
  </tc-properties>

  <!-- The rest of the Terracotta configuration goes here. -->

</tc:tc-config>
```

See the section *Overriding tc.properties* in [Configuration Guide and Reference](#) for more information.

6.2.3a HealthChecker

HealthChecker is a connection monitor similar to TCP keep-alive. HealthChecker functions between Terracotta server instances (in High Availability environments), and between Terracotta sever instances and clients. Using HealthChecker, Terracotta nodes can determine if peer nodes are reachable, up, or in a GC operation. If a peer node is unreachable or down, a Terracotta node using HealthChecker can take corrective action. HealthChecker is on by default.

You configure HealthChecker using certain Terracotta properties, which are grouped into three different categories:

- Terracotta server instance -> Terracotta client
- Terracotta Server -> Terracotta Server (HA setup only)

- Terracotta Client -> Terracotta Server

Property category is indicated by the prefix:

- `l2.healthcheck.l1` indicates L2 -> L1
- `l2.healthcheck.l2` indicates L2 -> L2
- `l1.healthcheck.l2` indicates L1 -> L2

For example, the `l2.healthcheck.l2.ping.enabled` property applies to L2 -> L2.

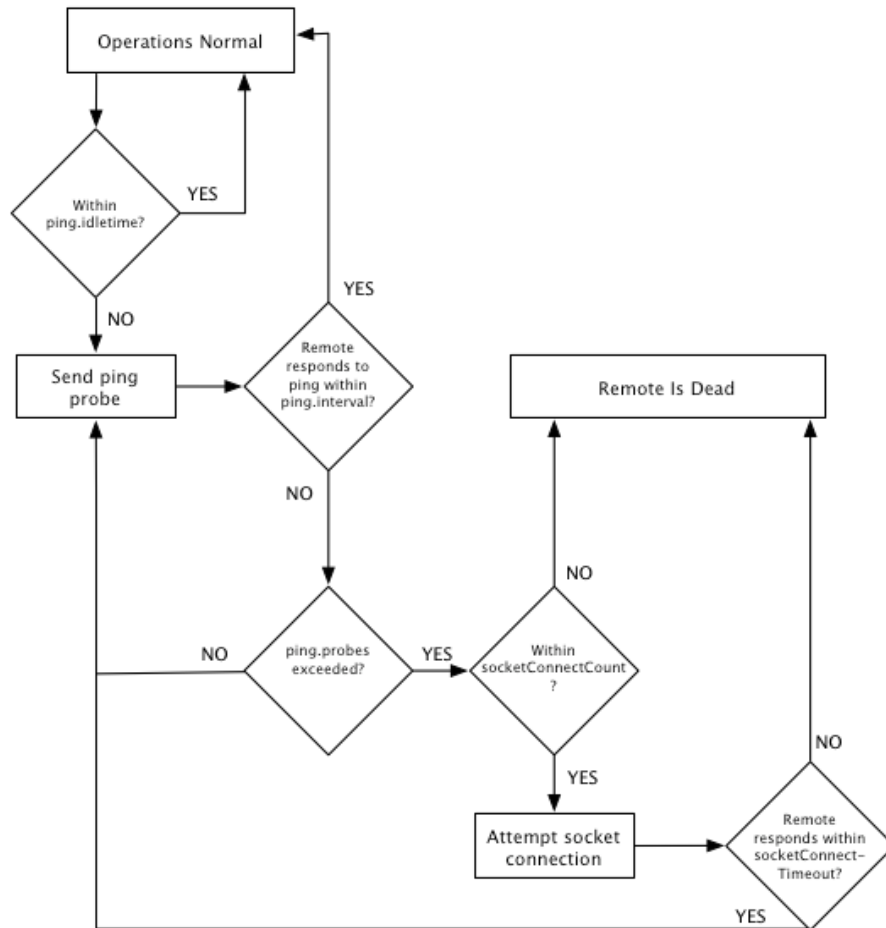
The following HealthChecker properties can be set in the <tc-properties> section of the Terracotta configuration file:

Property	Definition
<code>l2.healthcheck.l1.ping.enabled</code> <code>l2.healthcheck.l2.ping.enabled</code> <code>l1.healthcheck.l2.ping.enabled</code>	Enables (True) or disables (False) ping probes (tests). Ping probes are high-level attempts to gauge the ability of a remote node to respond to requests and is useful for determining if temporary inactivity or problems are responsible for the node's silence. Ping probes may fail due to long GC cycles on the remote node.
<code>l2.healthcheck.l1.ping.idleTime</code> <code>l2.healthcheck.l2.ping.idleTime</code> <code>l1.healthcheck.l2.ping.idleTime</code>	The maximum time (in milliseconds) that a node can be silent (have no network traffic) before HealthChecker begins a ping probe to determine if the node is alive.
<code>l2.healthcheck.l1.ping.interval</code> <code>l2.healthcheck.l2.ping.interval</code> <code>l1.healthcheck.l2.ping.interval</code>	If no response is received to a ping probe, the time (in milliseconds) that HealthChecker waits between retries.
<code>l2.healthcheck.l1.ping.probes</code> <code>l2.healthcheck.l2.ping.probes</code> <code>l1.healthcheck.l2.ping.probes</code>	If no response is received to a ping probe, the maximum number (integer) of retries HealthChecker can attempt.
<code>l2.healthcheck.l1.socketConnect</code> <code>l2.healthcheck.l2.socketConnect</code> <code>l1.healthcheck.l2.socketConnect</code>	Enables (True) or disables (False) socket-connection tests. This is a low-level connection that determines if the remote node is reachable and can access the network. Socket connections are not affected by GC cycles.
<code>l2.healthcheck.l1.socketConnectTimeout</code> <code>l2.healthcheck.l2.socketConnectTimeout</code> <code>l1.healthcheck.l2.socketConnectTimeout</code>	A multiplier (integer) to determine the maximum amount of time that a remote node has to respond before HealthChecker concludes that the node is dead (regardless of previous successful socket connections). The time is determined by multiplying the value in <code>ping.interval</code> by this value.
<code>l2.healthcheck.l1.socketConnectCount</code> <code>l2.healthcheck.l2.socketConnectCount</code> <code>l1.healthcheck.l2.socketConnectCount</code>	The maximum number (integer) of successful socket connections that can be made without a successful ping probe. If this limit is exceeded, HealthChecker concludes that the target node is dead.
<code>l1.healthcheck.l2.bindAddress</code>	Binds the client to the configured IP address. This is useful where a host has more than one IP address available for a client to use. The default value of "0.0.0.0" allows the system to assign an IP address.

```
11.healthcheck.12.bindPort
```

Set the client's callback port. Terracotta configuration does not assign clients a port for listening to cluster communications such as that required by HealthChecker. The default value of "0" allows the system to assign a port. A value of "-1" disables a client's callback port.

The following diagram illustrates how HealthChecker functions.



Calculating HealthChecker Maximum

The following formula can help you compute the maximum time it will take HealthChecker to discover failed or disconnected remote nodes:

Max Time = (ping.idletime) + socketConnectCount * [(ping.interval * ping.probes) + (socketConnectTimeout * ping.interval)]

Note the following about the formula:

- The response time to a socket-connection attempt is less than or equal to (socketConnectTimeout * ping.interval). For calculating the worst-case scenario (absolute maximum time), the equality is used. In most real-world situations the socket-connect response time is likely to be close to 0 and the formula can be simplified to the following:
Max Time = (ping.idletime) + [socketConnectCount * (ping.interval * ping.probes)]
- ping.idletime, the trigger for the full HealthChecker process, is counted once since it is in effect only once each time the process is triggered.
- socketConnectCount is a multiplier that is incremented as long as a positive response is received for each socket connection attempt.
- The formula yields an ideal value, since slight variations in actual times can occur.

Configuration Examples

The configuration examples in this section show settings for L1 -> L2 HealthChecker. However, they apply in the similarly to L2 -> L2 and L2 -> L1, which means that the server is using HealthChecker on the client.

Aggressive

The following settings create an aggressive HealthChecker with low tolerance for short network outages or long GC cycles:

```
<property name="l1.healthcheck.l2.ping.enabled" value="true" />
<property name="l1.healthcheck.l2.ping.idletime" value="2000" />
<property name="l1.healthcheck.l2.ping.interval" value="1000" />
<property name="l1.healthcheck.l2.ping.probes" value="3" />
<property name="l1.healthcheck.l2.socketConnect" value="true" />
<property name="l1.healthcheck.l2.socketConnectTimeout" value="2" />
<property name="l1.healthcheck.l2.socketConnectCount" value="5" />
```

According to the HealthChecker "Max Time" formula, the maximum time before a remote node is considered to be lost is computed in the following way:

$$2000 + 5 [(3 * 1000) + (2 * 1000)] = 27000$$

In this case, after the initial idletime of 2 seconds, the remote failed to respond to ping probes but responded to every socket connection attempt, indicating that the node is reachable but not functional (within the allowed time frame) or in a long GC cycle. This aggressive HealthChecker configuration declares node dead in no more than 27 seconds.

If at some point the node had been completely unreachable (a socket connection attempt failed), HealthChecker would have declared it dead sooner. Where, for example, the problem is a disconnected network cable, the "Max Time" is likely to be even shorter:

$$2000 + 1[3 * 1000] + (2 * 1000) = 7000$$

In this case, HealthChecker declares a node dead in no more than 7 seconds.

Tolerant

The following settings create a HealthChecker with a higher tolerance for interruptions in network communications and long GC cycles:

```
<property name="l1.healthcheck.l2.ping.enabled" value="true" />
<property name="l1.healthcheck.l2.ping.idletime" value="5000" />
<property name="l1.healthcheck.l2.ping.interval" value="1000" />
<property name="l1.healthcheck.l2.ping.probes" value="3" />
<property name="l1.healthcheck.l2.socketConnect" value="true" />
<property name="l1.healthcheck.l2.socketConnectTimeout" value="5" />
<property name="l1.healthcheck.l2.socketConnectCount" value="10" />
```

According to the HealthChecker "Max Time" formula, the maximum time before a remote node is considered to be lost is computed in the following way:

$$5000 + 10 [(3 * 1000) + (5 * 1000)] = 85000$$

In this case, after the initial idletime of 5 seconds, the remote failed to respond to ping probes but responded to every socket connection attempt, indicating that the node is reachable but not functional (within the allowed time frame) or excessively long GC cycle. This tolerant HealthChecker configuration declares a node dead in no more than 85 seconds.

If at some point the node had been completely unreachable (a socket connection attempt failed), HealthChecker would have declared it dead sooner. Where, for example, the problem is a disconnected network cable, the "Max Time" is likely to be even shorter:

$$5000 + 1[3 * 1000] + (5 * 1000) = 13000$$

In this case, HealthChecker declares a node dead in no more than 13 seconds.

6.2.3b Automatic Server Instance Reconnect

An automatic reconnect mechanism can prevent short network disruptions from forcing a restart for any Terracotta server instances in a server array with hot standbys. If not disabled, this mechanism is by default in effect in clusters set to networked-based HA mode.

NOTE: Increased Time-to-Failover

This feature increases time-to-failover by the timeout value set for the automatic reconnect mechanism.

This event-based reconnection mechanism works independently and exclusively of HealthChecker. If HealthChecker has already been triggered, this mechanism cannot be triggered for the same node. If this mechanism is triggered first by an internal Terracotta event, HealthChecker is prevented from being triggered for the same node. The events that can trigger this mechanism are not exposed by API but are logged.

Configure the following properties for the reconnect mechanism:

- `12.nha.tcgroupcomm.reconnect.enabled` - When set to "true" enables a server instance to attempt reconnection with its peer server instance after a disconnection is detected.
- `12.nha.tcgroupcomm.reconnect.timeout` - Enabled if `12.nha.tcgroupcomm.reconnect.enabled` is set to true. Specifies the timeout (in milliseconds) for reconnection. Default: 2000. This parameter can be tuned to handle longer network disruptions.

6.2.3c Automatic Client Reconnect

Clients disconnected from a Terracotta cluster normally require a restart to rejoin the cluster. An automatic reconnect mechanism can prevent short network disruptions from forcing a restart for Terracotta clients disconnected from a Terracotta cluster.

NOTE: Performance Impact of Using Automatic Client Reconnect

With this feature, clients waiting to reconnect continue to hold locks. Some application threads may block while waiting to for the client to reconnect.

This event-based reconnection mechanism works independently and exclusively of HealthChecker. If HealthChecker has already been triggered, this mechanism cannot be triggered for the same node. If this mechanism is triggered first by an internal Terracotta event, HealthChecker is prevented from being triggered for the same node. The events that can trigger this mechanism are not exposed by API but are logged.

Configure the following properties for the reconnect mechanism:

- `12.11reconnect.enabled` - When set to "true" enables a client to reconnect to a cluster after a disconnection is detected. This property controls a server instance's reaction to such an attempt. It is set on the server instance and is passed to clients by the server instance. A client cannot override the server instance's setting. If a mismatch exists between the client setting and a server instance's setting and the client attempts to reconnect to the cluster, the client emits a mismatch error and exits.
- `12.11reconnect.timeout.millis` - Enabled if `12.11reconnect.enabled` is set to true. Specifies the timeout (in milliseconds) for reconnection. This property controls a server instance's timeout during such an attempt. It is set on the server instance and is passed to clients by the server instance. A client cannot override the server instance's setting. Default: 2000. This parameter can be tuned to handle longer network disruptions.

6.2.3d Special Client Connection Properties

Client connections can also be tuned for the following special cases:

- Client failover after server failure
- First-time client connection

The connection properties associated with these cases are already optimized for most typical environments. If you attempt to tune these properties, be sure to thoroughly test the new settings.

Client Failover After Server Failure

When an active Terracotta server instance fails, and a "hot" standby Terracotta server is available, the formerly "passive" server becomes active. Terracotta clients connected to the previous active server automatically switch to the new active server. However, these clients have a limited window of time to complete the failover.

TIP: Clusters with a Single Server

This reconnection window also applies in a cluster with a single Terracotta server that is restarted. However, a single-server cluster must have its `<persistence>` element's `<mode>` subelement set to "permanent-store" for the reconnection window to take effect.

This window is configured in the Terracotta configuration file using the `<client-reconnect-window>` element:

```
<servers>
  <server>
    ...
    <dso>
      ...
      <!-- The reconnect window is configured in seconds, with a default value of 120. The default
value is "built in," so the element does not have to be explicitly added unless a different value is
required. -->
      <client-reconnect-window>120</client-reconnect-window>
    ...
  </server>
</servers>
```



```

    </dso>
    ...
  </server>
</servers>

```

Clients which fail to connect to the new active server must be restarted if they are to successfully rejoin the cluster.

First-Time Client Connection

When a Terracotta client is first started (or restarted), it attempts to connect to a Terracotta server instance based on the following properties:

```

# -1 == retry all configured servers eternally.
l1.max.connect.retries = -1
# Must the client and server be running the same version of Terracotta?
l1.connect.versionMatchCheck.enabled = true
# Time (in milliseconds) before a socket connection attempt is timed out.
l1.socket.connect.timeout=10000
# Time (in milliseconds; minimum 10) between attempts to connect to a server.
l1.socket.reconnect.waitInterval=1000

```

A client with `l1.max.connect.retries` set to a positive integer is given a limited number of attempts (equal to that integer) to connect. If the client fails to connect after the configured number of attempts, it exits.

6.3 Terracotta Server Arrays

This document shows you how to add cluster reliability, availability, and scalability to a Terracotta Server Array.

A Terracotta Server Array can vary from a basic two-node tandem to a multi-node array providing configurable scale, high performance, and deep failover coverage.

The main features of the Terracotta Server Array include:

- Scalability Without Complexity - Simple configuration to add server instances to meet growing demand and facilitate capacity planning
- High Availability - Instant failover for continuous uptime and services
- Configurable Health Monitoring - Terracotta [HealthChecker](#) for internode monitoring
- Persistent Application State - Automatic permanent storage of all current shared (in-memory) data
- Automatic Node Reconnection - Temporarily disconnected server instances and clients rejoin the cluster without operator intervention

TIP: Nomenclature

This document may refer to a Terracotta server instance as L2, and a Terracotta client (the node running your application) as L1. These are the shorthand references used in Terracotta configuration files.

6.3.1 Definitions and Functional Characteristics

The major components of a Terracotta installation are the following:

- Cluster - All of the Terracotta server instances and clients that work together to share application state or a data set.
- Terracotta Server Array - The platform, consisting of all of the Terracotta server instances in a single cluster. Clustered data, also called in-memory data, or shared data, is partitioned equally among active Terracotta server instances for management and persistence purposes.
- Terracotta mirror group - A unit in the Terracotta Server Array. Sometimes also called a "stripe," a mirror group is composed of exactly one active Terracotta server instance and at least one "hot standby" Terracotta server instance (simply called a "standby"). The active server instance manages and persists the fraction of shared data allotted to its mirror group, while each standby server in the mirror group replicates (or mirrors) the shared data managed by the active server. **Mirror groups add capacity to the cluster**. The standby servers are optional but highly recommended for providing failover.
- Terracotta server instance - A single Terracotta server. An *active* server instance manages Terracotta clients, coordinates shared objects, and persists data. Server instances have no awareness of the clustered applications running on Terracotta clients. A standby (sometimes called "passive") is a live backup server instance which continuously replicates the shared data of an active server instance, instantaneously replacing the active if the active fails. **Standby servers add failover coverage within each mirror group**.

- Terracotta client - Terracotta clients run on application servers along with the applications being clustered by Terracotta. Clients manage live shared-object graphs.

TIP: Switching Server-Array Databases

Another component of the cluster is the embedded database. The Terracotta Server Array uses a licensed database, called BerkeleyDB, to back up all shared (distributed) data. If you would like to switch to using Apache Derby as the embedded database, simply add the following property to the Terracotta configuration file's <tc-properties> block then start or restart the Terracotta Server Array:

```
<property name="l2.db.factory.name" value="com.tc.objectserver.storage.derby.DerbyDBFactory" />
```

You can set additional Apache Derby properties using <property> elements in the <tc-properties> block. For example, to set the property `derby.storage.pageCacheSize=10000` simply append "l2.derbydb" to the property name before adding it:

```
<property name="l2.derbydb.derby.storage.pageCacheSize" value="10000" />
```

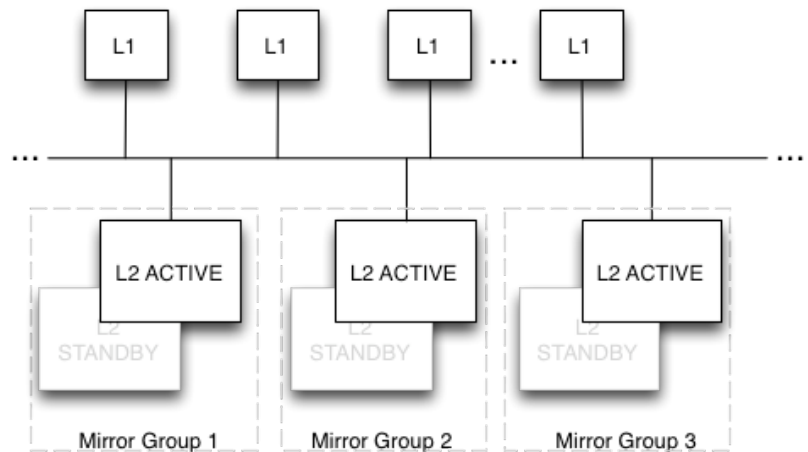
To reset to the default embedded database, remove all Derby-related properties from the Terracotta configuration file and restart the Terracotta Server Array.

If you are using Terracotta Server Array 3.5.1 or later, and want to completely remove BerkeleyDB from your environment, delete the file `je-4.1.7.jar` from the `${TERRACOTTA_HOME}/lib` directory.

Figure 1 illustrates a Terracotta cluster with three mirror groups. Each mirror group has an active server and a standby, and manages one third of the shared data in the cluster.

A Terracotta cluster has the following functional characteristics:

Fig. 1: A Server Array With 3 Mirror Groups



- Each mirror group automatically elects one active Terracotta server instance. There can never be more than one active server instance per mirror group, but there can be any number of standbys. In Fig. 1, Mirror Group 1 could have two standbys, while Mirror Group 3 could have four standbys. However, a performance overhead may become evident when adding more standby servers due to the load placed on the active server by having to synchronize with each standby.
- Every mirror group in the cluster must have a Terracotta server instance in active mode before the cluster is ready to do work.
- The shared data in the cluster is automatically partitioned and distributed to the mirror groups. The number of partitions equals the number of mirror groups. In Fig. 1, each mirror group has one third of the shared data in the cluster.
- Mirror groups **can not** provide failover for each other. Failover is provided *within* each mirror group, not across mirror groups. This is because mirror groups provide scale by managing discrete portions of the shared data in the cluster -- they do not replicate each other. In Fig. 1, if Mirror Group 1 goes down, the cluster must pause (stop work) until Mirror Group 1 is back up with its portion of the shared data intact.
- Active servers are self-coordinating among themselves. No additional configuration is required to coordinate active server instances.
- Only standby server instances can be hot-swapped in an array. In Fig. 1, the L2 PASSIVE (standby) servers can be shut down and replaced with no affect on cluster functions. However, to add or remove an entire mirror group, the cluster must be brought down. Note also that in this case the original Terracotta configuration file is still in effect and no new servers can be added. Replaced standby servers must have the same address (hostname or IP address). If you must swap in a standby with a different configuration, and you have an enterprise edition of Terracotta, see [Changing Cluster Topology in a Live Cluster](#).

6.3.2 Server Array Configuration Tips

To successfully configure a Terracotta Server Array using the Terracotta configuration file, note the

following:

- Two or more servers should be defined in the <servers> section of Terracotta configuration file.
- <l2-group-port> is the port used by the Terracotta server to communicate with other Terracotta servers.
- The <ha> section, which appears immediately after the last <server> section (or the <mirror-groups> section), should declare the mode as "networked-active-passive":

```
<ha>
  <mode>networked-active-passive</mode>
  <networked-active-passive>
    <election-time>5</election-time>
  </networked-active-passive>
</ha>
```

The active-passive mode "disk-based-active-passive" is not recommended except for demonstration purposes or where a networked connection is not feasible. See the *cluster architecture* section in the [Terracotta Concept and Architecture Guide](#) for more information on disk-based active-passive mode.

- The <networked-active-passive> subsection has a configurable parameter called <election-time> whose value is given in seconds. <election-time> sets the duration for electing an ACTIVE server, often a factor in network latency and server load. The default value is 5 seconds.

NOTE: Sharing Data Directories

When using networked-active-passive mode, Terracotta server instances must not share data directories. Each server's <data> element should point to a different and preferably local data directory.

- A reconnection mechanism restores lost connections between active and passive Terracotta server instances. See [Automatic Server Instance Reconnect](#) for more information.
- A reconnection mechanism restores lost connections between Terracotta clients and server instances. See [Automatic Client Reconnect](#) for more information.
- For data safety, persistence should be set to "permanent-store" for server arrays. "permanent-store" means that application state, or shared in-memory data, is backed up to disk. In case of failure, it is automatically restored. Shared data is removed from disk once it no longer exists in any client's memory.

NOTE: Terracotta and Java Versions

All servers and clients should be running the same version of Terracotta and Java.

For more information on Terracotta configuration files, see:

- [Working with Terracotta Configuration Files](#)
- [Configuration Guide and Reference](#) (*Servers Configuration Section*)

6.3.3 Backing Up Persisted Shared Data

Certain versions of Terracotta provide tools to create backups of the Terracotta Server Array disk store. See the [Terracotta Operations Center](#) and the [Database Backup Utility \(backup-data\)](#) for more information.

6.3.4 Client Disconnection

Any Terracotta Server Array handles perceived client disconnection (for example a network failure, a long client GC, or node failure) based on the configuration of the [HealthChecker](#) or [Automatic Client Reconnect](#) mechanisms. A disconnected client also attempts to reconnect based on these mechanisms. The client tries to reconnect first to the initial server, then to any other servers set up in its Terracotta configuration. To preserve data integrity, clients resend any transactions for which they have not received server acks.

For more information on client behavior, see [Cluster Structure and Behavior](#).

6.3.5 Cluster Structure and Behavior

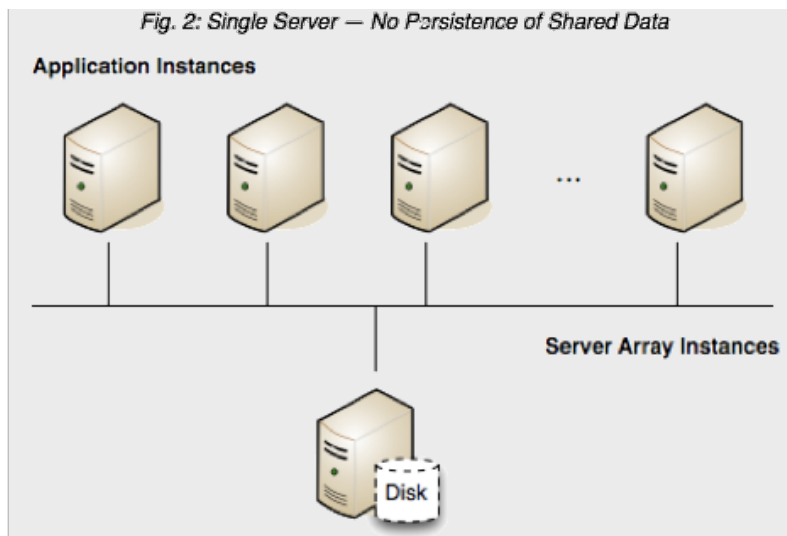
The Terracotta cluster can be configured into a number of different setups to serve both deployment stage and production needs. Note that in multi-setup setups, failover characteristics are affected by HA settings (see [Configuring Terracotta Clusters For High Availability](#)).

6.3.5a Terracotta Cluster in Development

Persistence: No | Failover: No | Scale: No

In a development environment, persisting shared data is often unnecessary and even inconvenient. It puts

more load on the server, while accumulated data can fill up disks or prevent automatic restarts of servers, requiring manual intervention. Running a single-server Terracotta cluster without persistence is a good solution for creating a more efficient development environment.



By default, a single Terracotta server is in "temporary-swap-mode", which means it lacks persistence. Its configuration could look like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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-5.xsd">
  <servers>
    <server name="Server1">
      <data>/opt/terracotta/server1-data</data>
      <l2-group-port>9530</l2-group-port>
    </server>
  </servers>
  ...
</tc:tc-config>
```

Server Restart

If this server goes down, all application state (all clustered data) in the shared heap is lost. In addition, when the server is up again, all clients must be restarted to rejoin the cluster.

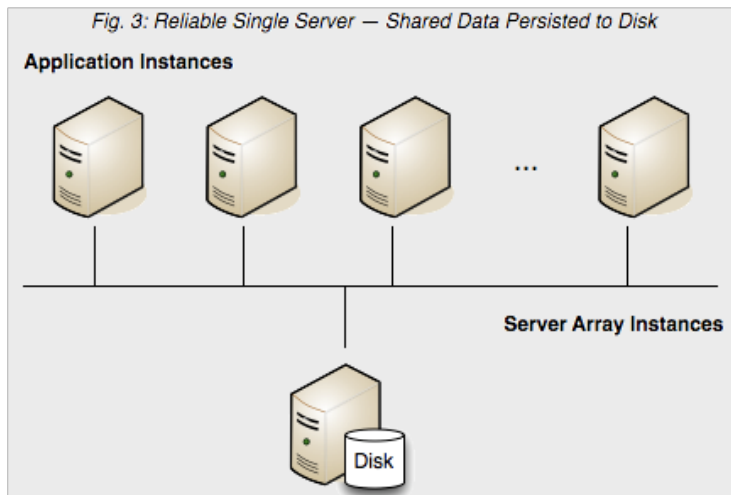
6.3.5b Terracotta Cluster With Reliability

Persistence: Yes | Failover: No | Scale: No

The "unreliable" configuration in [Terracotta Cluster in Development](#) may be advantageous in development, but if shared in-memory data must be persisted, the server's configuration must be expanded:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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-5.xsd">
  <servers>
    <server name="Server1">
      <data>/opt/terracotta/server1-data</data>
      <l2-group-port>9530</l2-group-port>
      <dso>
        <!-- The persistence mode is "temporary-swap-only" by default, so it must be changed explicitly. -->
        <persistence>
          <mode>permanent-store</mode>
        </persistence>
      </dso>
    </server>
  </servers>
  ...
</tc:tc-config>
```

The value of the `<persistence>` element's `<mode>` subelement is "temporary-swap-only" by default. By changing it to "permanent-store", the server now backs up all shared in-memory data to disk.



Server Restart

If the server is restarted, application state (all clustered data) in the shared heap is restored.

In addition, previously connected clients are allowed to rejoin the cluster within a window set by the `<client-reconnect-window>` element:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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-5.xsd">
  <servers>
    <server name="Server1">
      <data>/opt/terracotta/server1-data</data>
      <l2-group-port>9530</l2-group-port>
      <dso>
        <!-- By default the window is 120 seconds. -->
        <client-reconnect-window>120</client-reconnect-window>
        <!-- The persistence mode is "temporary-swap-only" by default, so it must be changed explicitly. -->
        <persistence>
          <mode>permanent-store</mode>
        </persistence>
      </dso>
    </server>
  </servers>
  ...
</tc:tc-config>
```

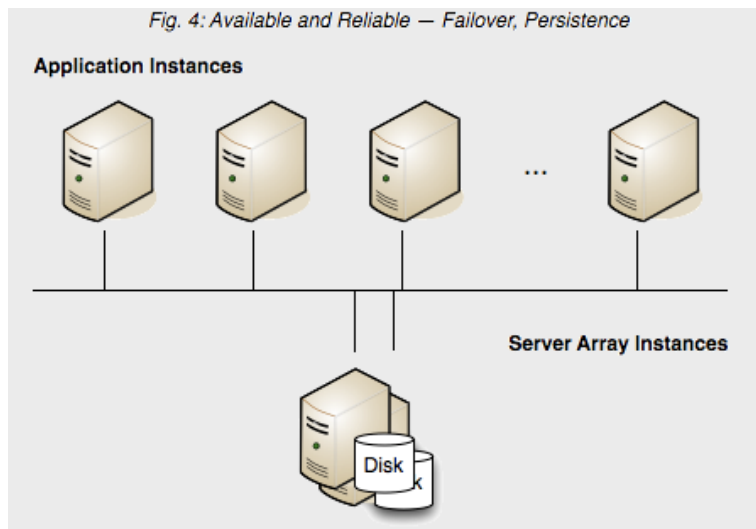
The `<client-reconnect-window>` does not have to be explicitly set if the default value is acceptable.

However, in a single-server cluster `<client-reconnect-window>` is in effect only if persistence mode is set to "permanent-store".

6.3.5c Terracotta Server Array with High Availability

Persistence: Yes | Failover: Yes | Scale: No

The example illustrated in Fig. 3 presents a reliable but *not* highly available cluster. If the server fails, the cluster fails. There is no redundancy to provide failover. Adding a standby server adds availability because the standby failover (see Fig. 4).



In this array, if the active Terracotta server instance fails then the standby instantly takes over and the cluster continues functioning. No data is lost.

The following Terracotta configuration file demonstrates how to configure this two-server array:

```
<?xml version="1.0" encoding="UTF-8" ?>
<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-5.xsd">
  <servers>
    <server name="Server1">
      <data>/opt/terracotta/server1-data</data>
      <l2-group-port>9530</l2-group-port>
      <dso>
        <persistence>
          <mode>permanent-store</mode>
        </persistence>
      </dso>
    </server>
    <server name="Server2">
      <data>/opt/terracotta/server2-data</data>
      <l2-group-port>9530</l2-group-port>
      <dso>
        <persistence>
          <mode>permanent-store</mode>
        </persistence>
      </dso>
    </server>
    <ha>
      <mode>networked-active-passive</mode>
      <networked-active-passive>
        <election-time>5</election-time>
      </networked-active-passive>
    </ha>
  </servers>
  ...
</tc:tc-config>
```

The recommended <mode> in the <ha> section is "networked-active-passive" because it allows the active and passive servers to synchronize directly, without relying on a disk.

You can add more standby servers to this configuration by adding more <server> sections. However, a performance overhead may become evident when adding more standby servers due to the load placed on the active server by having to synchronize with each standby.

Starting the Servers

How server instances behave at startup depends on when in the life of the cluster they are started.

In a single-server configuration, when the server is started it performs a startup routine and then is ready to run the cluster (ACTIVE status). If multiple server instances are started at the same time, one is elected the active server (ACTIVE-COORDINATOR status) while the others serve as standbys (PASSIVE-STANDBY status). The election is recorded in the servers' logs.

If a server instance is started while an active server instance is present, it syncs up state from the active server instance before becoming a standby. The active and passive servers must always be synchronized, allowing the passive to mirror the state of the active. The standby server goes through the following states:

1. **PASSIVE-UNINITIALIZED** - The standby is beginning its startup sequence and is *not* ready to perform failover should the active fail or be shut down. The server's status light in the [Terracotta Developer Console](#) switches from red to yellow.
2. **INITIALIZING** - The standby is synchronizing state with the active and is *not* ready to perform failover should the active fail or be shut down. The server's status light in the [Terracotta Developer Console](#) switches from yellow to orange.
3. **PASSIVE-STANDBY** - The standby is synchronized and is ready to perform failover should the active server fail or be shut down. The server's status light in the [Terracotta Developer Console](#) switches from orange to cyan.

The active server instance carries the load of sending state to the standby during the synchronization process. The time taken to synchronize is dependent on the amount of clustered data and on the current load on the cluster. The active server instance and standbys should be run on similarly configured machines for better throughput, and should be started together to avoid unnecessary sync ups.

Failover

If the active server instance fails and two or more standby server instances are available, an election determines the new active. Successful failover to a new active takes place only if at least one standby server is fully synchronized with the failed active server; successful client failover (migration to the new active) can happen only if the server failover is successful. Shutting down the active server before a fully-synchronized standby is available can result in a cluster-wide failure.

TIP: Hot-Swapping Standbys

Standbys can be hot swapped if the replacement matches the original standby's <server> block in the Terracotta configuration. For example, the new standby should use the same host name or IP address configured for the original standby. If you must swap in a standby with a different configuration, and you have an enterprise edition of Terracotta, see [Changing Cluster Topology in a Live Cluster](#).

Terracotta server instances acting as standbys can run either in persistent mode or non-persistent mode. If an active server instance running in persistent mode goes down, and a standby takes over, the crashed server's data directory must be cleared before it can be restarted and allowed to rejoin the cluster. Removing the data is necessary because the cluster state could have changed since the crash. During startup, the restarted server's new state is synchronized from the new active server instance. A crashed standby running in persistent mode, however, automatically recovers by wiping its own database.

NOTE: Manually Clearing a Standby Server's Data

Under certain circumstances, standbys may fail to automatically clear their data directory and fail to restart, generating errors. In this case, the data directory must be manually cleared.

Even if the standby's data is cleared, a copy of it is saved. By default, the number of copies is unlimited. Over time, and with frequent restarts, these copies may consume a substantial amount of disk space if the amount of shared data is large. You can manually delete these files, which are saved in the server's data directory under `/dirty-objectdb-backup/dirty-objectdb-<timestamp>`. You can also set a limit for the number of backups by adding the following element to the Terracotta configuration file's <tc-properties> block:

```
<property name="l2.nha.dirtydb.rolling" value="<myValue>" />
```

where <myValue> is an integer.

If both servers are down, and clustered data is persisted, the last server to be active should be started first to avoid errors and data loss. Check the server logs to determine which server was last active. (In setups where data is not persisted, meaning that persistence mode is set to "temporary-swap-only", then no data is saved and either server can be started first.)

A Safe Failover Procedure

To safely migrate clients to a standby server without stopping the cluster, follow these steps:

1. If it is not already running, start the standby server using the `start-tc-server` script.
The standby server must already be configured in the Terracotta configuration file.
2. Ensure that the standby server is ready for failover (PASSIVE-STANDBY status).
3. Shut down the active server using the `stop-tc-server` script.
Clients should connect to the new active server.
4. Restart any clients that fail to reconnect to the new active server within the configured reconnection

window.

5. If running with persistence, delete the database of the previously active server before restarting it. The previously active server can now rejoin the cluster as a standby server.

A Safe Cluster Shutdown Procedure

For a cluster with persistence, a safe cluster shutdown should follow these steps:

1. Shut down the standby server using the stop-tc-server script.
2. Shut down the clients.
The Terracotta client will shut down when you shut down your application.
3. Shut down the active server using the stop-tc-server script.

To restart the cluster, first start the server that was last active. If clustered data is not persisted, either server could be started first as no database conflicts can take place.

Split Brain Scenario

In a Terracotta cluster, "split brain" refers to a scenario where two servers assume the role of active server (ACTIVE-COORDINATOR status). This can occur during a network problem that disconnects the active and standby servers, causing the standby to both become an active server and open a reconnection window for clients (<client-reconnect-window>).

If the connection between the two servers is never restored, then two independent clusters are in operation. This is not a split-brain situation. However, if the connection is restored, one of the following scenarios results:

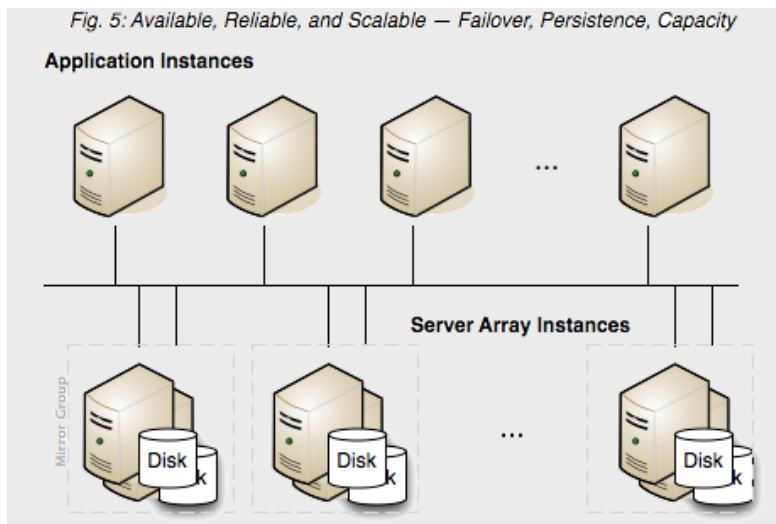
- No clients connect to the new active server - The original active server "zaps" the new active server, causing it to restart, wipe its database, and synchronize again as a standby.
- A minority of clients connect to the new active server - The original active server starts a reconnect timeout (based on HA settings; see [Configuring Terracotta Clusters For High Availability](#)) for the clients that it loses, while zapping the new active server. The new active restarts, wipes its database, and synchronizes again as a standby. Clients that defected to the new active attempt to reconnect to the original active, but if they do not succeed within the parameters set by that server, they must be restarted.
- A majority of clients connects to the new active server - The new active server "zaps" the original active server. The original active restarts, wipes its database, and synchronizes again as a standby. Clients that do not connect to the new active within its configured reconnection window must be restarted.
- An equal number of clients connect to the new active server - In this unlikely event, exactly one half of the original active server's clients connect to the new active server. The servers must now attempt to determine which of them holds the latest transactions (or has the freshest data). The winner zaps the loser, and clients behave as noted above, depending on which server remains active. Manual shutdown of one of the servers may become necessary if a timely resolution does not occur.

The cluster can solve almost all split-brain occurrences without loss or corruption of shared data. However, it is highly recommended that after such an occurrence the integrity of shared data be confirmed.

6.3.5d Scaling the Terracotta Server Array

Persistence: Yes | Failover: Yes | Scale: Yes

For capacity requirements that exceed the capabilities of a two-server active-passive setup, expand the Terracotta cluster using a mirror-groups configuration. Mirror groups are available with an enterprise version of Terracotta software. Using mirror groups with multiple coordinated active Terracotta server instances adds scalability to the Terracotta Server Array.



Mirror groups are specified in the `<servers>` section of the Terracotta configuration file. Mirror groups work by assigning group memberships to Terracotta server instances. The following snippet from a Terracotta configuration file shows a mirror-group configuration with four servers:

```
...
<servers>
  <server name="server1">
    ...
  </server>
  <server name="server2">
    ...
  </server>
  <server name="server3">
    ...
  </server>
  <server name="server4">
    ...
  </server>
  <mirror-groups>
    <mirror-group group-name="groupA">
      <members>
        <member>server1</member>
        <member>server2</member>
      </members>
    </mirror-group>
    <mirror-group group-name="groupB">
      <members>
        <member>server3</member>
        <member>server4</member>
      </members>
    </mirror-group>
  </mirror-groups>
  <ha>
    <mode>networked-active-passive</mode>
    <networked-active-passive>
      <election-time>5</election-time>
    </networked-active-passive>
  </ha>
</servers>
...
```

In this example, the cluster is configured to have two active servers, each with its own standby. If server1 is elected active in groupA, server2 becomes its standby. If server3 is elected active in groupB, server4 becomes its standby. server1 and server2 automatically coordinate their work managing Terracotta clients and shared data across the cluster.

In a Terracotta cluster designed for multiple active Terracotta server instances, the server instances in each mirror group participate in an election to choose the active. Once every mirror group has elected an active server instance, all the active server instances in the cluster begin cooperatively managing the cluster. The rest of the server instances become standbys for the active server instance in their mirror group. If the active

in a mirror group fails, a new election takes place to determine that mirror group's new active. Clients continue work without regard to the failure.

In a Terracotta cluster with mirror groups, each group, or "stripe," behaves in a similar way to an active-passive setup (see [Terracotta Server Array with High Availability](#)). For example, when a server instance is started in a stripe while an active server instance is present, it synchronizes state from the active server instance before becoming a standby. A standby cannot become an active server instance during a failure until it is fully synchronized. If an active server instance running in persistent mode goes down, and a standby takes over, the data directory must be cleared before bringing back the crashed server.

TIP: Hot-Swapping Standbys

Standbys can be hot swapped if the replacement matches the original standby's <server> block in the Terracotta configuration. For example, the new standby should use the same host name or IP address configured for the original standby. If you must swap in a standby with a different configuration, and you have an enterprise edition of Terracotta, see [Changing Cluster Topology in a Live Cluster](#).

Stripe and Cluster Failure

If the active server in a mirror group fails or is taken down, the cluster stops until a standby takes over and becomes active (ACTIVE-COORDINATOR status).

However, the cluster cannot survive the loss of an entire stripe. If an entire stripe fails and no server in the failed mirror-group becomes active within the allowed window (based on HA settings; see [Configuring Terracotta Clusters For High Availability](#)), the entire cluster must be restarted.

High Availability Per Mirror Group

High-availability configuration can be set per mirror group. The following snippet from a Terracotta configuration file shows a mirror-group configured with its own high-availability section:

```
...
<servers>
  <server name="server1">
    ...
  </server>
  <server name="server2">
    ...
  </server>
  <server name="server3">
    ...
  </server>
  <server name="server4">
    ...
  </server>
  <mirror-groups>
    <mirror-group group-name="groupA">
      <members>
        <member>server1</member>
        <member>server2</member>
      </members>
      <ha>
        <mode>networked-active-passive</mode>
        <networked-active-passive>
          <election-time>10</election-time>
        </networked-active-passive>
      </ha>
    </mirror-group>
    <mirror-group group-name="groupB">
      <members>
        <member>server3</member>
        <member>server4</member>
      </members>
    </mirror-group>
  </mirror-groups>
  <ha>
    <mode>networked-active-passive</mode>
    <networked-active-passive>
      <election-time>5</election-time>
    </networked-active-passive>
  </ha>
</servers>
```

...

In this example, the servers in groupA can take up to 10 seconds to elect an active server. The servers in groupB take their election time from the <ha> section outside the <mirror-groups> block, which is in force for all mirror groups that do not have their own <ha> section.

See the *mirror-groups* section in the [Configuration Guide and Reference](#) for more information on mirror-groups configuration elements.

6.4 Improving Server Performance With BigMemory

Servers can have a large amount of physical memory--16GB, 32GB, and more--but the long-standing problem of Java garbage collection (GC) limits the ability of all Java applications, including Terracotta server instances, to use that memory effectively. This drawback has limited Terracotta servers to using a small Java object heap as an in-memory store, backed by a limitless but slower disk store.

BigMemory gives Terracotta servers instant, effortless access to hardware memory free of the constraints of GC. BigMemory is available with enterprise versions of Terracotta software, and can also be used with Terracotta DSO.

6.4.1 How BigMemory Improves Performance

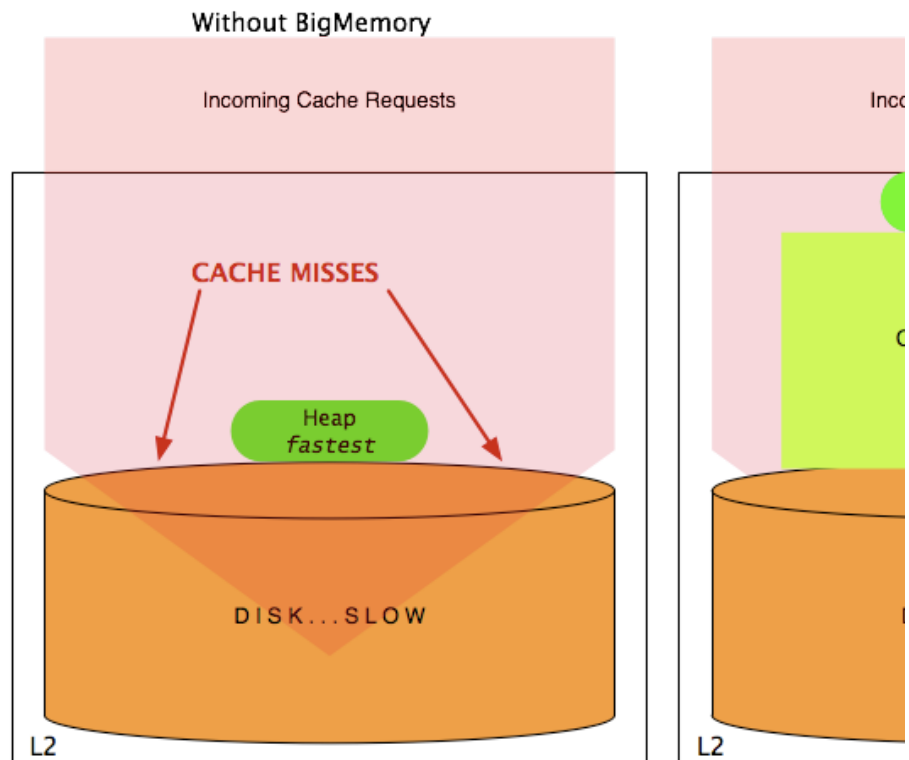
The performance of Terracotta server instances is affected by the amount of faulting required to make data available. In-memory data elements are fetched very quickly because memory is very fast. Data elements that are not found in memory must be faulted in from disk, and sometimes from an even slower system of record, such as a database. While disk-based storage slows applications down, it is limitless in size. While in-memory storage is limited in size by system and hardware constraints, even this limit is difficult to reach due to the heavy costs imposed by Java garbage collection (GC). Full GC operations can slow a system to a crawl, and the larger the heap, the more often these operations are likely to occur. In most cases, heaps have been limited to about 2GB in size.

BigMemory allows Terracotta servers to expand memory storage in a way that bypasses the limitations resulting from Java GC. Using this *off-heap* memory gives the Terracotta cluster a number of important advantages:

- Larger in-memory stores without the pauses of GC.
- Overall reduction in faulting from disk or database.
- Low latencies as a result of more data available to applications at memory speed.
- Fewer Terracotta server stripes required to efficiently handle the same amount of data.

Data stored in off-heap memory is stored in a cache, and therefore all data elements (keys and values) must be serializable. However, the costs imposed by serialization and deserialization are far outweighed by the performance gains noted above.

The following diagram illustrates how BigMemory adds a layer of off-heap memory storage that reduces faulting from the Terracotta server's disk yet remains outside of GC's domain.



6.4.2 Requirements

BigMemory runs on each Terracotta server in a Terracotta Server Array. To use BigMemory, you must install and run a Terracotta enterprise kit (version 3.4.0 or higher) and a valid Terracotta license key that includes BigMemory. For more information on installing a license key, see [Using a License File](#).

BigMemory requires the 32-bit or 64-bit Oracle HotSpot (formerly Sun HotSpot) JVM. 64-bit systems can operate with more memory than 32-bit systems. Running BigMemory on a 64-bit system allows for more off-heap memory to be allocated.

NOTE: Using a 32-bit JVM

The amount of heap-offload you can achieve is limited by addressable memory. For a 32-bit process model, the maximum virtual address size of the process is typically 4GB, though most 32-bit operating systems have a 2GB limit. The maximum heap size available to Java is lower still due to particular OS limitations, other operations that may run on the machine (such as mmap operations used by certain APIs), and various JVM requirements for loading shared libraries and other code.

A useful rule to observe is to allocate no more to off-heap memory than what is left over after `-Xmx` is set. For example, if you set `-Xmx3G`, then off-heap should be no more than 1GB. Breaking this rule may not cause an OOME on startup, but one is likely to occur at some point during the JVM's life.

6.4.3 Configuring BigMemory

BigMemory is configured in the Terracotta server's environment and in its configuration file.

6.4.3a Configuring Direct Memory Space

Before starting a Terracotta server with off-heap, direct memory space, also called direct (memory) buffers, must be allocated. Direct memory space is allocated using the Java property `MaxDirectMemorySize`:

```
-XX:MaxDirectMemorySize=<amount of memory allotted>[m|g]
```

where "m" stands for megabytes (MB) and "g" stands for gigabytes (GB).

Note the following about allocating direct memory space:

- `MaxDirectMemorySize` must be added to the Terracotta server's startup environment. For example, you can add it the server's Java options in `${TERRACOTTA_HOME}/bin/start-tc-server.sh` or `%TERRACOTTA_HOME%\bin\start-tc-server.bat`.
- Direct memory space, which is part of the Java process heap, is separate from the object heap allocated by `-Xmx`. The value allocated by `MaxDirectMemorySize` must not exceed physical RAM.

and is likely to be less than total available RAM due to other memory requirements.

- The amount of direct memory space allocated must be within the constraints of available system memory and configured off-heap memory (see [6.4.3b Configuring Off-Heap](#)).

6.4.3b Configuring Off-Heap

BigMemory is set up by configuring off-heap memory in the Terracotta configuration file for each Terracotta server, then allocating memory at startup using `MaxDirectMemorySize`. For example, to allocate up to 9GB of off-heap memory, add the `<offheap>` block as shown:

```
<server host="myHost" name="server1">
  <dso>
    ...
    <persistence>
      ...
      <offheap>
        <enabled>true</enabled>
<!-- Allocate 9GB of off-heap memory for clustered data. -->
        <maxDataSize>9g</maxDataSize>
      </offheap>
      ...
    </persistence>
    ...
  </dso>
  ...
</server>
```

The amount of configured off-heap memory must be at least 128MB and at least 32MB less than the amount of off-heap memory allocated by `MaxDirectMemorySize`. If, at startup, a server determines that the memory allocated by `MaxDirectMemorySize` is insufficient, an error similar to the following is logged:

```
2011-03-28 07:39:59,316 ERROR - The JVM argument -XX:MaxDirectMemorySize(128m) cannot be less than TC
minimum Direct memory requirement: 202.22m
```

In this case, you must set `MaxDirectMemorySize` to a value equal to or greater than the minimum given in the error.

6.4.3c Maximum, Minimum, and Default Values

The *maximum* amount of direct memory space you can use depends on the process data model (32-bit or 64-bit) and the associated operating system limitations, the amount of virtual memory available on the system, and the amount of physical memory available on the system. While 32-bit systems have strict limitations on the amount of memory that can be effectively managed, 64-bit systems can allow as much memory as the hardware and operating system can handle.

The *maximum* amount you can allocate to off-heap memory cannot exceed the amount of direct memory space, and should likely be less because direct memory space may be shared with other Java and system processes.

The *minimum* off-heap you can allocate per server is 160MB.

If you configure off-heap memory but do not allocate direct memory space with

`-XX:MaxDirectMemorySize`, the *default* value for direct memory space depends on your version of your JVM. Oracle HotSpot has a default equal to maximum heap size (`-Xmx` value), although some early versions may default to a particular value.

6.4.4 Optimizing BigMemory

You should thoroughly test BigMemory with your application before going to production. If performance or functional issues arise, see the suggested tuning tips in this section. **It is recommended that you test BigMemory with the actual amount of data you expect to use in production.**

6.4.4a General Memory allocation

Committing too much of a system's physical memory is likely to result in paging of virtual memory to disk, quite likely during garbage collection operations, leading to significant performance issues. On systems with multiple Java processes, or multiple processes in general, the sum of the Java heaps and off-heap stores for those processes should also not exceed the size of the physical RAM in the system. Besides memory allocated to the heap, Java processes require memory for other items, such as code (classes), stacks, and PermGen.

Note that `MaxDirectMemorySize` sets an upper limit for the JVM to enforce, but does not actually allocate the specified memory. Overallocation of direct memory (or buffer) space is therefore possible, and could lead to paging or even memory-related errors. The limit on direct buffer space set by `MaxDirectMemorySize` should

take into account the total physical memory available, the amount of memory that is allotted to the JVM object heap, and the portion of direct buffer space that other Java processes may consume.

Note also that there could be other users of direct buffers (such as NIO and certain frameworks and containers). Consider allocating additional direct buffer memory to account for that additional usage.

6.4.4b Compressed References

For 64-bit JVMs running Java 6 Update 14 or higher, consider enabling compressed references to improve overall performance. For heaps up to 32GB, this feature causes references to be stored at half the size, as if the JVM is running in 32-bit mode, freeing substantial amounts of heap for memory-intensive applications. The JVM, however, remains in 64-bit mode, retaining the advantages of that mode.

For the Oracle HotSpot, compressed references are enabled using the option `-XX:+UseCompressedOop`.

6.4.4c Swappiness and Huge Pages

An operating system (OS) may swap data from memory to disk even if memory is not running low. For the purpose of optimization, data that appears to be unused may be a target for swapping. Because BigMemory can store substantial amounts of data in RAM, its data may be swapped by the OS. But swapping can degrade overall cluster performance by introducing thrashing, the condition where data is frequently moved forth and back between memory and disk.

To make memory use more efficient, Linux, Microsoft Windows, and Oracle Solaris users should review their configuration and usage of swappiness as well as the size of the swapped memory pages. In general, BigMemory benefits from lowered swappiness and the use of *huge pages* (also known as *big pages* or *large pages*), and *superpages*).

6.5 Cluster Security

The Enterprise Edition of the Terracotta kit provides standard authentication methods to control access to Terracotta servers. Enabling one of these methods causes a Terracotta server to require credentials before allowing a JMX connection to proceed.

6.5.1 Configuring Security

You can configure security using the Lightweight Directory Access Protocol (LDAP) or JMX authentication.

6.5.1a How to Configure Security Using LDAP (via JAAS)

LDAP security is based on JAAS and requires Java 1.6. Using an earlier version of Java will not prevent Terracotta servers from running; however security will *not* be enabled.

To configure security using LDAP, follow these steps:

1. Save the following configuration to the file `.java.login.config`:

```
Terracotta {
com.sun.security.auth.module.LdapLoginModule REQUIRED
java.naming.security.authentication="simple"
userProvider="ldap://orgstage:389"
authIdentity="uid={USERNAME},ou=People,dc=terracotta,dc=org"
authzIdentity=controlRole
useSSL=false
bindDn="cn=Manager"
bindCredential="*****"
bindAuthenticationType="simple"
debug=true;
};
```

2. Save the file `.java.login.config` to the directory named in the Java property `user.home`.
3. Add the following configuration to each `<server>` block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
  <authentication>
    <mode>
      <login-config-name>Terracotta</login-config-name>
    </mode>
  </authentication>
...
```



```
</server>
```

4. Start the Terracotta server and look for a log message containing "INFO - Credentials: loginConfig[Terracotta]" to confirm that LDAP security is in effect.

NOTE: Incorrect Setup

If security is set up incorrectly, the Terracotta server can still be started. However, you may not be able to shut down the server using the shutdown script (`stop-tc-server`) or the Terracotta console.

6.5.1b How to Configure Security Using JMX Authentication

Terracotta can use the standard Java security mechanisms for JMX authentication, which relies on the creation of `.access` and `.password` files with correct permissions set. The default location for these file for JDK 1.5 or higher is `$JAVA_HOME/jre/lib/management`.

To configure security using JMX authentication, follow these steps:

1. Ensure that the desired usernames and passwords for securing the target servers are in the JMX password file `jmxremote.password` and that the desired roles are in the JMX access file `jmxremote.access`.
2. If both `jmxremote.access` and `jmxremote.password` are in the default location (`$JAVA_HOME/jre/lib/management`), add the following configuration to each `<server>` block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
  <authentication />
...
</server>
```

3. If `jmxremote.password` is not in the default location, add the following configuration to each `<server>` block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
  <authentication>
    <mode>
      <password-file>/path/to/jmx.password</password-file>
    </mode>
  </authentication>
...
</server>
```

4. If `jmxremote.access` is not in the default location, add the following configuration to each `<server>` block in the Terracotta configuration file:

```
<server host="myHost" name="myServer">
...
  <authentication>
    <mode>
      <password-file>/path/to/jmxremote.password</password-file>
    </mode>
    <access-file>/path/to/jmxremote.access</access-file>
  </authentication>
...
</server>
```

File Not Found Error

If the JMX password file is not found when the server starts up, an error is logged stating that the password file does not exist.

6.5.2 Using Scripts Against a Server with Authentication

A script that targets a secured Terracotta server must use the correct login credentials to access the server. If you run a Terracotta script such as `backup-data` or `server-stat` against a secured server, pass the credentials using the `-u` (followed by username) and `-w` (followed by password) flags.

For example, if Server1 is secured with username "user1" and password "password", you run the `server-stat` script by entering the following:

UNIX/LINUX

```
[PROMPT]${TERRACOTTA_HOME}/bin/server-stat.sh -s Server1 -u user1 -w password
```

MICROSOFT WINDOWS

```
[PROMPT]%TERRACOTTA_HOME%\bin\server-stat.bat -s Server1 -u user1 -w password
```

6.5.3 Extending Server Security

Since JMX messages are not encrypted, server authentication does not provide secure message transmission once valid credentials are provided by a listening client. To extend security beyond the login threshold, consider the following options:

- Place Terracotta servers in a secure location on a private network.
- Restrict remote queries to an encrypted tunnel such as provided by SSH or stunnel.
- If using public or outside networks, use a VPN for all communication in the cluster.
- If using Ehcache, add a cache decorator to the cache that implements your own encryption and decryption.

6.6 Changing Cluster Topology in a Live Cluster

Using the Terracotta Operations Center, a standalone enterprise-only console for operators, you can change the topology of a live cluster by reloading an edited Terracotta configuration file.

Note the following restrictions:

- Only the removal or addition of <server> blocks in the <servers> section of the Terracotta configuration file are allowed.
- All servers and clients must load the same configuration file to avoid topology conflicts.

6.6.1 Adding a New Server

To add a new server to a Terracotta cluster, follow these steps:

1. Add a new <server> block to the <servers> section in the Terracotta configuration file being used by the cluster.
The new <server> block should contain the minimum information required to configure a new server. It should appear similar to the following, with your own values substituted:

```
<server host="myHost" name="server2" >
  <data>%(user.home)/terracotta/server2/server-data</data>
  <logs>%(user.home)/terracotta/server2/server-logs</logs>
  <statistics>%(user.home)/terracotta/server2/server-stats</statistics>
  <dso-port>9513</dso-port>
</server>
```

2. If you are using mirror groups, be sure to add a <member> element to the appropriate group listing the new server.
3. Open the Terracotta Operations Center by running the following script:

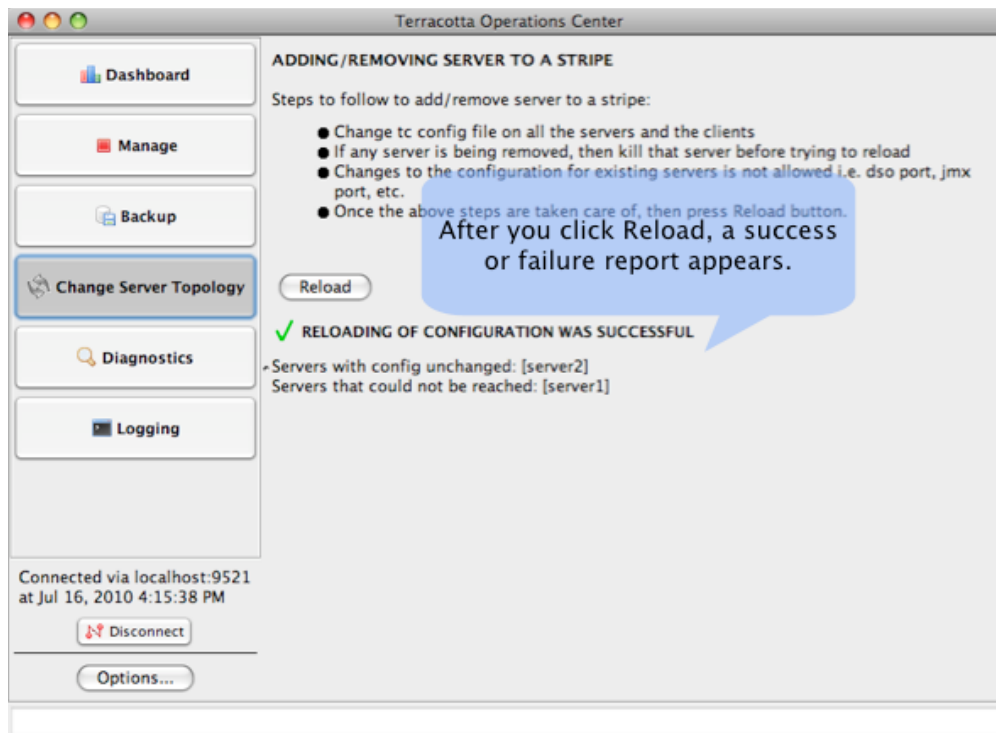
UNIX/LINUX

```
${TERRACOTTA_HOME}/bin/ops-center.sh
```

MICROSOFT WINDOWS

```
%TERRACOTTA_HOME%\bin\ops-center.bat
```

4. After connecting to the cluster with the Operations Center, open the **Change Server Topology** panel.
5. Click **Reload** .
A message appears with the result of the reload operation.



6. Start the new server.

6.6.2 Removing an Existing Server

To remove a server from a Terracotta cluster configuration, follow these steps:

1. Shut down the server you want to remove from the cluster.
If you shutting down an active server, first ensure that a backup sever is online to enable failover.
2. Delete the <server> block associated with the removed server from the <servers> section in the Terracotta configuration file being used by the cluster.
3. If you are using mirror groups, be sure to remove the <member> element associated with the remover server.
4. Open the Terracotta Operations Center by running the following script:

UNIX/LINUX

```
${TERRACOTTA_HOME}/bin/ops-center.sh
```

MICROSOFT WINDOWS

```
%TERRACOTTA_HOME%\bin\ops-center.bat
```

5. After connecting to the cluster with the Operations Center, open the **Change Server Topology** panel.
6. Click **Reload** .
A message appears with the result of the reload operation.

6.6.3 Editing the Configuration of an Existing Server

If you edit the configuration of an existing server and attempt to reload its configuration, the reload operation will fail. However, you can successfully edit an existing server's configuration by following these steps:

1. Remove the server by following the steps in [Removing an Existing Server](#).
Instead of deleting the server's <server> block, you can comment it out.
2. Edit the server's <server> block with the changed values.
3. Add (or uncomment) the edited <server> block.
4. If you are using mirror groups, be sure to add the <member> element associated with the server back to the appropriate mirror group.
5. In the Operations Center's **Change Server Topology** panel, click **Reload** .
A message appears with the result of the reload operation.

7 Developing Applications With the Terracotta Toolkit

The Terracotta Toolkit is intended for developers working on scalable applications, frameworks, and software tools. The Terracotta Toolkit provides the following features:

- Ease-of-use - A stable API, fully documented classes (see the [Terracotta Toolkit Javadoc](#)), and a versioning scheme that's easy to understand.
- Guaranteed compatibility - Verified by a Terracotta Compliance Kit that tests all classes to ensure backward compatibility.
- Extensibility - Includes all of the tools used to create Terracotta products, such as concurrent maps, locks, counters, queues.
- Flexibility - Can be used to build clustered products that communicate with multiple clusters.
- Platform independence - Runs on any Java 1.5 or 1.6 JVM and requires no boot-jars, agents, or container-specific code.

The Terracotta Toolkit is available with Terracotta kits version 3.3.0 and higher.

7.0.1 Installing the Terracotta Toolkit

The Terracotta Toolkit is contained in the following JAR file:

`${TERRACOTTA_HOME}/common/terracotta-toolkit-<API version>-runtime-<JAR version>.jar`

The Terracotta Toolkit JAR file should be on your application's classpath or in `WEB-INF/lib` if using a WAR file.

Maven users can add the Terracotta Toolkit as a dependency:

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

7.0.2 Understanding Versions

The products you create with the Terracotta Toolkit depend on the API at its heart. The Toolkit's API has a version number with a major digit and a minor digit that indicate its compatibility with other versions. The major version number indicates a breaking change, while the minor version number indicates a compatible change. For example, Terracotta Toolkit API version 1.1 is compatible with version 1.0. Version 1.2 is compatible with both versions 1.1 and 1.0. Version 2.0 is not compatible with any version 1.x, but will be forward compatible with any version 2.x.

7.1 Working With the Terracotta Toolkit

The Terracotta Toolkit provides access to a number of useful classes, or tools, such as distributed collections. To access the tools in the Toolkit, your application must also first initialize the Terracotta Toolkit.

7.1.1 Initializing the Toolkit

Initializing the Terracotta Toolkit always begins with starting a Terracotta client:

```
...
// These classes must be imported:
import org.terracotta.api.ClusteringToolkit;
import org.terracotta.api.TerracottaClient;
...
TerracottaClient client = new TerracottaClient("localhost:9510"); // Start the client.
ClusteringToolkit toolkit = client.getToolkit(); // Make the Toolkit available to your application.
...
```

Or more compactly:

```
...
import org.terracotta.api.ClusteringToolkit;
import org.terracotta.api.TerracottaClient;
...
ClusteringToolkit toolkit = new TerracottaClient("localhost:9510").getToolkit();
...
```

When a Terracotta client is started, it must load a Terracotta configuration. Programmatically, the `TerracottaClient` constructor takes as argument the source for the client's configuration. In the example above, the configuration source is a Terracotta server running on the local host, with DSO port set to 9510. In general, a filename, an URL, or a resolvable hostname or IP address and DSO port number can be used. The specified server instance must be running and accessible before the code that starts the client executes.

7.1.2 Using Toolkit Tools

The data structures and other tools provided by the Toolkit are automatically distributed (clustered) when your application is run in a Terracotta cluster. Since the Toolkit is obtained from an instantiated client, all Toolkit tools must be used clustered. Unclustered use is not supported in this version.

7.1.2a Toolkit Data Structures and Serialization

Only serialized objects can be added to Toolkit data structures, with the following exceptions:

- Java primitives (int, char, byte, etc.)
- Wrapper classes for Java primitives (Integer, Character, Byte, etc.)
- BigInteger and BigDecimal (from java.math.)
- String, Class, and StackTraceElement (from java.lang)
- java.util.Currency
- All Enum types

In addition, Arrays of any of the types listed above also do not require serialization. Note that your application must perform the serialization.

7.1.2b Maps

Clustered collections are found in the package `org.terracotta.collections`. This package includes a clustered `Map`, `BlockingQueue`, `Set`, and `List`. You can access these clustered collections directly through the Terracotta Toolkit.

For example, the following gets a reference to a clustered map:

```
...
import org.terracotta.api.ClusteringToolkit;
import org.terracotta.api.TerracottaClient;
import org.terracotta.collections.ClusteredMap;
...
ClusteringToolkit toolkit = new TerracottaClient("localhost:9510").getToolkit();
...
ClusteredMap<int, Object> myClusteredMap = toolkit.getMap("myMap");
```

The returned map is a fully concurrent implementation of the `ClusteredMap` interface, which means that locking is provided.

TIP: Does a Collection Provide Locking?

If a class's name includes *concurrent*, it provides locking. For example, the `List` implementation, `TerracottaList`, does not provide locking.

7.1.2c Queues

To obtain a clustered `BlockingQueue`, use the following:

```
BlockingQueue<byte[]> queue = clusterToolkit.getBlockingQueue(String MY_QUEUE);
```

where the `String MY_QUEUE` holds the name of the queue. This `BlockingQueue` has unlimited capacity.

To obtain a clustered `BlockingQueue` with a limited capacity, use the following:

```
BlockingQueue<byte[]> queue = clusterToolkit.getBlockingQueue(MY_QUEUE, MAX_ITEMS);
```

where the `int MAX_ITEMS` is the maximum capacity of the queue.

Producers in the Terracotta cluster can add to the clustered queue with `add()`, while consumers can take from the queue with `take()`. The clustered queue's data is automatically shared and updated across the

Terracotta cluster so that all nodes have the same view.

7.1.2d Cluster Information

The Terracotta Toolkit allows you to access cluster information for monitoring the nodes in the cluster, as well as obtaining information about those nodes.

For example, you can set up a cluster listener to receive events about the status of client nodes:

```
import org.terracotta.api.ClusteringToolkit;
import org.terracotta.api.TerracottaClient;
import org.terracotta.cluster;
...

// Start a client and access cluster events and meta data such as topology for the cluster that the
// client belongs to:
ClusteringToolkit toolkit = new TerracottaClient("localhost:9510")
    .getToolkit();
ClusterInfo clusterInfo = toolkit.getClusterInfo();

// Register a cluster listener and implement methods for events:
clusterInfo.addClusterListener(new ClusterListener())
{
    // Implement methods for nodeJoined, nodeLeft, etc. here:
    public void nodeJoined(ClusterEvent event) {
        // Do something when event is received.
    }
    public void nodeLeft(ClusterEvent event) {
        // Do something when event is received.
    }
    public void operationsEnabled(ClusterEvent event) {
        // Do something when event is received.
    }
    public void operationsDisabled(ClusterEvent event) {
        // Do something when event is received.
    }
}
```

You can write your own listener classes that implements the event methods, and add or remove your own listeners:

```
clusterInfo.addClusterListener(new MyClusterListener());

// To remove a listener:
clusterInfo.removeClusterListener(myClusterListener);
```

7.1.2e Locks

Clustered locks allow you to perform safe operations on clustered data. The following types of locks are available:

- READ - This is a read lock that blocks writes.
- WRITE - This is a write lock that blocks reads and writes. To improve performance, this lock flushes changes to the Terracotta Server Array asynchronously.
- SYNCHRONOUS-WRITE - A write lock that blocks until the Terracotta Server Array acknowledges commitment of the changes that the lock has flushed to it. Maximizes safety at the cost of performance.
- CONCURRENT - A lock that makes no guarantees that any of the changes flushed to the Terracotta Server Array have been committed. This lock is high-risk and used only where data integrity is unimportant.

To obtain a clustered lock, use `ClusteringToolkit.createLock(Object monitor, LockType type)`. For example, to obtain a write lock:

```
import org.terracotta.api.ClusteringToolkit;
import org.terracotta.api.TerracottaClient;
import org.terracotta.locking;
import org.terracotta.locking.strategy;
...

// Start a client.
ClusteringToolkit toolkit = new TerracottaClient("localhost:9510")
    .getToolkit();
```

```
// Obtain a clustered lock. The monitor object must be clustered and cannot be null.
Lock myLock = toolkit.createLock(myMonitorObject, WRITE);
myLock.lock();
try {
    // some operation under the lock
} finally {
    myLock.unlock();
}
```

To obtain a clustered read-write lock:

```
TerracottaClient client = new TerracottaClient("myServer:9510");

// If the identified lock exists, it is returned instead of created.
Lock rwlock = client.getToolkit().getReadWriteLock("my-lock-identifier").writeLock();

rwlock.lock();
try {
    // some operation under the lock
} finally {
    rwlock.unlock();
}
```

If you are using Enterprise Ehcache, you can use explicit locking methods on specific keys. See [2.2.5 Explicit Locking](#) for more information.

7.1.2f Clustered Barriers

Coordinating independent nodes is useful in many aspects of development, from running more accurate performance and capacity tests to more effective management of workers across a cluster. A clustered barrier is a simple and effective way of coordinating client nodes.

To get a clustered barrier:

```
import org.terracotta.api.ClusteringToolkit;
import org.terracotta.api.TerracottaClient;
import org.terracotta.coordination;
...

// Start a client.
ClusteringToolkit toolkit = new TerracottaClient("localhost:9510")
    .getToolkit();

// Get a clustered barrier. Note that getBarrier() as implemented in Terracotta Toolkit returns a
CyclicBarrier.
Barrier clusteredBarrier = toolkit.getBarrier(String barrierName, int numberOfParties);
```

7.1.2g Utilities

Utilities such as a clustered `AtomicLong` help track counts across a cluster. You can get (or create) a `ClusteredAtomicLong` using `toolkit.getAtomicLong(String name)`.

Another utility, `ClusteredTextBucket`, shares string outputs from all nodes. Printed output from each local node is available on every other node via this bucket. You can get (or create)

`ClusteredTextBucket` using `toolkit.getTextBucket(String name)`.

7.2 Terracotta Toolkit Reference

This section describes functional aspects of the Terracotta Toolkit.

7.2.1 Client Failures

Clients that fail will fail with `System.exit()` and therefore shut down the JVM. The node on which the client failed will go down, as will all other clients and applications in that JVM.

7.2.2 Connection Issues

Client creation can block on resolving URL at this point:

```
TerracottaClient client = new TerracottaClient("myHost:9510");
```


If it is known that resolving "myHost" may take too long or hang, your application can should wrap client instantiation with code that provides a reasonable timeout.

A separate connection issue can occur after the server URL is resolved but while the client is attempting to connect to the server. The timeout for this type of connection can be set using the Terracotta property `ll.socket.connect.timeout` (see [First-Time Client Connection](#)).

7.2.3 Multiple Terracotta Clients in a Single JVM

When using the Terracotta Toolkit, you may notice that there are more Terracotta clients in the cluster than expected.

7.2.3a Multiple Clients With a Single Web Application

This situation can arise whenever multiple classloaders are involved with multiple copies of the Toolkit JAR.

For example, to run a web application in Tomcat, one copy of the Toolkit JAR may need to be in the application's `WEB-INF/lib` directory while another may need to be in Tomcat's common `lib` directory to support loading of the context-level `<Valve>`. In this case, two Terracotta clients will be running with every Tomcat instance.

7.2.3b Clients Sharing a Node ID

Clients instantiated using the same constructor (a constructor with matching parameters) in the same JVM will share the same node ID. For example, the following clients will have the same node ID:

```
TerracottaClient client1 = new TerracottaClient("myHost:9510");
TerracottaClient client2 = new TerracottaClient("myHost:9511");
```

Cluster events generated from client1 and client2 will appear to come from the same node. In addition, cluster topology methods may return ambiguous or useless results.

Web applications, however, can get a unique node ID even in the same JVM as long as the Terracotta Toolkit JAR is loaded by a classloader specific to the web application instead of a common classloader.

8 Terracotta Cluster Tools

Cluster tools provide control, visibility, security, and management capabilities for setting up, maintaining, and troubleshooting a Terracotta cluster.

8.1 Terracotta Developer Console

The Terracotta Developer Console delivers a full-featured monitoring and diagnostics tool aimed at assisting the development and testing phases of an application clustered with Terracotta. Use the Developer Console to isolate issues, discover tuning opportunities, observe application behavior under clustering, and learn how the cluster holds up under load and failure conditions.

The console functions as a JMX client with a graphical user interface. It must be able to connect to the JMX ports of server instances in the target cluster.

Enterprise versions of Terracotta also include the Terracotta Operations Center, a GUI operator's console offering features such as backups of shared data, client disconnect, and server shutdown controls. To learn more about the many benefits of an enterprise version of Terracotta, see the [Terracotta products page](#).

Using the developer console, you can perform the following tasks:

- Monitor and manage clustered applications using Ehcache, Ehcache for Hibernate, Quartz, and Sessions.
- View all cluster events in a single window.
- View cluster-wide statistics.
- Trigger distributed garbage-collection operations.
- Monitor the health of servers and clients under changing conditions.
- Record cluster statistics for later display.
- Receive console status and server log messages in a console window.
- Discover which classes are being shared in the cluster.

These and other console features are described below.

8.1.1 Launching the Terracotta Developer Console

You can launch the Terracotta Developer Console from a command line.

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh&
```

Microsoft Windows

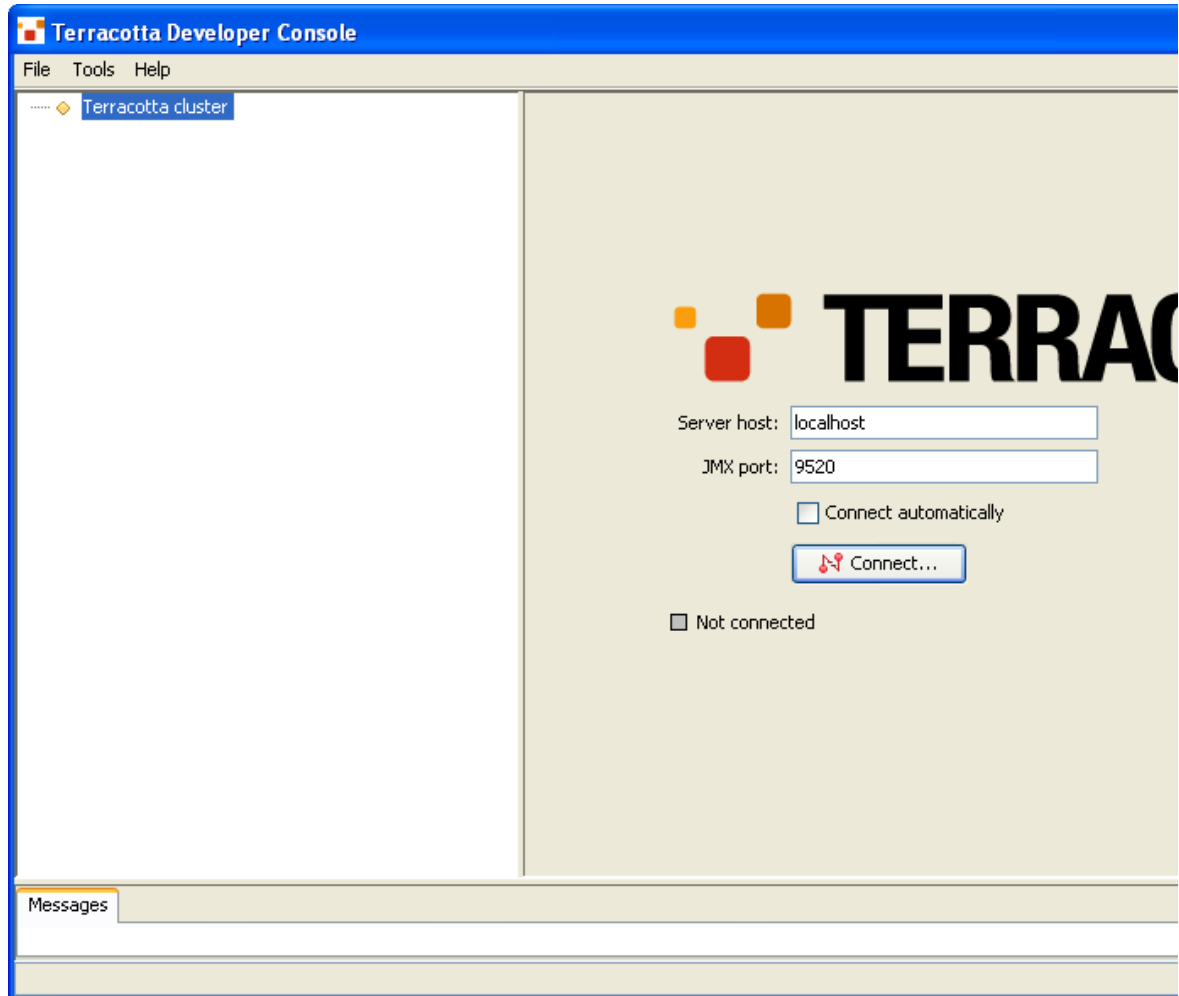
```
[PROMPT] %TERRACOTTA_HOME%\bin\dev-console.bat
```

TIP: Console Startup

When the console first starts, it waits until every Terracotta server configured to be active has reached [active status](#) before fully connecting to the cluster. The console does not wait for standby (or passive) servers to complete startup.

8.1.1a The Console Interface

When not connected to a server, the console displays a connect/disconnect panel, message-log window, status line, and an inactive cluster node in the clusters panel.



The cluster list in the clusters panel could already be populated because of pre-existing references to previously defined Terracotta clusters. These references are maintained as Java properties and persist across sessions and product upgrades. If no clusters have been defined, a default cluster (host=localhost, jmx-port=9520) is created.

The JMX port is set in each server's <server> block in the Terracotta configuration file:

```
<server host="host1" name="server1">
...
  <jmx-port>9521</jmx-port>
...
</server>
```

To learn more about setting JMX ports, see the [Configuration Guide and Reference](#) .

Once the console is connected to a cluster, the cluster node in the clusters panel serves as the root of an expandable/collapsible tree with nested displays and controls. One cluster node appears for each cluster you connect to.

8.1.1b Console Messages

Click **Messages** in the Status panel to view messages from the console about its operations.

8.1.1c Menus

The following menus are available from the console's menu bar.

File

- New Cluster - Create a new cluster node in the clusters panel.
- Quit - Shut down the console application. Has no effect on the cluster except to reduce load if the console has been recording statistics or profiling locks.



Tools

- Show SVT - Opens the Terracotta [Snapshot Visualization Tool](#).
- Options - Opens the Options dialog box (see [Runtime Statistics](#)).

Help

- Developer Console Help - Go to the Developer Console documentation.
- Visit Terracotta Forums - Go to the community forums to post questions and search for topics.
- Contact Terracotta Technical Support - Go to the contact page for Terracotta technical support.
- Check for Updates - Automatically check for updates to Terracotta.
- Check Server Version - Automatically check for console-server version mismatch.
- About Terracotta Developer Console - Display information on Terracotta and the local host.

8.1.1d Context-Sensitive Help

Context-sensitive help is available wherever  (help button) appears in the Terracotta Developer Console. Click  in a console panel to open a web-browser page containing help on the features in that panel.

8.1.1e Context Menus

Some console features have a context menu accessed by right-clicking the feature. For example, to open a context menu for creating a new cluster root in the clusters panel, right-click in the clusters panel.

8.1.2 Working with Clusters

Clusters are the highest-level nodes in the expandable cluster list displayed by the Terracotta Developer Console. A single Terracotta cluster defines a domain of Terracotta server instances and clients (application servers) being clustered by Terracotta. A single Terracotta cluster can have one or more servers and one or more clients. For example, two or more Terracotta servers configured as a server array, along with their clients, appear under the same cluster.

The **Cluster Panel** displays Terracotta application quick-view buttons as well as the cluster list. Click a quick-view button to go to a Terracotta application's panel, or click the name of the application under the My Application node. If an application is not running in the cluster, its quick-view button opens an informational window.

8.1.2a Adding and Removing Clusters

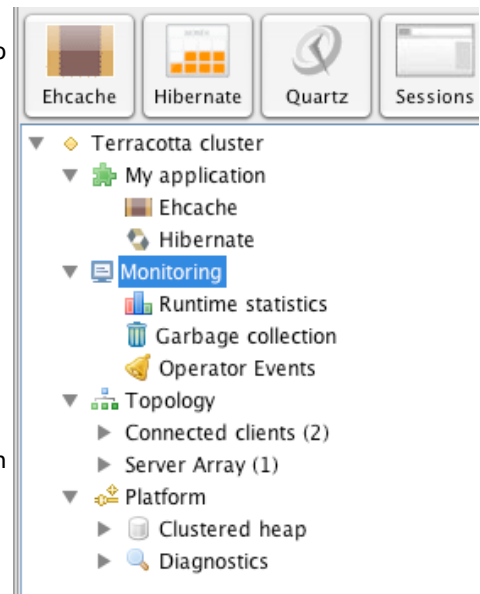
To add a new cluster reference, choose **New cluster** from the **File** or context menu.

The cluster topology is determined from the server specified in the connection panel's **Server Host** and **JMX Port** fields. These fields are editable when the console is not connected to the cluster.

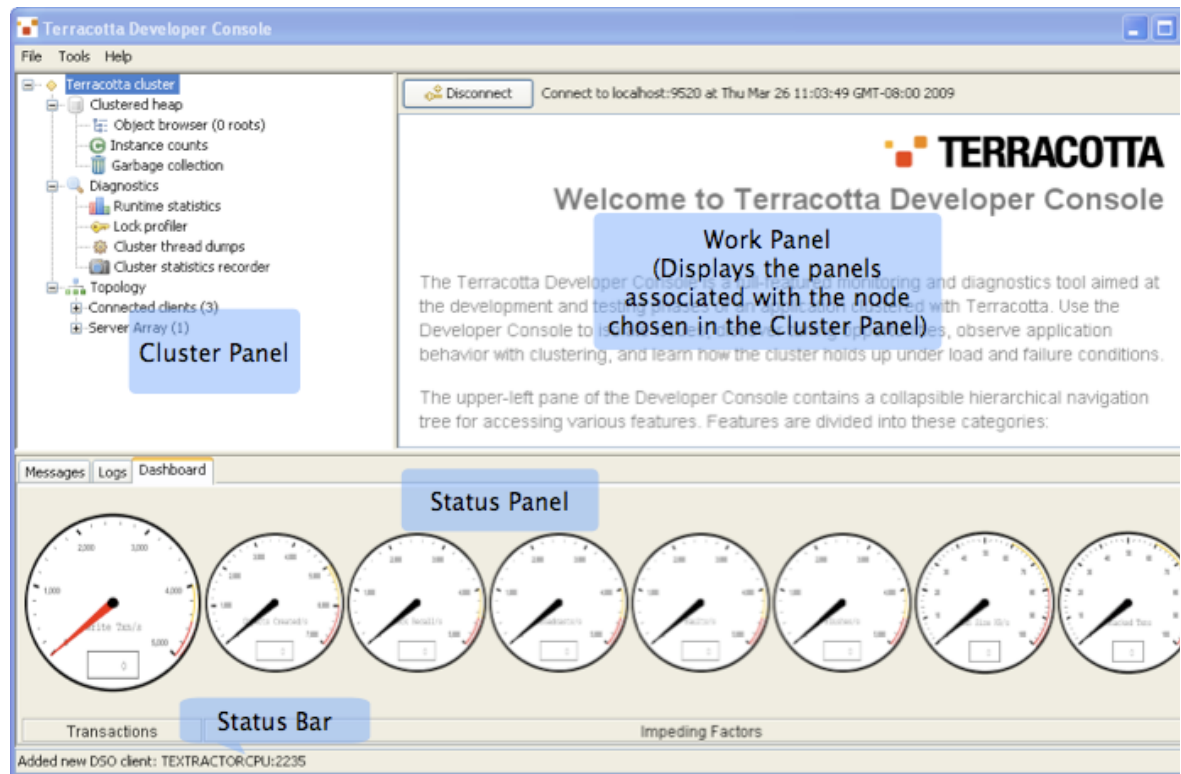
To remove an existing cluster reference, right-click the cluster in the cluster list to open the context menu, then choose **Delete**.

8.1.2b Connecting to a cluster

To connect to an existing cluster, select the cluster node in the cluster list, then click the **Connect** button in the connect/disconnect panel. You can also connect to a specific cluster by choosing **Connect** from its context menu. After a successful connection, the cluster node becomes expandable and a connection message



appears in the status bar.

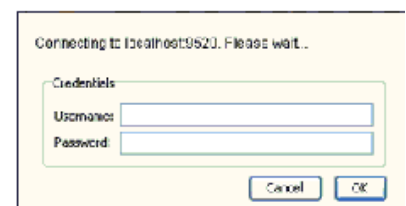


To automatically connect to a cluster whenever the Terracotta Developer Console starts or when at least one of the cluster's servers is running, enable **Auto-connect** in the cluster context menu. Automatic connections are attempted in the background and do not interfere with normal console operation.

8.1.2c Connecting to a Secured Cluster

A Terracotta cluster can be secured for JMX access, requiring authentication before access is granted. Connecting to a secured cluster prompts users to enter a username and password.

For instructions on how to secure your Terracotta cluster for JMX, see [Cluster Security](#).



8.1.2d Disconnecting from a Cluster

To disconnect from a cluster, select that cluster's node in the on the clusters panel and either click the **Disconnect** button above the help panel or select **Disconnect** from the cluster context menu.

8.1.3 Enterprise Ehcache Applications

If you are using Enterprise Ehcache with your application, the Ehcache views are available. These views offer the following features:

- Deep visibility into cached data
- Controls for enabling, disabling, and clearing all CacheManager caches
- Per-cache controls for enabling, disabling, clearing, and setting consistency
- Live statistics for the entire cluster, or per CacheManager, cache, or client
- Graphs with performance metrics for cache activity
- Historical data for trend analysis
- Parameters for control over the size and efficiency of cache regions
- A configuration generator

To access the Ehcache views, expand the **My application** node in the cluster navigation pane, then click the **Ehcache** node. If your cluster has more than one CacheManager, use the **Cache Manager** drop-down menu (available in all Ehcache panels) to choose the caches you want to view.

8.1.3a Overview Panel

The **Overview** panel lists all client nodes in the cluster running the CacheManager selected in the **Cache Manager** drop-down menu. Any operations, such as clearing cache content, performed on the caches or

nodes listed in this panel affect only caches belonging to the selected CacheManager.

Cache Manager: **ColorCache** has 2 clustered instances containing 1 total cache instances

Overview Performance Statistics

Manage Active Caches... Cache BulkLoading... Cache Statistics... Clear Cache Content

View by: ☒ CacheManager Instances ☐ Caches

Instances of CacheManager ColorCache

Node	Caches	Enabled	BulkLoad
10.2.0.133:57925	1	1	1
10.2.0.133:57936	1	1	1

Summary of CacheManager ColorCache on node 10.2.0.133:57936

Cache	Terracotta-clustered	Enabled	BulkLoad	Consistency
colors	✓	✓	✗	STRO

The nodes are listed in a summary table with the following columns:

- Node - The address of the client where the current CacheManager is running.
- Caches - The number of caches resident on the client.
- Enabled - The number of caches that are available to the application. Get operations will return data from an enabled cache or cause the cache to be updated with the missing data. Get operations return null from disabled caches, which are never updated.
- Bulkload - The number of caches whose data was loaded using the [Bulk-Load API](#).
- Statistics - The number of caches from which the console is gathering statistics. Caches with disabled statistics gathering do not appear in the **Performance** or **Statistics** panels and do not contribute to aggregate statistics. If all caches have statistics disabled, the **Performance** or **Statistics** panels cannot display any statistics.

Selecting a node displays a secondary table summarizing the caches resident on that node. The caches table has the following columns:

- Cache - The name of the cache.
- Terracotta-clustered - Indicates whether the cache is clustered (green checkmark) or not (red X).
- Enabled - Indicates whether the cache is available to the application (green checkmark) or not (red X). Get operations will return data from an enabled cache or cause the cache to be updated with the missing data. Get operations return null from disabled caches, which are never updated.
- Bulkload - Indicates whether the cache is in bulk-load mode (green checkmark) or not (red X). For more on bulk loading, see [2.2.3 Bulk-Load API](#).
- Consistency - Indicates what mode of data consistency the cache is in. Available modes are STRONG and EVENTUAL. For more on cache consistency, see [2.3.1d Terracotta Clustering Configuration Elements](#).
- Statistics - Indicates whether the console is collecting statistics from the cache (green checkmark) or not (red X). Caches with disabled statistics gathering do not appear in the **Performance** or **Statistics** panels and do not contribute to aggregate statistics. If all caches have statistics disabled, the **Performance** or **Statistics** panels cannot display any statistics.

TIP: Keeping Statistics On

By default, statistics are off for caches to improve performance. Each time you start the Terracotta Developer Console and connect to a client, the client's caches will have statistics off again even if you turned statistics on previously.

To change this behavior for a cache so that statistics remain on, use Ehcache configuration:

```
<cache name="myCache" ... statistics="true" >  
...  
</cache>
```

You can also set the view to list the selected CacheManager's caches in the summary table instead of the nodes. In this case, in its first column the secondary table lists the nodes on which the cache is resident.

TIP: Working With the Overview User Interface

- To work with a tree view of nodes, caches, and CacheManager instances, use the control buttons arranged at the top of the panel to open a dialog box.
- To save any changes you make in a dialog box, click **OK** . To discard changes, click **Cancel** .
- You can also select caches or nodes and use the context menu to perform operations.

Enable/Disable Caches

Click **Manage Active Caches** to open the **Manage Active Caches** window. This window gives you fine-grained control over enabling caches.

To enable (or disable) caches by CacheManager instances, choose **CacheManager Instances** (at the top of the **Manage Active Caches** window), then select (enable) or unselect (disable) from the hierarchy displayed.

To enable (or disable) caches by a specific cache, choose **Caches** (at the top of the **Manage Active Caches** window), then select (enable) or unselect (disable) from the hierarchy displayed.

To save any changes you make in this window, click **OK** . To discard changes, click **Cancel** .

You can also select caches or nodes and use the context menu to enable/disable the selected caches or all caches on the selected node.

Enable/Disable Cache Bulk Loading

Click **Cache Bulk Loading** to open the **Manage Bulk Load Mode** window. This window gives you fine-grained control over enabling cache bulk-load mode.

To enable (or disable) bulk loading by CacheManager instances, choose **CacheManager Instances** (at the top of the **Manage Bulk Load Mode** window), then select (enable) or unselect (disable) from the hierarchy displayed.

To enable (or disable) bulk loading for a specific cache, choose **Caches** (at the top of the **Manage Bulk Load Mode** window), then select (enable) or unselect (disable) from the hierarchy displayed.

To save any changes you make in this window, click **OK** . To discard changes, click **Cancel** .

You can also select caches or nodes and use the context menu to enable/disable coherence for the selected caches or for all caches on the selected node.

Enable/Disable Cache Statistics

Click **Cache Statistics** to open the **Manage Cache statistics** . This window gives you fine-grained control over enabling statistics gathering. To save any changes you make in this window, click **Enable Cache Statistics** . To discard changes, click **Cancel** .

Caches with disabled statistics gathering do not appear in the **Performance** or **Statistics** panels and do not contribute to aggregate statistics. If all caches have statistics disabled, the **Performance** or **Statistics** panels cannot display any statistics.

To enable (or disable) statistics gathering by CacheManager instances, choose **CacheManager Instances** (at the top of the **Manage Cache statistics window**), then select (enable) or unselect (disable) from the hierarchy displayed.

To enable (or disable) statistics gathering for a specific cache, choose **Caches** (at the top of the **Manage Cache statistics window**), then select (enable) or unselect (disable) from the hierarchy displayed.

You can also select caches or nodes and use the context menu to enable/disable statistics for the selected caches or for all caches on the selected node.

Clear Caches

Click **Clear Cache Contents** to open a dialog for clearing caches.

You can also clear caches using different context menus:

- To clear the data from all caches in node, select the node, then choose **Clear Caches** from the node's context menu.
- To clear the data from all caches in all nodes, select all nodes, then choose **Clear Caches** from the nodes context menu.
- To clear the data from a specific cache (or caches) in a node, select the node, select the cache (or caches) in the cache summary table, then choose **Clear Caches** from the cache's context menu.

Cache Configuration

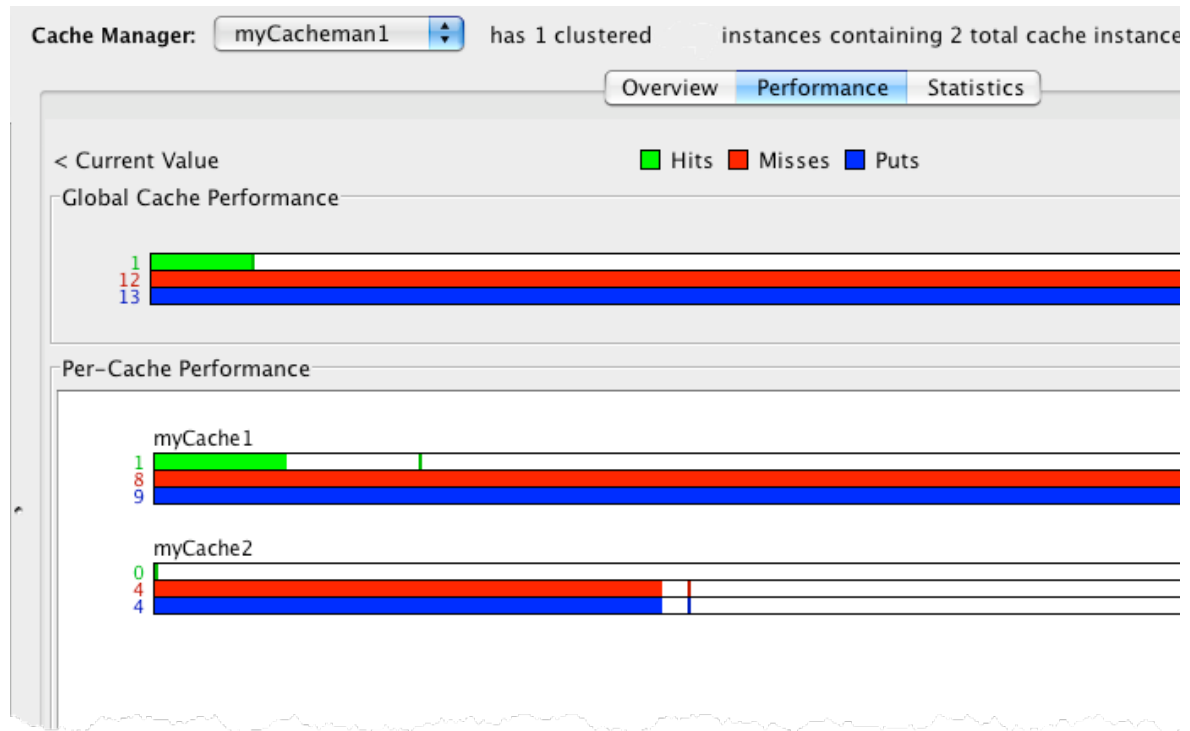
Click **Cache Configuration** to open a dialog with editable cache configuration. See [8.1.3d Editing Cache Configuration](#) for more information on editing cache configurations in the Terracotta Developer Console.

To view the Ehcache in-memory configuration file of a node, select the node, then choose **Show Configuration** from the context menu.

To view the Ehcache in-memory configuration of a cache, select the cache, then choose **Show Configuration** from the context menu.

8.1.3b Performance Panel

The **Performance** panel displays real-time performance statistics for both **Global Cache Performance** (aggregated from all caches) and **Per-Cache Performance**. The **Performance** panel is useful for viewing current activity in clustered caches.



Performance statistics are displayed as color-coded bar graphs with current values shown on the left end and "high-water" (current maximum) values shown on the right end:

- Hits - (Green) Counts get operations that return data from the cache.
- Puts - (Blue) Counts each new (or updated) element added to the cache.
- Misses - (Red) Counts each cache miss; each miss causes data fault-in from outside the cache.

TIP: The Relationship of Puts to Misses

The number of puts can be greater than the number of misses because updates are counted as puts. For more information on how cache events are configured and handled, see [2.3.1f Cache Events Configuration](#).

If the **Performance** panel is selected and statistics gathering is disabled for all caches, a warning dialog appears. This dialog offers three choices:

- Click **OK** to enable statistics gathering for all caches.
- Click **Advanced** to open the **Manage Statistics** window and set statistics gathering for individual caches (see the [Overview Panel](#) for more information).
- Click **Cancel** to leave statistics gathering off.

NOTE: Statistics and Performance

Gathering statistics may have a negative impact on overall cache performance.

8.1.3c Statistics Panel

The **Statistics** panel displays cache statistics in history graphs and a table. The graphs are useful for recognizing trends, while the table presents a snapshot of statistics and can display ordered high-low lists based on the counts shown.

If the **Performance** panel is selected and statistics gathering is disabled for all caches, a warning dialog appears. This dialog offers three choices:

- Click **OK** to enable statistics gathering for all caches.
- Click **Advanced** to open the **Manage Statistics** window and set statistics gathering for individual caches (see the [Overview Panel](#) for more information).
- Click **Cancel** to leave statistics gathering off.

NOTE: Statistics and Performance

Gathering statistics may have a negative impact on overall cache performance.

Some of the main tasks you can perform in this panel are:

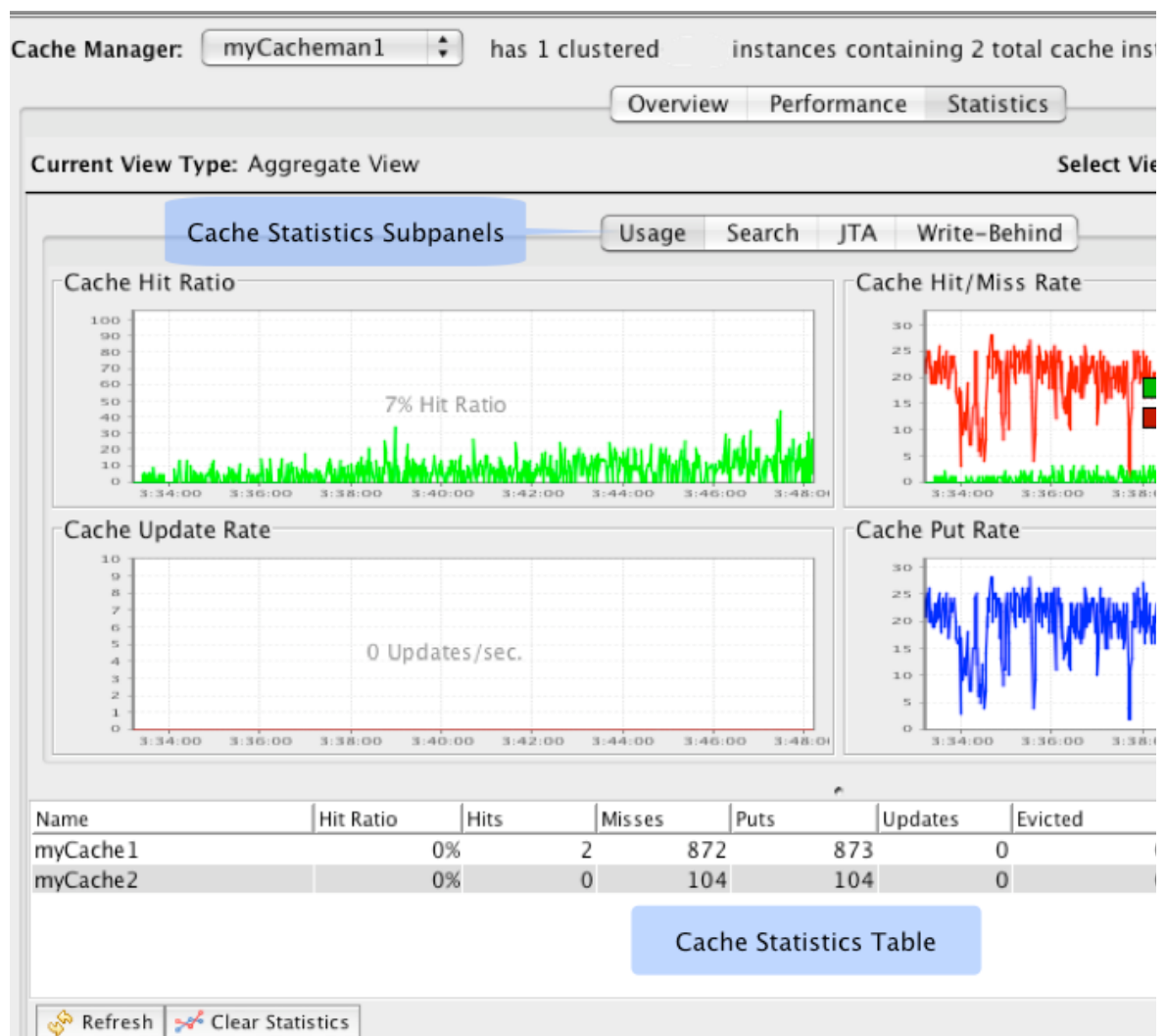
- View cache statistics for the entire cluster.
- View cache statistics for each Terracotta client (application server).

Cache statistics are sampled at the rate determined by the rate set in the **Options** dialog box (see [Runtime Statistics](#)).

Use the following controls to control the statistics:

- **Select View** - Set the scope of the statistics display using the **Select View** menu. To view statistics for the entire cluster, select **Cluster stats**. To view statistics for an individual Terracotta client (application server), choose that client from the **Per Client View** submenu.
- **Subpanel** - Click the button for the subpanel you want to view (Usage, Search, JTA, or Write-Behind).
- **Refresh** - Click **Refresh** to force the console to immediately poll for statistics. A refresh is executed automatically each time a new view is chosen from the **Select View** menu.
- **Clear** - Click **Clear Statistics** to wipe the current display of statistics. Restarts the recording of statistics (zero out all values).

Cache Statistics Usage Graphs



The line graphs available display the following statistics over time:

- **Cache Hit Ratio** - The ratio of cache hits to get attempts. A ratio of 1.00 means that all requested data was obtained from the cache (every put was a hit). A low ratio (closer to 0.00) implies a higher number of misses that result in more faulting of data from outside the cache.
- **Cache Hit/Miss Rate** - The number of cache hits per second (green) and the number of cache misses per second (red). Current values are overlaid on the graph. An effective cache shows a high number of hits relative to misses.
- **Cache Update Rate** - The number of updates to elements in the cache, per second. The current value is overlaid on the graph. A high number of updates implies a high eviction rate or rapidly changing data.
- **Cache Put Rate** - The number of cache puts executed per second. The current value is overlaid on the graph. The number of puts always equals or exceeds the number of misses, since every miss leads to a put. In addition, updates are also counted as puts. Efficient caches have a low overall put rate.

Cache Statistics Table

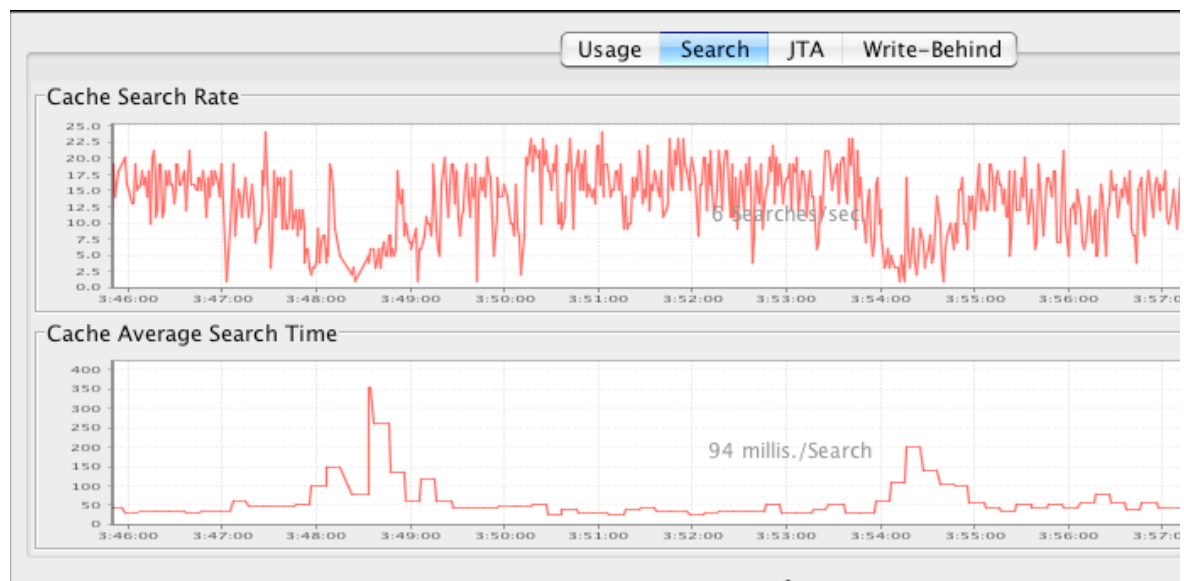
The cache statistics table displays a snapshot of the following statistics for each cache:

- **Name** - The name of the cache as it is configured in the CacheManager configuration resource.
- **Hit Ratio** - The aggregate ratio of hits to gets.
- **Hits** - The total number of successful data requests.
- **Misses** - The total number of unsuccessful data requests.
- **Puts** - The total number of new (or updated) elements added to the cache.
- **Updates** - The total number of updates made to elements in the cache.
- **Expired** - The total number of expired cache elements.
- **Removed** - The total number of evicted cache elements.
- **In-Memory Size** - The total number of elements in the cache on the client selected in **Select View** . This statistic is not available in the cluster-wide view.
- **On-disk Size** - The total number of elements in the cache. Even when a client is selected in **Select View** , this statistic always displays the cluster-wide total.

TIP: Working With the Statistics Table

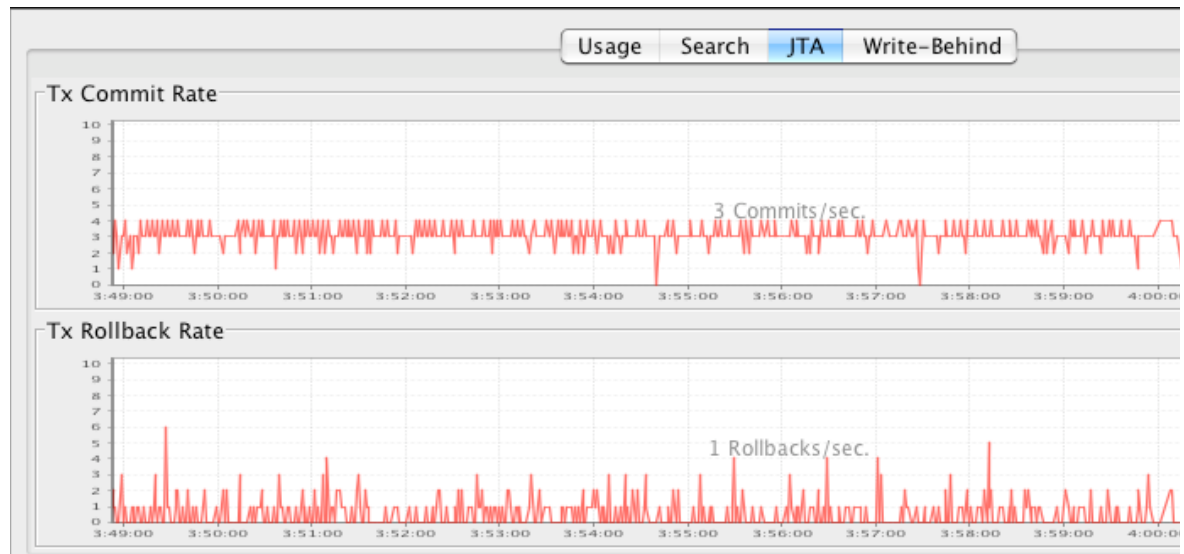
- The snapshot is refreshed each time you display the **Statistics** panel. To manually refresh the table, click **Refresh** .
- The fully qualified name of a cache shown in a table may be abbreviated. You can view the unabbreviated name in a tooltip by placing the mouse pointer over the abbreviated name. You can also view the full name of a cache by expanding the width of the **Name** column.
- You can view the total sum of a column of numbers (such as **Misses** or **Puts**) in a tooltip by placing the mouse pointer anywhere in the column.
- To order a table along the values of any column, double-click its heading. An arrow appears in the column heading to indicate the direction of the order. You can reverse the order by double-clicking the column head again.

Cache Statistics Search Graphs



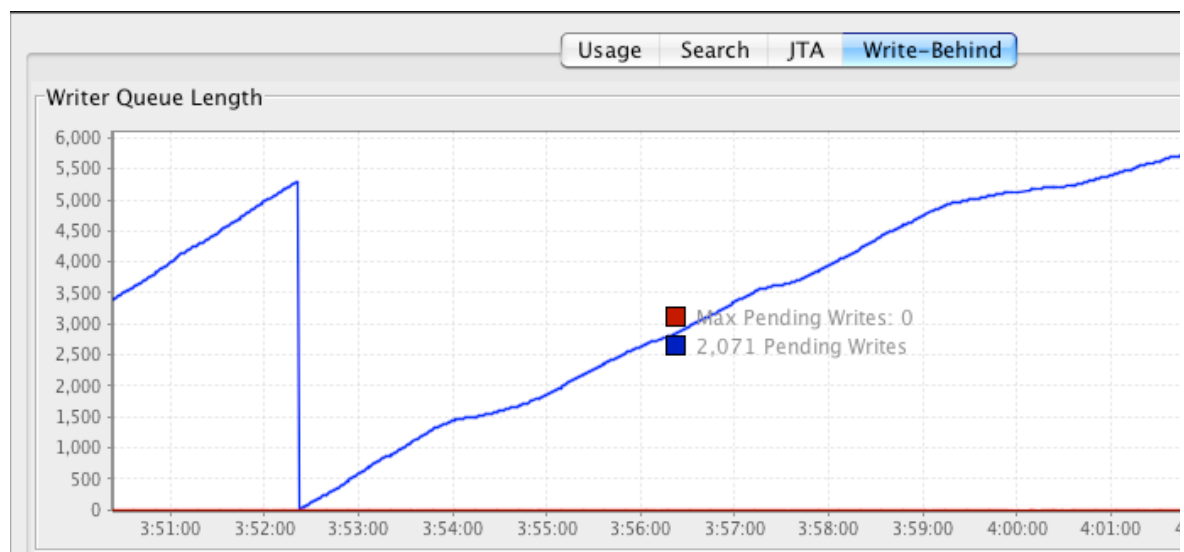
The search-related historical graphs provide a view into how quickly cache searches are being performed. The search-rate graph displays how many searches per second are being executed. The current values for these metrics are also displayed. The search-time graph displays how long each search operation takes. A correlation between how long searches are taking and how many are executed may be seen over time.

Cache Statistics JTA Graphs



The JTA historical graphs display the transaction commit and rollback rates as well as the current values for those rates. For more information about transactional caches, see [2.3.8 Working With Transactional Caches](#).

Cache Statistics Write-Behind Graphs



The Write-Behind historical graph displays the total number of writes in the write-behind queue or queues (blue line), as well as the current value. The graph also displays the maximum number of pending writes, or the number of elements that can be stored in the queue while waiting to be processed (red line). This value is derived from the `<cacheWriter />` attribute `writeBehindMaxQueueSize`. Note that a value of zero ("0") sets no limit on the number of elements that can be in the queue. For more information on the write-behind queue, see [2.2.7 Write-Behind Queue in Enterprise Ehcache](#).

8.1.3d Editing Cache Configuration

In the **Overview** panel, click **Cache Configuration** to open the **Manage Cache Configuration** dialog. The dialog displays a table of existing clustered and unclustered caches with storage and eviction properties. The configuration shown is loaded from the initial configuration resource. For example, if the **CacheManager** is initialized with a configuration file, the values from that file appear in **Configuration** panel.

The cache-configuration tables display the following editable configuration properties for each cache:

- **Cache** - The name of the cache as it is configured in the **CacheManager** configuration resource. Since

unclustered caches, also called *standalone* caches, are local only, a drop-down menu allowing you to select the standalone caches CacheManager is provided for their table.

- **Max Memory Elements** - The maximum number of elements allowed in the cache in any one client (any one application server). If this target is exceeded, eviction occurs to bring the count within the allowed target. 0 means no eviction takes place (infinite size is allowed).
- **Max Disk Elements** - The maximum total number of elements allowed in the cache in all clients (all application servers). If this target is exceeded, eviction occurs to bring the count within the allowed target. 0 means no eviction takes place (infinite size is allowed).
- **Time-To-Idle (TTI)** - The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. 0 means no TTI eviction takes place (infinite lifetime).
- **Time-To-Live (TTL)** - The maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from the cache. 0 means no TTL eviction takes place (infinite lifetime).

NOTE: Setting Eviction in Caches

Having the values of TTI, TTL, Max Memory Elements, and Max Disk Elements all set to 0 for a cache in effect *turns off* all eviction for that cache. Unless you want cache elements to *never* be evicted from a cache, you should set these properties to non-zero values that are optimal for your use case.

To edit a configuration property, click the field holding the value for that property, then type a new value. Changes are not saved to the cache configuration file and are not persisted beyond the lifetime of the CacheManager.

To create a configuration file based on the configuration shown in the panel, select a node in the **Overview** panel and choose **Show Configuration** to open a window containing a complete Ehcache configuration file. Copy this configuration and save it to a configuration file loaded by the CacheManager (for example, `ehcache.xml`).

To get the configuration for a single cache, select the cache in the **Overview** panel and choose **Show Configuration** to open a window containing the cache's configuration.

For more information on the Enterprise Ehcache configuration file, see [Ehcache Configuration File](#).

8.1.4 Enterprise Ehcache for Hibernate Applications

If you are using Enterprise Ehcache with your Hibernate-based application, the **Hibernate** and second-level cache views are available. These views offer you the following:

- Deep visibility into Hibernate and cached data
- Live statistics
- Graphs, including puts and misses
- Historical data for trend analysis
- Parameters for control over the size and efficiency of cache regions
- A configuration generator

To access the **Hibernate** and second-level cache views, expand the **My application** node in the cluster navigation pane, then click the **Hibernate** node.

NOTE: Statistics and Performance

Each time you connect to the Terracotta cluster with the Developer Console, Hibernate and cache statistics gathering is automatically started. Since this may have a negative impact on performance, consider disabling statistics gathering during performance tests and in production. To disable statistics gathering, navigate to the **Overview** panel in the second-level cache view, then click **Disable Statistics**.

Use the view buttons to choose **Hibernate** (Hibernate statistics) or **Second-Level Cache** (second-level cache statistics and controls).

TIP: Working With the User Interface

The following are productivity tips for using the Developer Console:

- The fully qualified name of a region, entity, or collection shown in a table may be abbreviated. You can view the unabbreviated name in a tooltip by placing the mouse pointer over the abbreviated name. Note that expanding the width of a column does not undo abbreviations.
- Queries are not abbreviated, but can still appear to be cut off by columns that are too narrow. To view the full query string, you can expand the column or view the full query string in a tooltip by placing the mouse pointer over the cut-off query string.
- You can view the total sum of a column of numbers (such as Hibernate Inserts) in a tooltip by placing the mouse pointer anywhere in the column.
- To order a table along the values of any column, double-click its heading. An arrow appears in the column heading to indicate the direction of the order. You can reverse the order by double-clicking the column head again.
- Some panels have a **Clear All Statistics** button. Clicking this button clears statistics from the current panel and all other Hibernate and cache panels that display statistics.
- If your cluster has more than one second-level cache, use the **Persistence Unit** drop-down menu (available in all panels) to choose the cache you want to view.

Hibernate View

Click **Hibernate** view to display a table of Hibernate statistics. Some of the main tasks you can perform in this view are:

- View statistics for the entire cluster on Hibernate entities, collections, and queries.
- View statistics for each Terracotta client (application server) on Hibernate entities, collections, and queries.

Hibernate statistics are sampled at the rate determined by the rate set in the Options dialog box (see [Runtime Statistics](#)). Use the controls that appear below the statistics table to update the statistics:

- Refresh - Click **Refresh** to force the console to immediately poll for statistics.
- Clear - Click **Clear All Statistics** to wipe the current display of statistics.

You can set the scope of the statistics display using the **Select View** menu. To view statistics for the entire cluster, select **Cluster stats**. To view statistics for an individual Terracotta client (application server), choose that client from the **Per Client View** submenu.

Entities

Click **Entities** to view the following standard Hibernate statistics on Hibernate entities in your application:

- Name
- Loads
- Updates
- Inserts
- Deletes
- Fetches
- Optimistic Failures

Collections

Click **Collections** to view the following standard Hibernate statistics on Hibernate collections in your application:

- Role
- Loads
- Fetches
- Updates
- Removes
- Recreates

Queries

Click **Queries** to view the following standard Hibernate statistics on Hibernate queries in your application:

- Query
- Executions
- Rows
- Avg Time (Average Time)
- Max Time (Maximum Time)
- Min Time (Minimum Time)

8.1.4a Second-Level Cache View

The second-level cache view provides performance statistics and includes per-region cache configuration. Some of the main tasks you can perform in this view are:

- View both live and historical performance metrics in graphs.
- Enable and disable cache regions.
- Flush cache regions.
- Set eviction parameters per region.
- Generate a configuration file based on settings in the Second-Level Cache view.

Overview

The **Overview** panel displays the following real-time performance statistics for both Global Cache Performance (covering the entire cache) and Per-Region Cache Performance:

- Hits - (Green) Counts queries that return data from the second-level cache.
- Puts - (Blue) Counts each new (or updated) element added to the cache.
- Misses - (Red) Counts each cache miss; each miss causes data fault-in from the database.

TIP: The Relationship of Puts to Misses

The number of puts can be greater than the number of misses because updates are counted as puts.

These statistics are displayed as color-coded bar graphs with current values shown on the left end and "high-water" (current maximum) values shown on the right end.

The **Overview** panel provides the following controls:

- **Enable All Cache Regions** - Turns on all configured cache regions.
- **Disable All Cache Regions** - Turns off all configured cache regions shown.
- **Evict All Entries** - Removes all entries from the cache (clears the cache).
- **Disable Statistics** - Turns off the gathering of statistics. Only **DB SQL Execution Rate** (queries per second), **In-Memory Count**, and **Total Count** of cache elements continue to function. When statistics gathering is off, this button is called **Enable Statistics**. *Gathering statistics may have a negative impact on performance.*
- **Clear All Statistics** - Restart the recording of statistics (zero out all values).

Statistics

The **Statistics** panel displays second-level cache statistics in history graphs and a table. The graphs are useful for recognizing trends, while the table presents a snapshot of statistics.

Some of the main tasks you can perform in this panel are:

- View cache statistics for the entire cluster.
- View cache statistics for each Terracotta client (application server).

Cache statistics are sampled at the rate determined by the rate set in the Options dialog box (see [Runtime Statistics](#)). Use the controls that appear below the statistics table to update the statistics:

- **Refresh** - Click **Refresh** to force the console to immediately poll for statistics.
- **Clear** - Click **Clear All Statistics** to wipe the current display of statistics.

You can set the scope of the statistics display using the **Select View** menu. To view statistics for the entire cluster, select **Cluster stats**. To view statistics for an individual Terracotta client (application server), choose that client from the **Per Client View** submenu.

Cache Statistics Graphs

The line graphs available display the following statistics over time:

- **Cache Hit Ratio** - The ratio of cache hits to queries. A ratio of 1.00 means that all queried data was obtained from the cache. A low ratio (closer to 0.00) implies a higher number of misses that result in more faulting of data from the database.
- **Cache Hit/Miss Rate** - The number of cache hits per second (green) and the number of cache misses per second (red). Current values are overlaid on the graph.
- **DB SQL Execution Rate** - The number of queries executed per second. The current value is overlaid on the graph.
- **Cache Put Rate** - The number of cache puts executed per second. The number of puts always equals or exceeds the number of misses, since every miss leads to a put. The current value is overlaid on the graph.

Cache Statistics Table

The cache statistics table displays a snapshot of the following statistics for each region:

- **Region** - The fully qualified name of the region (abbreviated).
- **Hit Ratio** - The aggregate ratio of hits to queries.
- **Hits** - The total number of successful queries on the cache.
- **Misses** - The total number of unsuccessful queries on the cache.
- **Puts** - The total number of new (or updated) elements added to the cache.
- **In-Memory Count** - The total number of cache elements in the region on the client selected in **Select View**. This statistic is not available in the cluster-wide view.
- **Total Count** - The total number of cache elements in the region. Even when a client is selected in **Select View**, this statistic always displays the cluster-wide total.
- **Hit Latency** - The time (in milliseconds) it takes to find an element in the cache. Long latency times may indicate that the cache element is not available locally and is being faulted from the Terracotta server.
- **Load Latency** - The time (in milliseconds) it takes to load an entity from the database after a cache miss.

The snapshot is refreshed each time you display the **Statistics** panel. To manually refresh the table, click **Refresh**.

Configuration

The **Configuration** panel displays a table with the following configuration properties for each cache region:

- **Region** - The fully qualified name of the region (abbreviated).
- **Cached** - Whether the region is currently being cached ("On") or not ("Off").
- **TTI (Time to idle)** - The maximum number of seconds an element can exist in the cache without being accessed. The element expires at this limit and will no longer be returned from the cache. 0 means no TTI eviction takes place (infinite lifetime).
- **TTL (Time to live)** - The maximum number of seconds an element can exist in the cache regardless of use. The element expires at this limit and will no longer be returned from the cache. 0 means no TTL eviction takes place (infinite lifetime).
- **Target Max In-Memory Count** - The maximum number of elements allowed in a region in any one client (any one application server). If this target is exceeded, eviction occurs to bring the count within the allowed target. 0 means no eviction takes place (infinite size is allowed).
- **Target Max Total Count** - The maximum total number of elements allowed for a region in all clients (all application servers). If this target is exceeded, eviction occurs to bring the count within the allowed target. 0 means no eviction takes place (infinite size is allowed).

NOTE: Setting Eviction in Caches

Having the values of TTI, TTL, Max Memory Elements, and Max Disk Elements all set to 0 for a cache in effect *turns off* all eviction for that cache. Unless you want cache elements to *never* be evicted from a cache, you should set these properties to non-zero values that are optimal for your use case.

Configuration is loaded from the initial configuration resource. For example, if the second-level cache is initialized with a configuration file, the values from that file appear in **Configuration** panel.

Region Operations

To stop a region from being cached, select that region in the configuration table, then click **Disable Region**. Disabled regions display "Off" in the configuration table's **Cached** column. Queries for elements that would be cached in the region must go to the database to return the desired data.

To return a region to being cached, select that region in the configuration table, then click **Enable Region**. Disabled regions display "On" in the configuration table's **Cached** column.

To clear a region, select that region in the configuration table, then click **Evict All Entries in Cache**. This operation removes from all clients all of the entries that were cached in that region.

If you are troubleshooting or otherwise require more detailed visibility into the workings of the second-level cache, enable **Logging enabled**.

Region Settings

To change the configuration for a region, select that region in the configuration table, then change the values in the fields provided:

- Time to idle
- Time to live
- Target max total count
- Target max in-memory count

You can also turn logging on for the region by selecting **Logging enabled**.

The settings you change in the second-level cache view are not saved to the cache configuration file and are not persisted beyond the lifetime of the cache. To create a configuration file based on the configuration shown in the second-level cache view, click **Generate Cache Configuration...** to open a window containing a complete configuration file. Copy this configuration and save it to a configuration file loaded by the used for configuring the second-level cache (such as the default `ehcache.xml`). For more information on Enterprise Ehcache configuration file, see [Ehcache Configuration File](#).

8.1.5 Clustered Quartz Scheduler Applications

The Terracotta JobStore for Quartz Scheduler clusters the Quartz job-scheduling service. If you are clustering Quartz Scheduler, the Quartz view is available. The Quartz view offers the following features:

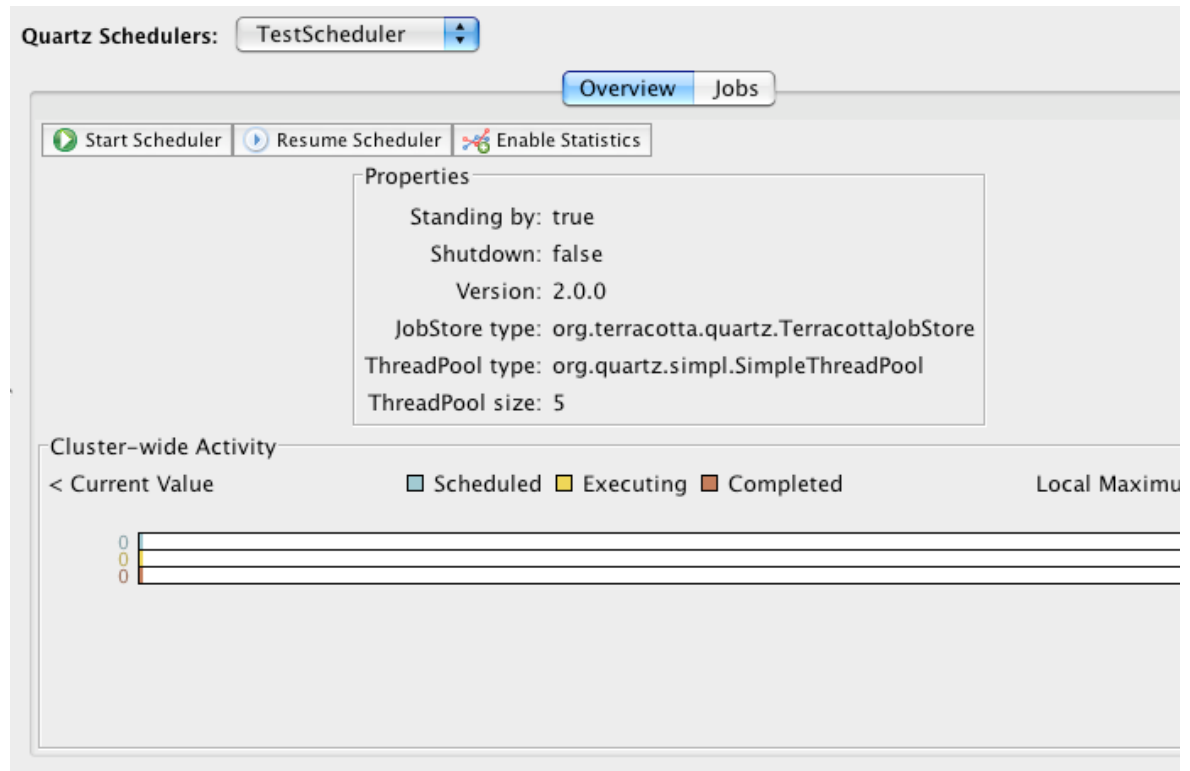
- Activity meters
- Start/stop and pause controls for schedulers, job groups, and jobs
- Job execution history

- Ability to delete individual job details

To access the **Quartz** view, expand the **My application** node in the cluster navigation pane, then click the **Quartz** node.

Choose the scheduler to display from the **Quartz Schedulers** menu, available at the top of any of the Quartz view's panels. For the currently chosen scheduler, the **Quartz** view provides the following panels:

Overview



The **Overview** panel displays the following real-time cluster-wide activity meters:

- Scheduled - (Green) Shows count of new jobs scheduled at the time of polling.
- Executing - (Yellow) Shows count of currently executing jobs.
- Completed - (Red) Shows count of jobs that have completed at the time of polling.

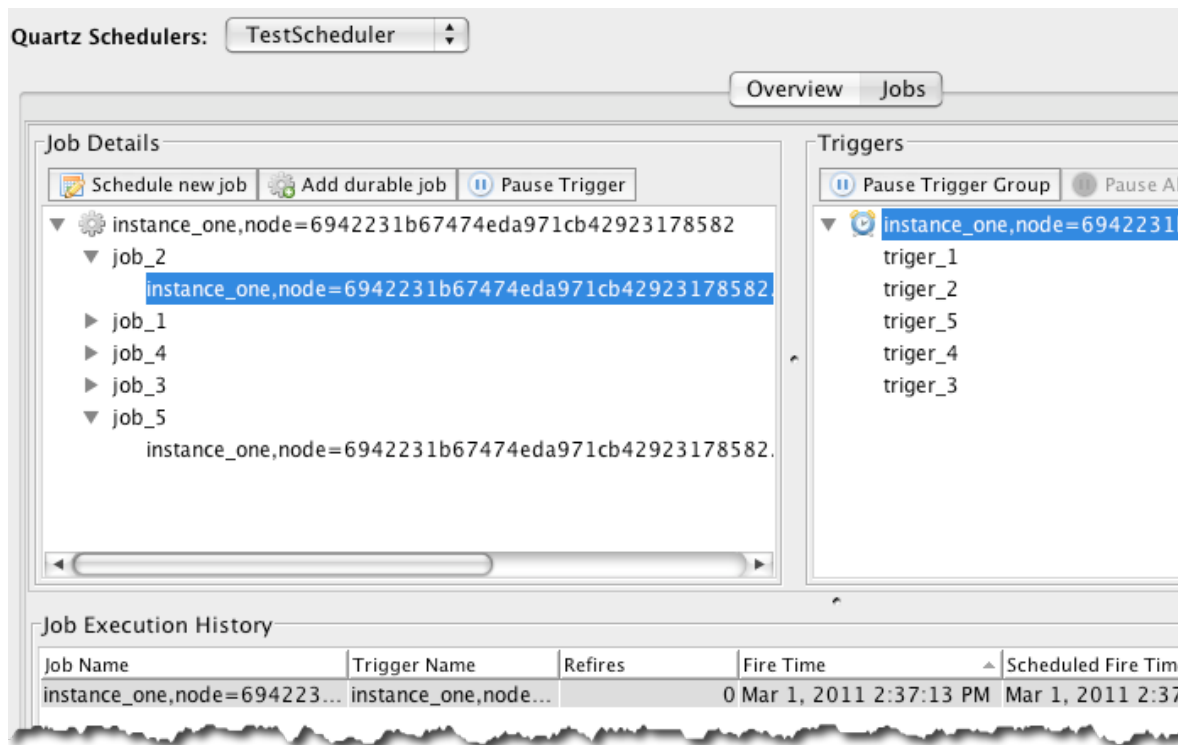
These counts are displayed as color-coded bar graphs with current values shown on the left end and "high-water" (current maximum) values shown on the right end. If statistics are disabled, no counts are displayed on the activity meters (see below for how to enable statistics).

The **Overview** panel provides the following controls:

- Stop Scheduler - Stop (destroy) the currently selected scheduler. The scheduler cannot be restarted.
- Pause Scheduler - Suspend the scheduler from any activity. When a scheduler is paused, click Resume Scheduler to restart it.
- Disable Statistics - Turns off the gathering of statistics. When statistics gathering is off, this button is called **Enable Statistics**. *Gathering statistics may have a negative impact on performance.*

The panel also displays a summary of properties for the selected scheduler.

Jobs



The **Jobs** panel displays information about the jobs and triggers managed by the selected scheduler. The **Jobs** panel is composed of the following:

Job Details Subpanel

Contains an expandable/collapsible list of job groups. When expanded, each job-group node lists its jobs, and each job lists its triggers when it's expanded. Selecting a node displays context-sensitive controls (buttons) along the top of the subpanel:

- Group selected - **Pause Job Group** (**Resume Job Group**). These controls also appear in the node's context menu.
- Job selected - **Delete Job** , **Schedule Job** . These controls also appear in the node's context menu along with **Pause Job** (**Resume Job**), **Trigger Job** .
- Trigger selected - **Pause Trigger** (**Resume Trigger**). These controls, along with **Pause All Triggers** (pauses all triggers for the job) and **Unschedule Job** (removes the trigger from the job), also appear in the node's context menu.

Triggers subpanel

Contains an expandable/collapsible list of trigger groups. When expanded, each trigger-group node lists its triggers. The control **Pause All Triggers** appears along the top of the panel and will pause all triggers in the selected trigger group.

Selecting a node displays additional context-sensitive controls (buttons) along the top of the subpanel:

- Trigger group selected - **Pause Trigger Group** (**Resume Trigger Group**). These controls also appear in the node's context menu.
- Trigger selected - **Pause Trigger** (**Resume Trigger**). These controls also appear in the node's context menu.

Job Execution History table

The Job Execution History table lists the jobs that have been run by the selected scheduler. The table shows the following data:

- Job Name
- Trigger Name
- Refires (number of times job has been refired)
- Fire Time
- Scheduled Fire Time
- Previous Fire (time)
- Next Fire (time)
- Job Completion Time (milliseconds)

The table can be cleared by selecting **Clear** from its context (right-click) menu.

8.1.6 Clustered HTTP Sessions Applications

If you are clustering Web sessions, the HTTP sessions view is available. The sessions view offers the following features:

- Live statistics
- Graphs, including session creation, hop, and destruction rates
- Historical data for trend analysis
- Ability to expire any individual session or all sessions

To access the **HTTP Sessions** view, expand the **My application** node in the cluster navigation pane, then click the **HTTP Sessions** node.

NOTE: Statistics and Performance

Each time you connect to the Terracotta cluster with the Developer Console, statistics gathering is automatically started. Since this may have a negative impact on performance, consider disabling statistics gathering during performance tests and in production. To disable statistics gathering, navigate to the **Overview** or **Runtime Statistics** panel, then click **Disable Statistics**. The button's name changes to **Enable Statistics**.

The **HTTP Sessions** view has the following panels:

Overview

The **Overview** panel displays the following real-time performance statistics:

- Sessions Created - (Green) Counts new sessions.
- Sessions Hopped - (Yellow) Counts each time a session changes to another application server.
- Sessions Destroyed - (Red) Counts each time a session is torn down.

These statistics are displayed as color-coded bar graphs with current values shown on the left end and "high-water" (current maximum) values shown on the right end.

The **Overview** panel provides the following controls:

- Expire All Sessions - Close all sessions.
- Disable Statistics - Turns off the gathering of statistics. When statistics gathering is off, this button is called **Enable Statistics**. *Gathering statistics may have a negative impact on performance.*

Statistics

The **Statistics** panel displays session statistics in history graphs and a table. The graphs are useful for recognizing trends, while the table presents a snapshot of statistics.

Some of the main tasks you can perform in this panel are:

- View session statistics for the entire cluster.
- View session statistics for each Terracotta client (application server).

Session statistics are sampled at the rate determined by the rate set in the Options dialog box (see [Runtime Statistics](#)). Use the controls that appear below the statistics table to update the statistics:

- Refresh - Click **Refresh** to force the console to immediately poll for statistics.
- Clear - Click **Clear All Statistics** to wipe the current display of statistics.

You can set the scope of the statistics display using the **Select View** menu. To view statistics for the entire cluster, select **Cluster stats**. To view statistics for an individual Terracotta client (application server), choose that client from the **Per Client View** submenu.

session Statistics Graphs

The line graphs available display the following statistics over time:

- **Active Sessions** - The number of active sessions. The current total is overlaid on the graph.
- **Session Creation Rate** - The number of sessions being created per second. The current rate is overlaid on the graph.
- **Session Destruction Rate** - The number of sessions being torn down per second. The current rate is overlaid on the graph.
- **Session Creation Rate** - The number of sessions that have changed application servers, per second. The current rate is overlaid on the graph.

Session Statistics Table

The session statistics table displays a snapshot of the following statistics for each clustered application:

- App - The hostname and application context.
- Active Sessions - The aggregate number of active sessions.
- Sessions Created - The total number of sessions created.
- Sessions Destroyed - The total number of sessions destroyed.
- Sessions Hopped - The total number of sessions hopped.

The snapshot is refreshed each time you display the **Statistics** panel. To manually refresh the table, click **Refresh** .

Browse

The **Browse** panel lists active sessions. Click **Get Sessions** to obtain or refresh the list session IDs of all active sessions. The session IDs are sorted from most recently created sessions at the top of the list to oldest session at the bottom.

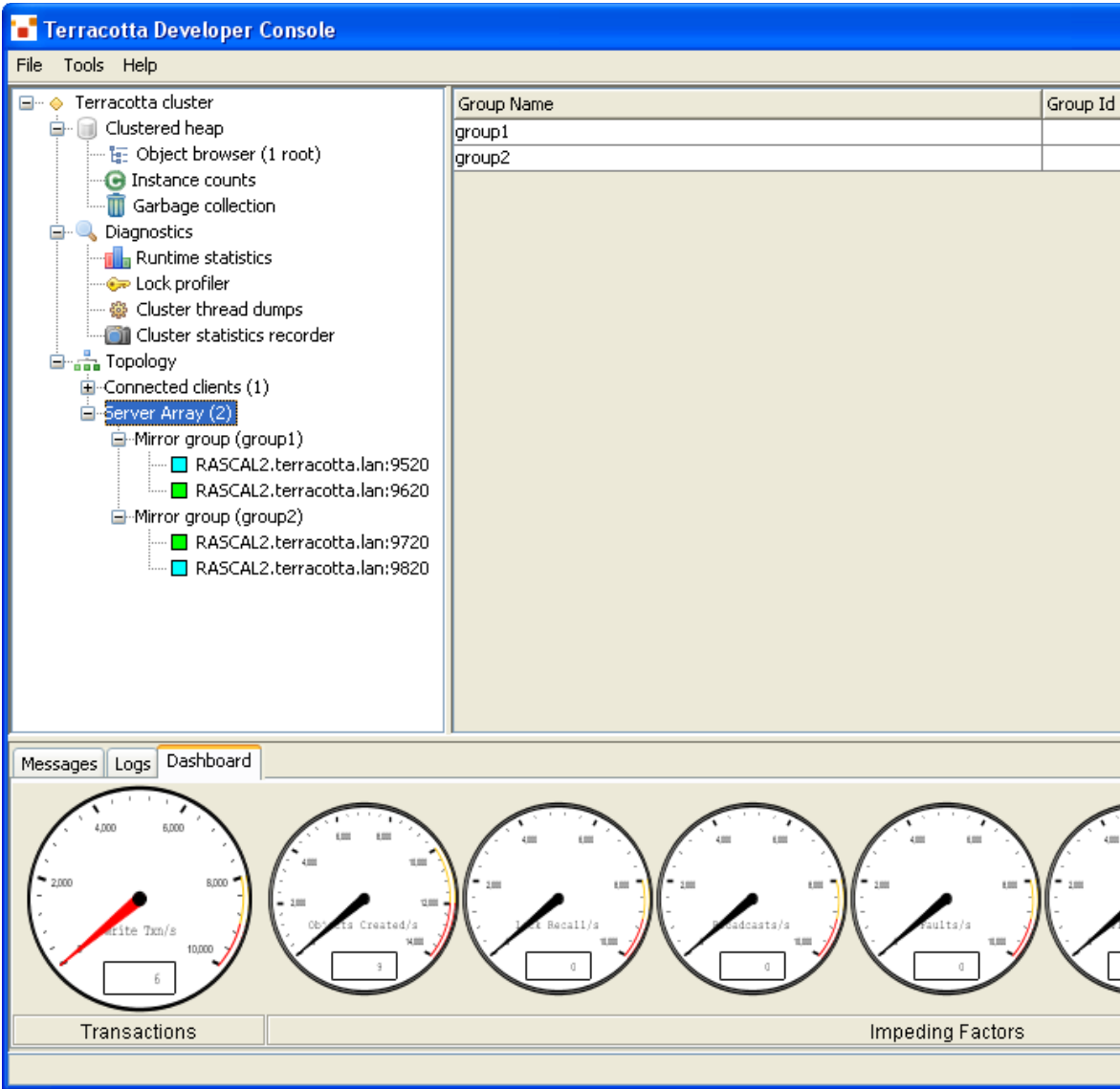
To view the attributes and attribute values for a session, select the session ID. The session's existing attributes and values are displayed in the panel below

To manually expire a session, select the session's ID and open its context menu (for example, right-click on the session ID), then choose **Expire** from the context menu.

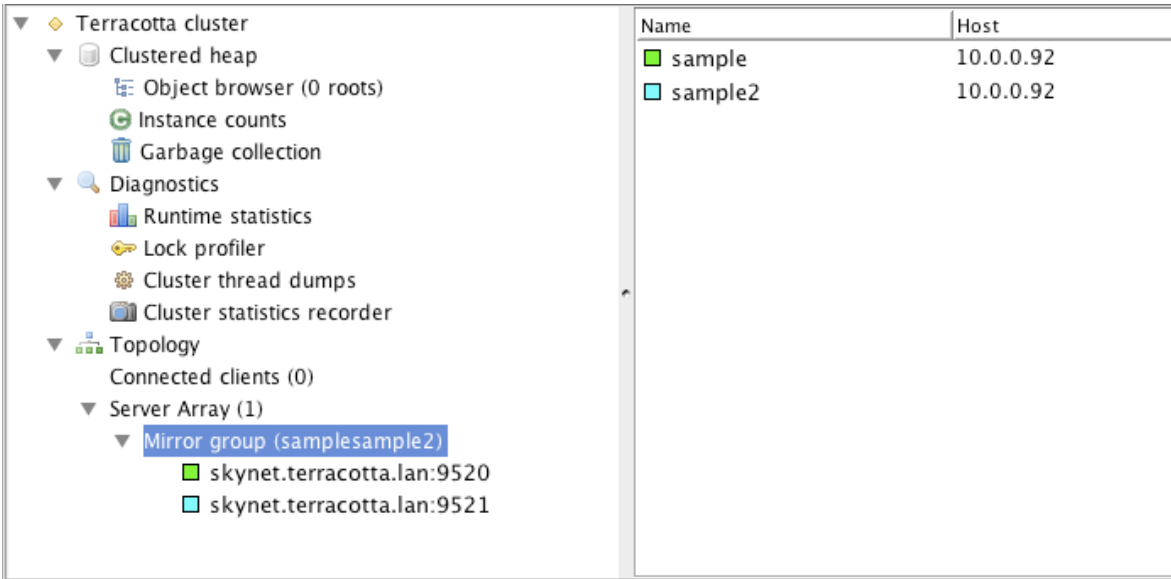
8.1.7 Working with Terracotta Server Arrays

Terracotta servers are arranged in *mirror groups* , each of which contains at least one active server instance. A High Availability mirror group also contains one backup server instance, sometimes called a passive server or "hot standby." Under the **Topology** node, a **Server Array** node lists all of the mirror groups in the cluster.

To view a table of mirror groups with their group IDs, expand the **Topology** node, then click **Server Array** .



To view a table of the servers in a mirror group, expand the **Server Array** node, then click the mirror group whose servers you want to display. The table of servers includes each server's status and name, hostname or IP address, and JMX port.



To view the servers' nodes under a mirror-group node, expand the mirror-group node.

8.1.7a Server Panel

Selecting a specific server's node displays that server's panel, with the **Main**, **Environment**, **Config**, and **Logging Settings** tabs.

The **Main** tab displays the server status and a list of properties, including the server's IP address, version, license, and persistence and failover modes.

Standing by on Tue Mar 31 10:49:55 PDT 2011

⚠ This Terracotta server array is configured for permanent-store mode. In the event that all Terracotta server nodes fail, all clustered data will be lost and no existing cluster. To ensure that in the event of a full cluster, clients may rejoin the cluster, change the configuration to permanent-store and restart:

```
<server>
  <dso>
    <persistence>
      <mode>permanent-store</mode>
    </persistence>
  </dso>
</server>
```

Field	Value
Host	skynet.terracotta.lan
Address	10.10.10.10
JMX port	9520
DSO port	9521
Version	Terracotta 3.5.0
Build	20090310
License	root
Persistence mode	temporary
Failover mode	network

The **Environment** tab displays the server's JVM system properties and provides a Find tool.

Environment tab showing JVM system properties:

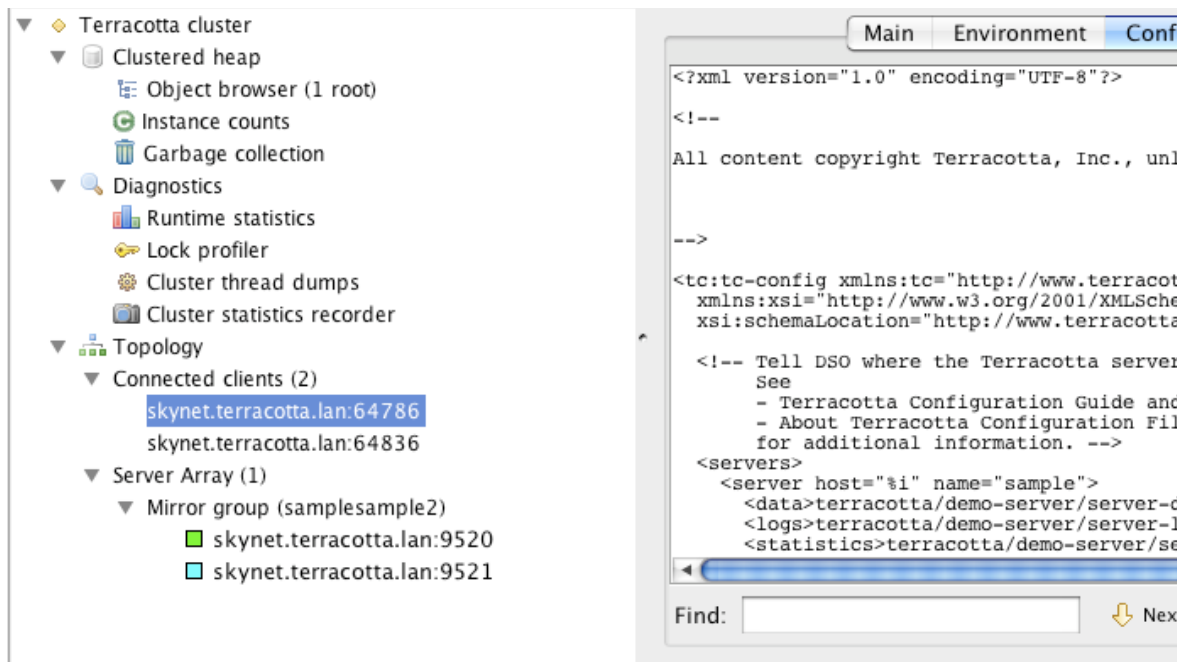
```

awt.toolkit: sun.awt.windows.WToolkit
com.sun.management.jmxremote:
file.encoding: Cp1252
file.encoding.pkg: sun.io
file.separator: \
h2.maxFileRetry: 8
java.awt.graphicsenv: sun.awt.Win32GraphicsEnvironment
java.awt.printerjob: sun.awt.windows.WPrinterJob
java.class.path: C:\Program Files\Terracotta\Terracotta-3.0.0-stable1\lib\
java.class.version: 50.0
java.endorsed.dirs: C:\Program Files\Java\jdk1.6.0_06\jre\lib\endorsed
java.ext.dirs: C:\Program Files\Java\jdk1.6.0_06\jre\lib\ext;C:\WINDOWS\system32\java
java.home: C:\Program Files\Java\jdk1.6.0_06\jre

```

Find: Next Previous

The **Config** tab displays the Terracotta configuration the server is using and provides a Find tool.



The **Logging Settings** tab displays logging options, each of which reports on data captured in five-second intervals:

- **FaultDebug** - Logs the types of objects faulted from disk and the number of faults by type.
- **RequestDebug** - Logs the types of objects requested by clients and the number of requests by type.
- **FlushDebug** - Logs the types of objects flushed from clients and a count of flushes by type.
- **BroadcastDebug** - Logs the types of objects that changed and caused broadcasts to other clients, and a count of those broadcasts by type.
- **CommitDebug** - Logs the types of objects committed to disk and a count of commits by type.

8.1.7b Connecting and Disconnecting from a Server

The Terracotta Developer Console connects to a cluster through one of the cluster's Terracotta servers. Being connected to a server means that the console is listening for JMX events coming from that server.

NOTE: Inability to Connect to Server

If you have confirmed that a Terracotta server is running, but the Terracotta Developer Console is unable to connect to it, a firewall on your network could be blocking the server's JMX port.

The console is disconnected from a cluster's servers when it's disconnected from the cluster. The console is also disconnected from a server when that server is shut down, even though the server may still appear in the console as part of the cluster. A server's connection status is indicated by its status light (see [Server Status \(server-stat\)](#)).

Note that disconnecting from a server does not shut the server down or alter its status in the cluster. Servers can be shut down using the stop-tc-server script (see [Start and Stop Server Scripts \(start-tc-server, stop-tc-server\)](#)).

TIP: Server Shutdown Button

A server shutdown button is available in the [Terracotta Operations Center](#).

8.1.7c Server Connection Status

A Terracotta server's connection status is indicated by a status light next to the server's name. The light's color indicates the server's current connection status. A cluster can have one server, or be configured with multiple servers that communicate state over the network or use a shared file-system.

The following table summarizes the connection status lights.

Status Light	Server Status	Notes
GREEN	Active	The server is connected and ready for work.

RED	Unreachable	The server, or the network connection to the server, is down.
YELLOW	Starting or Standby	A server is starting up; in a disk-based multi-server cluster, a passive server goes into standby mode until a file lock held by the active server is released. Normally the file lock is released only when the active server fails. The passive will then move to ACTIVE state (green status light).
ORANGE	Initializing	In a network-based multi-server cluster, a passive server must initialize its state before going into standby mode.
CYAN	Standby	In a network-based multi-server cluster, a passive server is ready to become active if the active server fails.

8.1.8 Working with Clients

Terracotta clients that are part of the cluster appear under the **Connected clients** node. To view the **Connected clients** node, expand the **Topology** node. The **Connected clients** panel displays a table of connected clients. The table has the following columns:

- Host - The client machine's hostname.
- Port - The client's DSO Port.
- ClientID - The client's unique ID number.
- Live Objects - The number of shared objects currently in the client's heap.

Terracotta cluster			
<ul style="list-style-type: none"> Clustered heap <ul style="list-style-type: none"> Object browser (1 root) Instance counts Garbage collection Diagnostics <ul style="list-style-type: none"> Runtime statistics Lock profiler Cluster thread dumps Cluster statistics recorder Topology <ul style="list-style-type: none"> Connected clients (2) <ul style="list-style-type: none"> skynet.terracotta.lan:64786 skynet.terracotta.lan:64836 Server Array (1) <ul style="list-style-type: none"> Mirror group (samplesample2) <ul style="list-style-type: none"> skynet.terracotta.lan:9520 skynet.terracotta.lan:9521 			
Host	Port	Client ID	Live Objects
skynet.terracotta.lan	64786		
skynet.terracotta.lan	64836		

To view the client nodes that appear in the **Connected clients** panel, expand the **Connected clients** node.

8.1.8a Client Panel

Selecting a specific client's node displays that client's panel, with the **Main** , **Environment** , **Config** , and **Logging** tabs.

The **Main** tab displays a list of client properties such as hostname and DSO port.

Field	Value
Host	skynet.terracotta.lan
Port	64786
Client ID	0
Version	Ter
Build	200

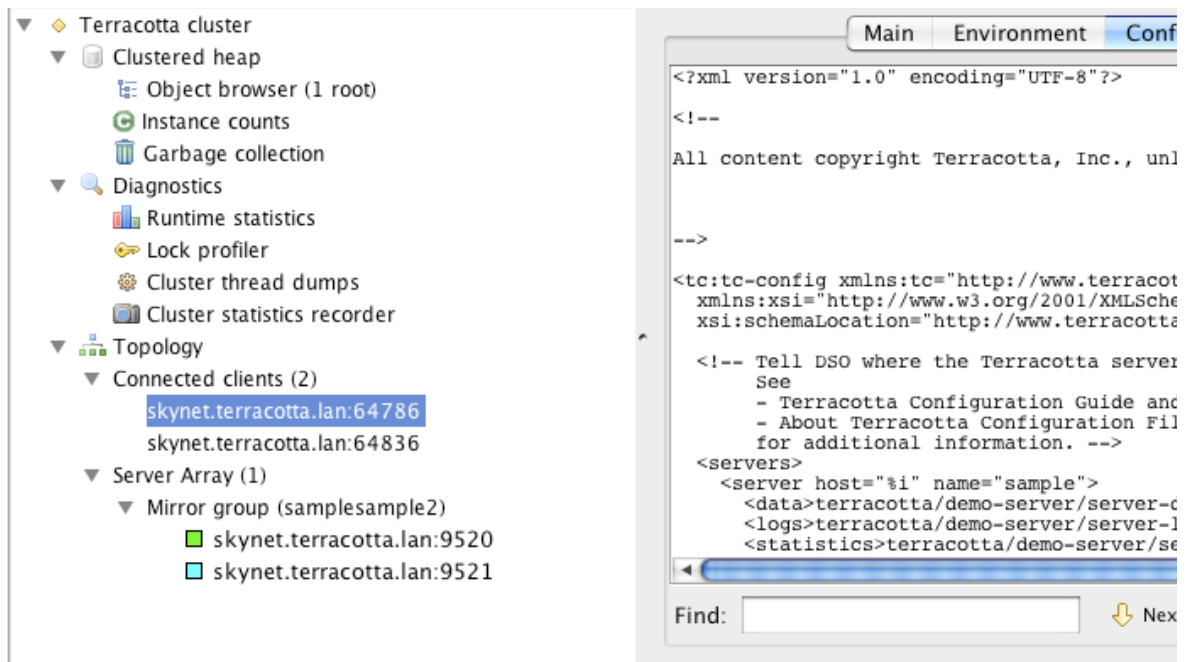
The **Environment** tab displays the client's JVM system properties and provides a **Find** tool.

```

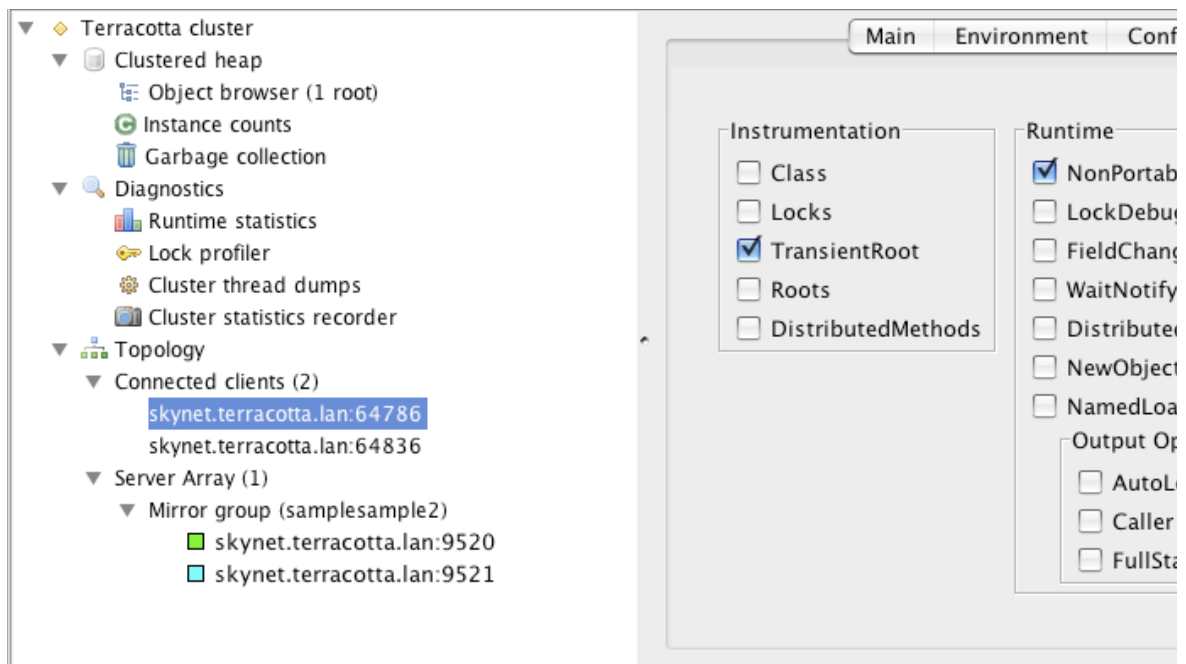
apple.awt.graphics.UseQuartz:
awt.nativeDoubleBuffering:
awt.toolkit:
file.encoding:
file.encoding.pkg:
file.separator:
ftp.nonProxyHosts:
gopherProxySet:
h2.maxFileRetry:
http.nonProxyHosts:
java.awt.Window.locationByPlatform:
java.awt.graphicsenv:
java.awt.printerjob:
java.class.path:
java.class.version:
java.endorsed.dirs:
java.ext.dirs:
java.home:
java.io.tmpdir:
java.library.path:
java.runtime.name:
java.runtime.version:
java.specification.name:
java.specification.vendor:
  
```

Find: Next

The **Config** tab displays the Terracotta configuration the client is using and provides a **Find** tool.



The **Logging** tab displays options to add logging items corresponding to DSO client debugging. See the [Configuration Guide and Reference](#) for details on the various debug logging options.



8.1.8b Connecting and Disconnecting Clients

When started up properly, a Terracotta client is automatically added to the appropriate cluster.

When a Terracotta client is shut down or disconnects from a server, that client is automatically removed from the cluster and no longer appears in the Terracotta Developer Console .

TIP: Client Disconnection

A client disconnection button is available in the [Terracotta Operations Center](#) .

8.1.9 Monitoring Clusters, Servers, and Clients

The Terracotta Developer Console provides visual monitoring functions using dials, icons, graphs, statistics, counters, and both simple and nested lists. You can use these features to monitor the immediate and overall health of your cluster as well as the health of individual cluster components.

Client Flush and Fault Rate Graphs

Client flush and fault rates are a measure of shared data flow between Terracotta servers and clients. These graphs can reflect trends in the flow of shared objects in a Terracotta cluster. Upward trends in flow can indicate insufficient heap memory, poor locality of reference, or newly changed environmental conditions. For more information, see [Client Flush Rate \(Cluster, Server, Client\)](#) and [Client Fault Rate \(Cluster, Server, Client\)](#).

Cache Miss Rate Graph

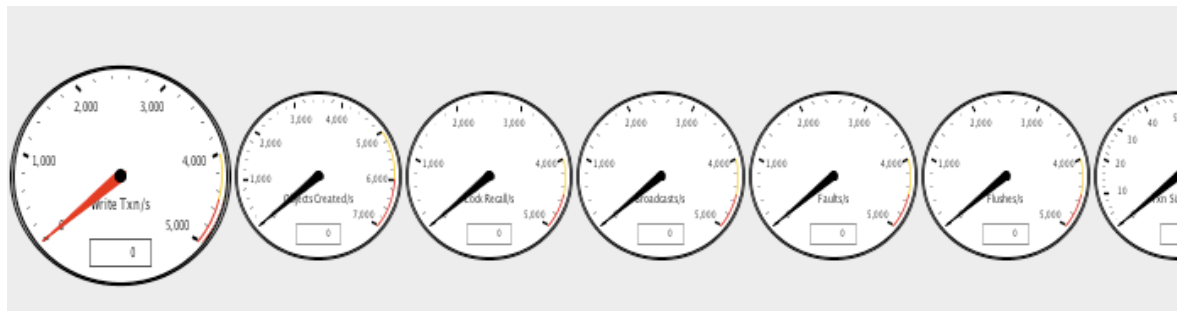
The Cache Miss Rate measures the number of client requests for an object that cannot be met by a server's cache and must be faulted in from disk. An upward trend in this graph can expose a bottleneck in your cluster. For more information, see [onheap fault/flush Rate \(Cluster, Server\)](#).

8.1.9a Real-Time Performance Monitoring

Real-time cluster monitoring allows you to spot issues as they develop in the cluster.

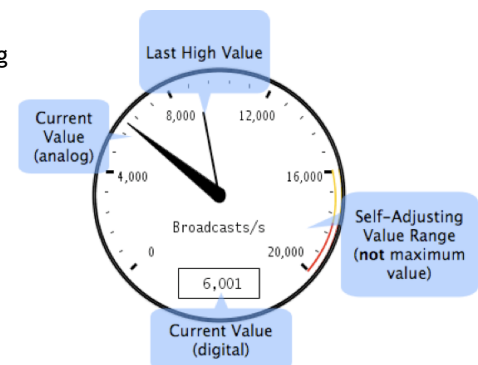
Dashboard

The cluster activity gauges provide real-time readings of critical cluster metrics.



Each gauge has the following characteristics:

- Yellow and red zones on the dial indicate when the metric value has reached warning or extreme levels.
- A digital readout field displays the metric's current value.
- A tooltip shows the metric's full name, last maximum value, and average value (over all samples).
- By default, values are sampled over one-second intervals (except for **Unacked Txns**). The sample rate can be changed in the Options dialog box (see [Runtime Statistics](#)).
- A "high-water" mark tracks the last high value, fading after several seconds.
- A self-adjusting value range uses a built-in multiplier to automatically scale with the cluster.



The left-most gauge (the large dial with the red needle) measures the rate of write transactions, which reflects the work being done in the cluster, based on [Terracotta transactions](#). This gauge may have a high value or trend higher in a busy cluster. An indication that the cluster may be overloaded or out of tune is when this gauge is constantly at the top of its range.

The remaining gauges, which measure "impeding factors" in your cluster, typically fluctuate or remain steady at a low value. If any impeding factors consistently trend higher over time, or remain at a high value, a problem may exist in the cluster. These gauges are listed below:

- **Objects Created/s** -- The rate of shared objects being created. A rising trend can have a negative impact on performance by reducing available memory and necessitating more garbage collection.

- **Lock Recalls/s** -- The number of locks being recalled by Terracotta servers. Growing lock recalls result from a high contention for shared objects, and have a negative performance impact. Higher locality of reference can usually lower the rate of lock recalls.
- **Broadcasts/s** -- The number of object changes being communicated by the server to affected clients. High broadcast rates raise network traffic and can have a negative performance impact. Higher locality of reference can usually lower the need for broadcasts.
- **Faults/s** -- Rate of faulting objects from servers to all connected clients. A high or increasing value can indicate one or more clients running low on memory or poor locality of reference.
- **Flushes/s** -- Rate of flushing objects from all connected clients to servers. A high or increasing value can indicate one or more clients running low on memory.
- **Transaction Size KB/s** -- Average size of total transactions.
- **Unacked Txns** -- The current count of unacknowledged client transactions. A high or increasing value can indicate one more troubled clients.

Runtime Statistics

Runtime statistics provide a continuous feed of sampled real-time data on a number of server and client metrics. The data is plotted on a graph with configurable polling and historical periods. Sampling begins automatically when a runtime statistic panel is first viewed, but historical data is not saved.

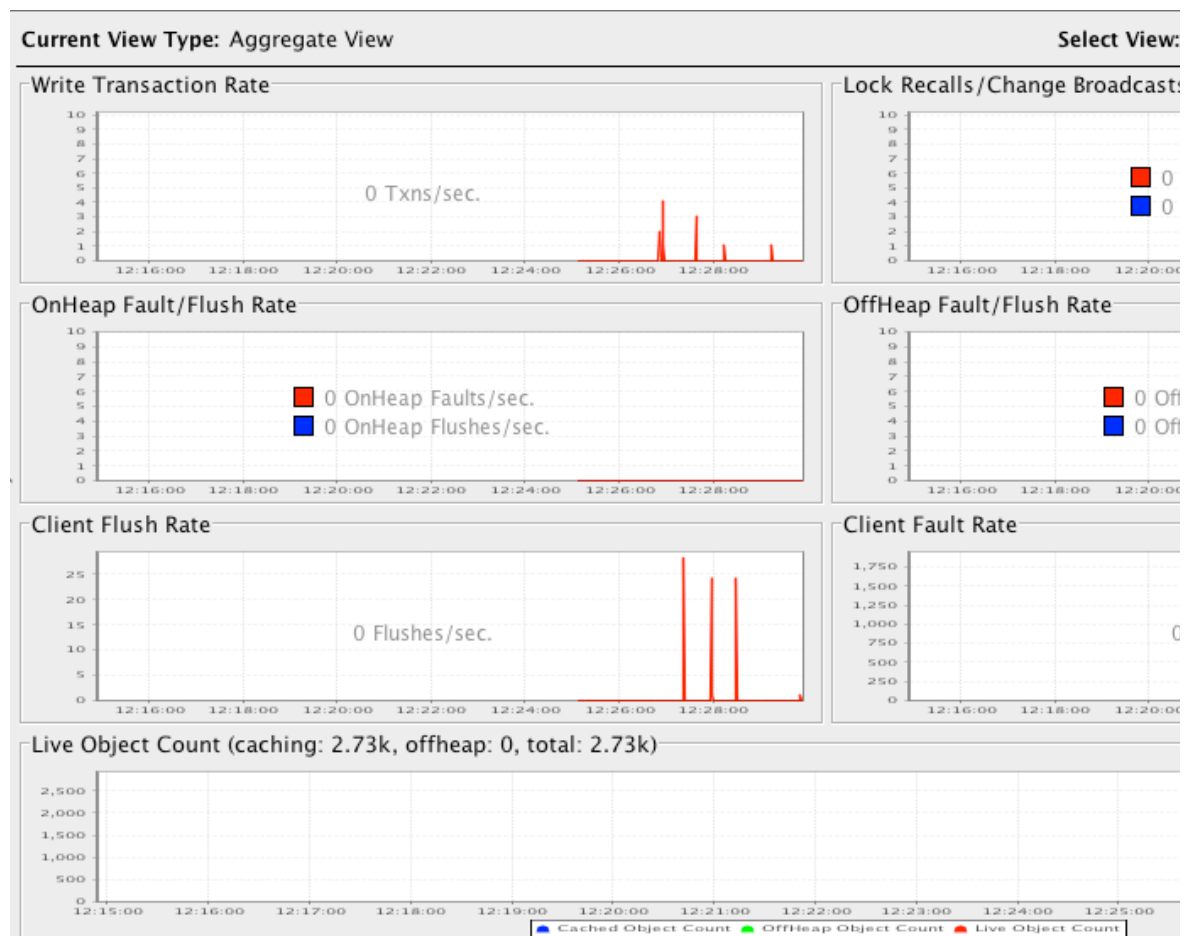
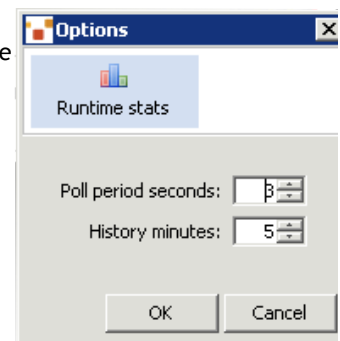
To adjust the poll and history periods, choose **Options** from the **Tools** menu. In the **Options** dialog, adjust the values in the polling and history fields. These values apply to all runtime-statistics views.

To record and save historical data, see [Cluster Statistics Recorder](#).

To view runtime statistics for a cluster, expand the cluster's **Monitoring** node, then click the **Runtime statistics** node.

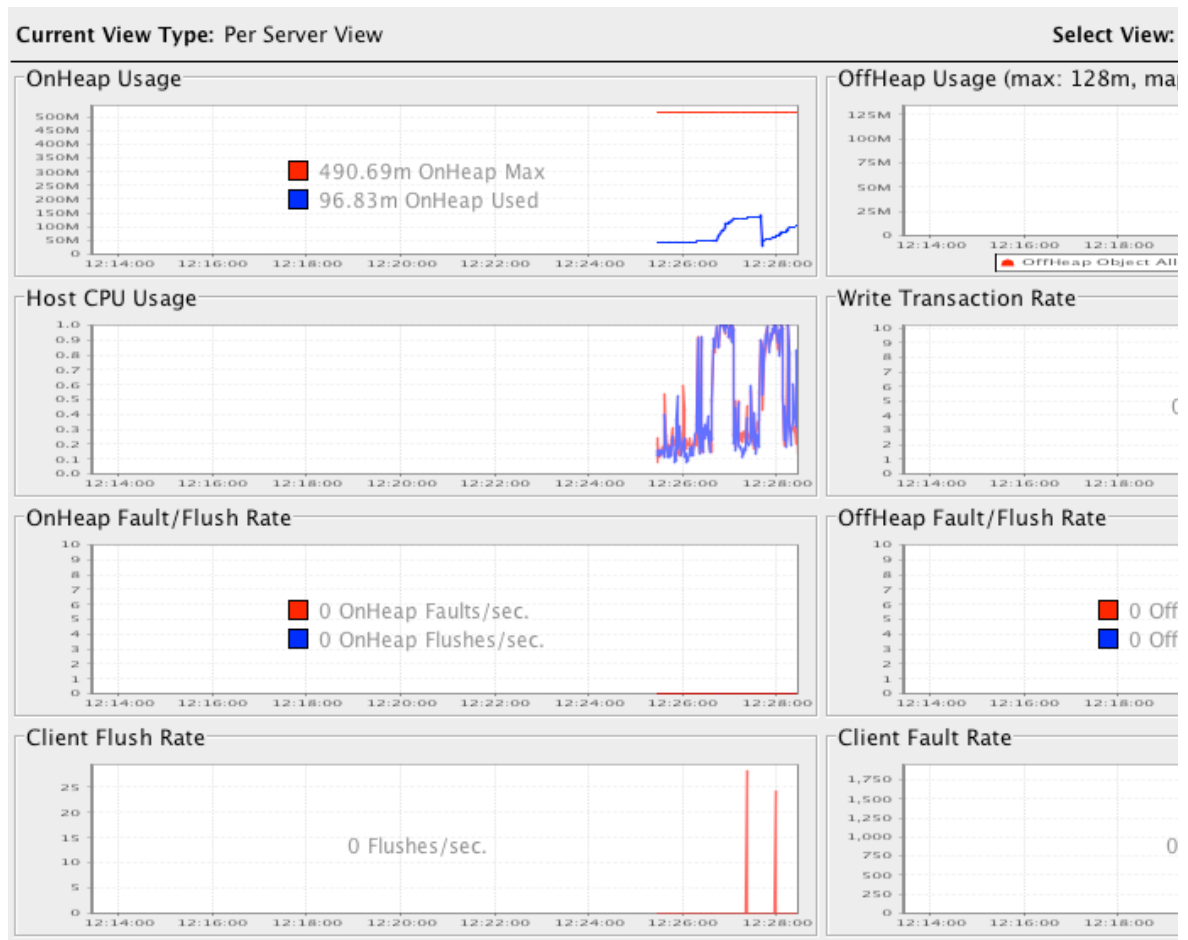
Use the **Select View** menu to set the runtime statistics view to one of the following:

- **Aggregate View** - Choose **Aggregate Server Stats** to display cluster-wide statistics.



- **Per-Client View** - Choose a client to display runtime statistics for that client.

- Per-Server View - Choose a server to display runtime statistics for that server.



Specific runtime statistics are defined in the following sections. The cluster components for which the statistic is available are indicated in parentheses.

WARNING: Fatal Errors Due to Statistics Gathering

Fatal errors can occur when collecting resource-specific statistics, such as those related to CPU and disk usage, due to incompatibilities between the Hyperic SIGAR statistics-collection framework and certain platforms. See errors related to "Hyperic" in the [Technical FAQ](#) for information on how to prevent these errors.

heap or onHeap Usage (Server, Client)

Shows the amount, in megabytes, of maximum available heap and heap being used.

NOTE: Aggregate View

For all statistics, if "Cluster" is indicated as a cluster component, it indicates the aggregate for all servers in the clusters.

offheap usage (server)

This statistic appears only if BigMemory is being used (see [Improving Server Performance With BigMemory](#)).

Shows the amount, in megabytes or gigabytes, of maximum available off-heap memory and off-heap memory being used.

Host CPU Usage (Server, Client)

Shows the CPU load as a percentage. If more than one CPU is being used, each CPU's load is shown as a separate graph line.

Write Transaction Rate (Cluster, Server, Client)

Shows the number of completed Terracotta transactions. Terracotta transactions are sets of one or more clustered object changes, or writes, that must be applied atomically.

TIP: Terracotta Transactions

Some statistics available through the Terracotta Developer Console are about [Terracotta transactions](#). Terracotta transactions are not application transactions. One Terracotta transaction is a batch of one or more writes to shared data.

onheap fault/flush Rate (Cluster, Server)

Faults from disk occur when an object is not available in a server's in-memory (on-heap) cache. Flushes occur when the on-heap cache must clear data due to memory constraints. The OnHeap Fault/Flush Rate statistic is a measure of how many objects (per second) are being faulted and flushed from and to the disk in response to client requests. Objects being requested for the first time, or objects that have been flushed from the server heap before a request arrives, must be faulted in from disk. High rates could indicate inadequate memory allocation at the server.

If BigMemory is being used (see [Improving Server Performance With BigMemory](#)), faults and flushes are to off-heap memory.

offheap fault/flush Rate (Cluster, Server)

This statistic appears only if BigMemory is being used (see [Improving Server Performance With BigMemory](#)).

Faults from disk occur when an object is not available in a server's in-memory off-heap cache. Flushes occur when the off-heap cache must clear data due to memory constraints. The OffHeap Fault/Flush Rate statistic is a measure of how many objects (per second) are being faulted and flushed from and to the disk in response to client requests. Objects being requested for the first time, or objects that have been flushed from off-heap memory before a request arrives, must be faulted in from disk. High rates could indicate inadequate memory allocation at the server.

Unacknowledged Transaction Broadcasts (Client)

Every [Terracotta transactions](#) in a Terracotta cluster must be acknowledged by Terracotta clients with in-memory shared objects that are affected by that transaction. For each client, Terracotta server instances keep a count of transactions that have not been acknowledged by that client. The Unacknowledged Transaction Broadcasts statistic is a count of how many transactions the client has yet to acknowledge. An upward trend in this statistic indicates that a client is not keeping up with transaction acknowledgments, which can slow the entire cluster. Such a client may need to be disconnected.

Client Flush Rate (Cluster, Server, Client)

The Client Flush Rate statistic is a measure of how many objects are being flushed out of client memory to the Terracotta server. These objects are available in the Terracotta server if needed at a later point in time. A high flush rate could indicate inadequate memory allocation at the client.

On a server, the Client Flush Rate is a total including all clients. On a client, the Client Flush Rate is a total of the objects that client is flushing.

Client Fault Rate (Cluster, Server, Client)

The Client Fault Rate statistic is a measure of how many objects are being faulted into client memory from the server. A high fault rate could indicate poor locality of reference or inadequate memory allocation at the client.

On a server, the Client Fault Rate is a total including all clients. On a client, the Client Fault Rate is a total of the objects that have been faulted to that client.

NOTE: Fault Count

When the Terracotta server faults an object, it also faults metadata for constructing a certain number of the objects referenced by or related to that object. This improves locality of reference. See the definition of the [fault-count property in the Terracotta Configuration Guide and Reference](#) for more information.

Lock Recalls / Change Broadcasts (Cluster)

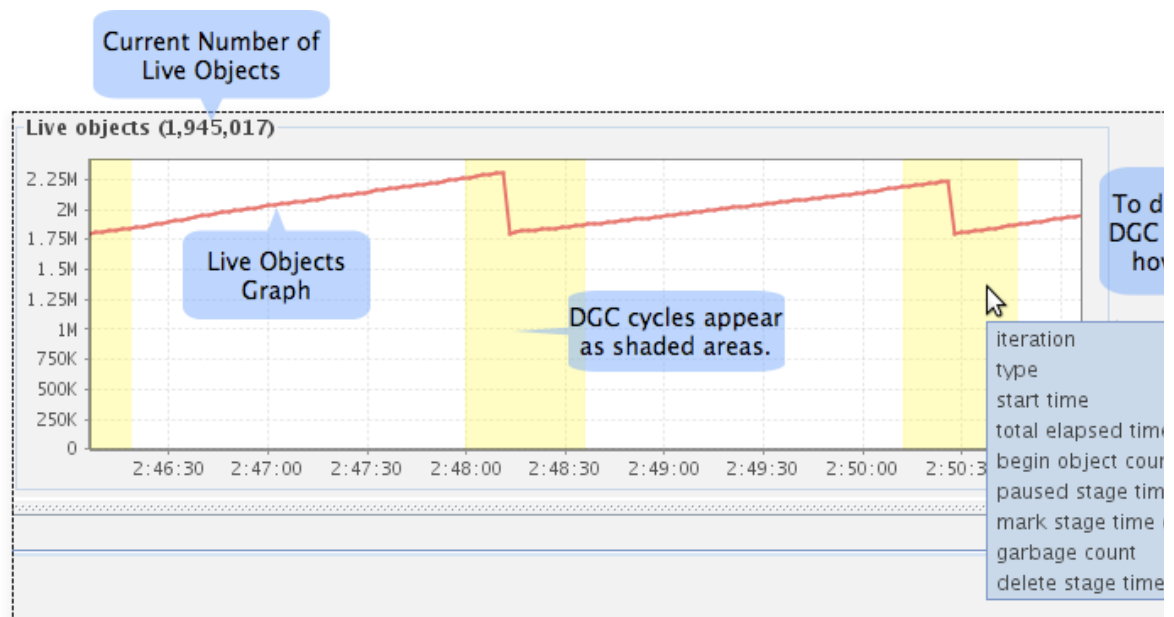
Terracotta servers recall a lock from one client as a response to lock requests from other clients. An upward trend in lock recalls could indicate poor locality of reference.

Change broadcasts tracks the number of object-change notifications that Terracotta servers are sending. See [l2 changes per broadcast](#), [l2 broadcast count](#), and [l2 broadcast per transaction](#) for more information on broadcasts.

Live Objects (Cluster)

Shows the total number of live objects on disk (red), in the off-heap cache (green), and in the on-heap cache (blue).

If the trend for the total number of live objects goes up continuously, clients in the cluster will eventually run out of memory and applications may fail. Upward trends indicate a problem with application logic, garbage collection, or a tuning issue on one or more clients. The total number of live objects is given in the graph's title.



Distributed Garbage Collection

Objects in a DSO root object graph can become unreferenced and no longer exist in the Terracotta client's heap. These objects are eventually marked as garbage in a Terracotta server instance's heap and from persistent storage by the Terracotta Distributed Garbage Collector (DGC). The DGC is unrelated to the Java garbage collector.

TIP: Distributed Garbage Collector

For more information on the DGC, see the [Terracotta Concept and Architecture Guide](#).

To view a history table of DGC activity in the current cluster, expand the cluster's **Cluster heap** node, then click the **Garbage collection** node. The history table is automatically refreshed each time a collection occurs. Each row in the history table represents one distributed garbage collection cycle, with the following columns:

Column	Definition	Values
Iteration	The index number of the DGC cycle	Sequential integer
Type	The type of cycle	<p>Full - Running a full collection cycle targeting all eligible objects.</p> <p>Young - Running a collection cycle targeting Young Generation objects.</p>
Status	The collection cycle's current state	<p>START - Monitoring for object reference changes and collecting statistics such as the object begin count.</p> <p>MARK - Determining which objects should be collected and which should not.</p> <p>PAUSE - Determining if any marked objects should not be collected.</p>

		(finalizing marked object list). DELETE - Deleting objects. COMPLETE - Completed cycle.
Start time	The date and time the cycle began	Date and time stamp (local server time)
Begin count	The total number of shared objects held by the server	Integer counter
Paused stage	The total time the DGC paused	Milliseconds
Mark stage	The total time the DGC took to mark objects for collection	Milliseconds
Garbage count	The number of shared objects marked for collection	Integer counter
Delete stage	The total time the DGC took to collect marked objects	Milliseconds
Total elapsed time	The total time the DGC took to pause, mark objects, and collect marked objects	Milliseconds

The DGC graph combines a real-time line graph (with history) displaying the DGC total elapsed time with a bar graph showing the total number of freed objects.

Triggering a DGC Cycle

The DGC panel displays a message stating the configured frequency of DGC cycles. To manually trigger a DGC cycle, click **Run DGC**.

8.1.9b Logs and Status Messages

Click the **Logs** tab in the Status Panel to display log messages for any of the servers in the cluster. From the **View log for** menu, choose the server whose logs you want to view.

The status bar at the bottom of the console window displays messages on the latest changes in the cluster, such as nodes joining or leaving.

8.1.9c Operator Events

The **Operator Events** panel, available with enterprise editions of Terracotta, displays cluster events received by the Terracotta server array. You can use the **Operator Events** panel to quickly view these events in one location in an easy-to-read format, without having to search the Terracotta logs.

To view the **Operator Events** panel, expand the Monitoring node in the cluster list, then click the **Operator Events** node.

Events are listed in a table with the following columns:

- Event Type - The level of the event (INFO, WARN, DEBUG, ERROR, CRITICAL) along with a color-coded light corresponding to the severity of the event.
- Time of Event - The event's date and time stamp.
- Node - The server receiving the event. Concatenated events are indicated when more than one server is listed in the Node column.
- Event System - The Terracotta subsystem that generated the event. The choices are MEMORY_MANAGER (virtual memory manager), DGC (distributed garbage collector), LOCK_MANAGER (cluster-wide locks manager), DCV2 (server-side caching), CLUSTER_TOPOLOGY (server status), and HA (server array).
- Message - Message text reporting on a discrete event.

An event appears in bold text until it is manually selected (highlighted). The text of an event that has been selected is displayed in regular weight.

TIP: Viewing Concatenated Events

The Operator Events panel concatenates events received by more than one server so that they appear in one row. Concatenated events are indicated when more than one server is listed in the Node column. To view these concatenated events, float your mouse button over the event to open a tool-tip list.

The **Operator Events** panel has the following controls:

- Mark All Viewed - Click this button to change the text of all listed events from bold to regular weight.
- Export - Click this button to export a text file containing all listed events.
- Select View - Use this drop-down menu to filter the list of displayed events. You can filter the list of events based on type (level) or by the generating system (or subsystem). For example, if you choose INFO from the menu, only events with this event type are displayed in the event list.

8.1.10 Advanced Monitoring and Diagnostics

Tools providing deep views into the clustered data and low-level workings of the Terracotta cluster are available under the **Platform** node. These are recommended for developers who are experienced with Java locks, concurrency, reading thread dumps, and understanding statistics.

8.1.10a Shared Objects

Applications clustered with Terracotta use shared objects to keep data coherent. Monitoring shared objects serves as an important early-warning and troubleshooting method that allows you to:

- Confirm that appropriate object sharing is occurring;
- be alerted to potential memory issues;
- learn when it becomes necessary to tune garbage collection;
- locate the source of object over-proliferation.

The Terracotta Developer Console provides the following tools for monitoring shared objects:

- [Object Browser](#)
- [Classes Browser](#)
- [Runtime Logging of New Shared Objects](#)

These tools are discussed in the following sections.

Object Browser

The Object Browser is a panel displaying shared object graphs in the cluster. To view the Object Browser, expand the **Clustered heap** node, then click the **Object browser** node.

The Object Browser does not refresh automatically. You can refresh it manually in any of the following ways

- Expand or collapse any part of any object graph.
- Press the F5 key on your keyboard.
- Right-click any item on the object graph that has an object ID (for example, @1001), then select **Refresh** from the context menu.

The following are important aspects of the object graph display:

- The top-level objects in an object graph correspond to the shared roots declared in the Terracotta server's configuration file.
- Objects referencing other objects can be expanded or collapsed to show or hide the objects they reference.
- Objects in the graph that are a collections type, and reference other objects, indicate the number of referenced objects they display when expanded. This number is given as a ratio in the format [X/Y], where X is the number of child elements being displayed and Y is the total number of child elements in the collection. Collections also have **More** and **Less** items in their context menus for manual control over the number of child elements displayed. By default, up to ten children (fields) are displayed when you expand a collections-type object in the object graph.
- A delay can occur when the object browser attempts to display very large graphs.
- An entry in the graph that duplicates an existing entry has an "up" arrow next to it. Click the up arrow to go to the existing entry.
- An entry in the graph called "Collected" with no data indicates an object that was made known to the console but no longer exists on the graph. The collected object will eventually disappear from the graph on refresh.
- Each element in the object appears with unique identifying information, as appropriate for its type. Each object appears with its fully qualified name.

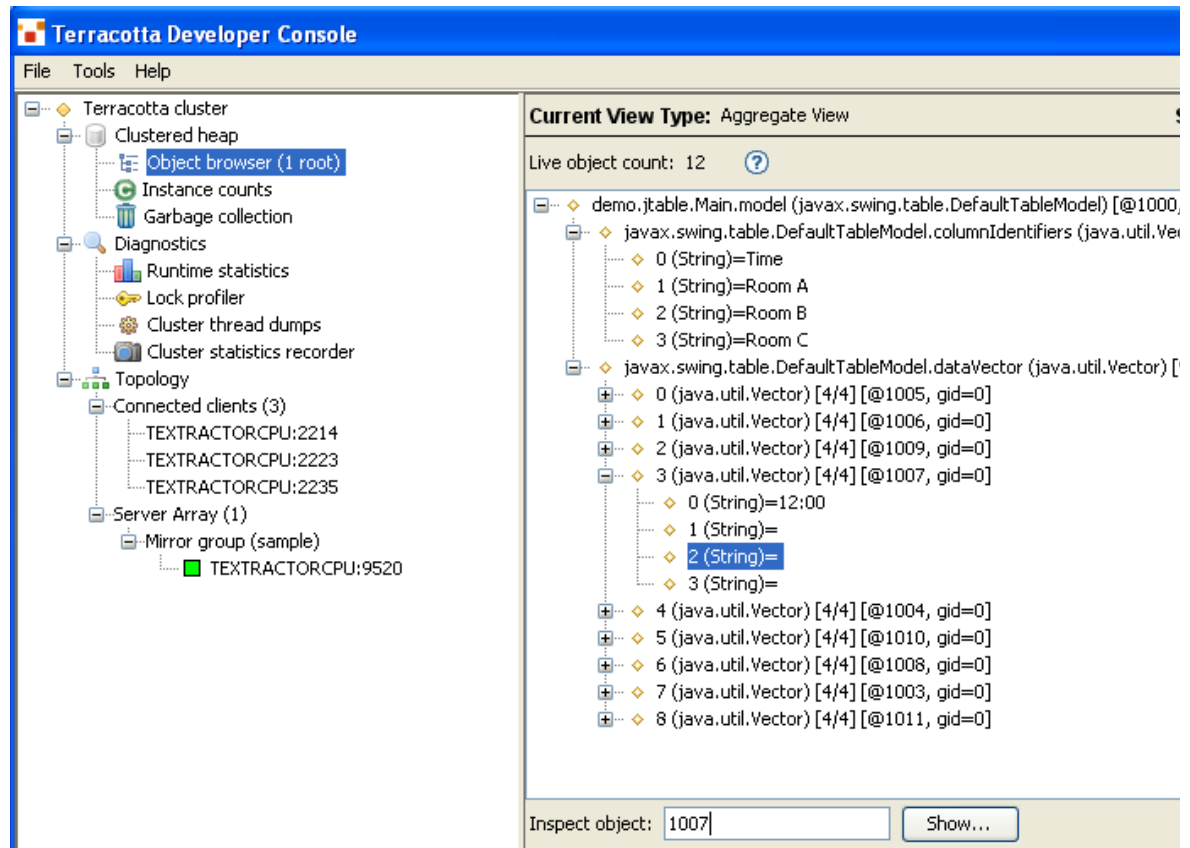
To inspect a portion of an object graph, follow these steps:

1. Find the object ID of the object at the root of the portion you want to inspect.
An object ID has the format @<integer>. For example, @1001 can be an object ID.
2. Enter that object ID in **Inspect object**.
3. Click **Show...**.

A window opens containing the desired portion of the graph.

Cluster Object Browsing

To browse the shared-object graphs for the entire cluster, select **Cluster Heap** from the **Select View** menu. All of the shared objects in the cluster-wide heap are graphed, but the browser doesn't indicate which client are sharing them.

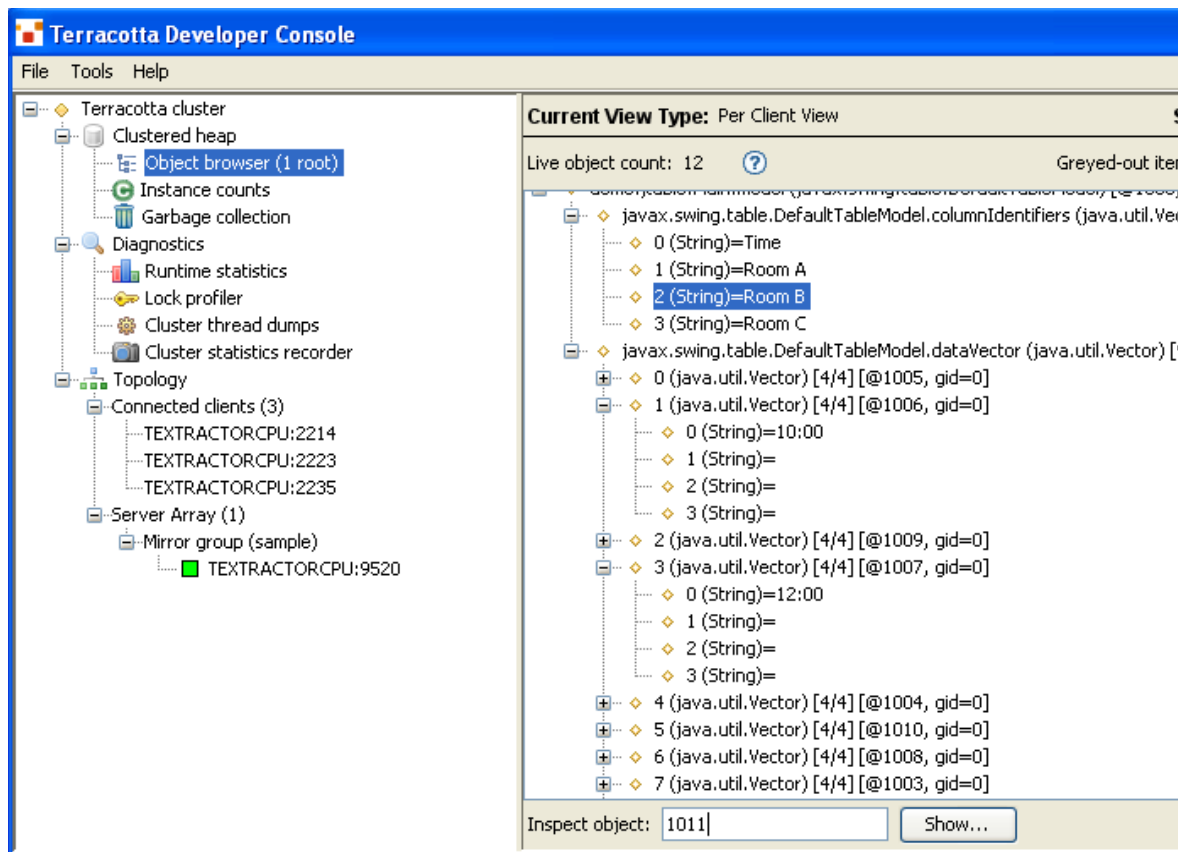


To see object graphs specific to a client, see [Client Object Browsing](#).

The browser panel displays a running total of the live objects in the cluster. This is the number of objects currently found in the cluster-wide heap; however, this total does not correspond to the number of objects you see in the object graph because certain objects, including literals such as strings, are not counted. These uncounted objects appear in the object graph without an object ID.

Client Object Browsing

To browse the shared-object graphs in a specific client, select the client from the **Select View** menu. All of the shared object graphs known to that client are graphed, but the ones not being shared by the client are grayed out.



The browser panel displays a running total of the live objects in the client. This is the number of objects currently found in the client heap; however, this total does not correspond to the number of objects you see in the object graph because the following types of objects are not counted:

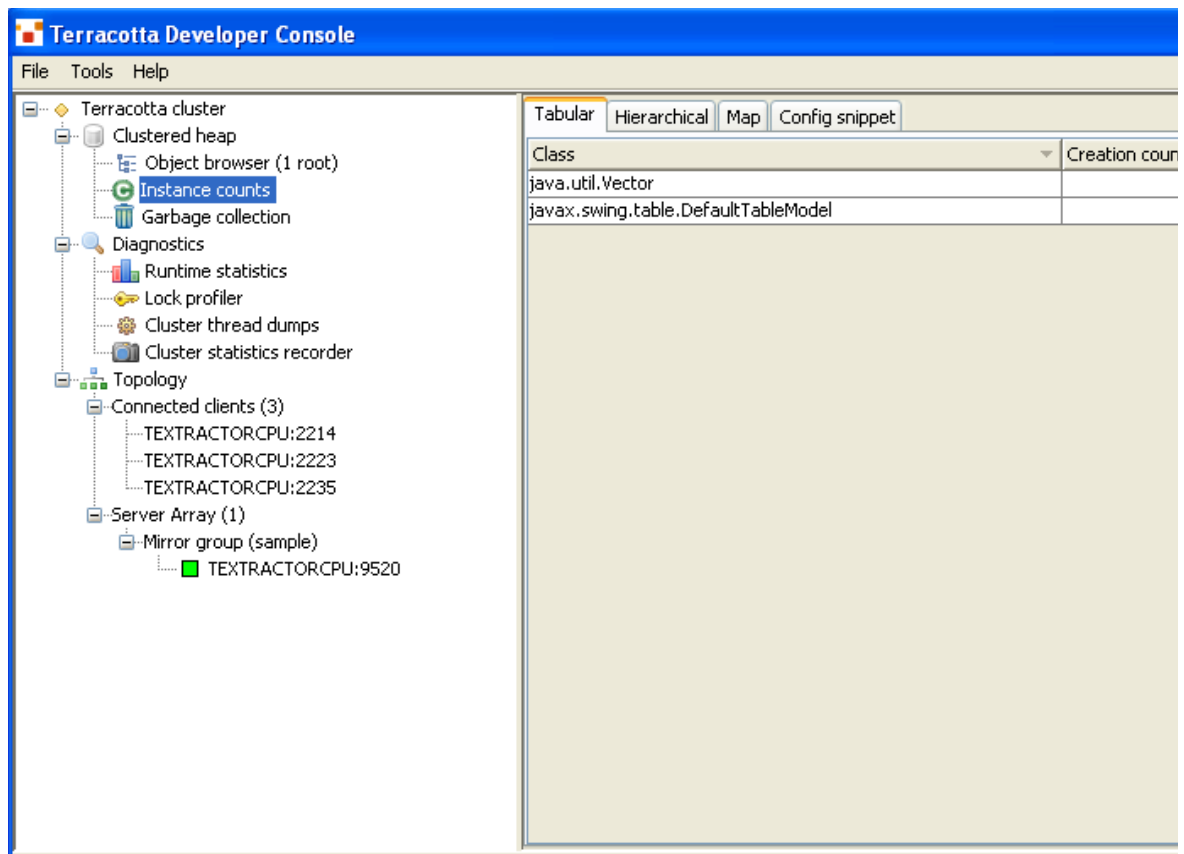
- Objects *not* being shared by the client
 - These unshared objects are grayed out in the object-browser view.
 - Literals such as strings
- These objects appear in the object graph without an object ID.

Classes Browser

Terracotta allows for transparent, clustered object state synchronization. To accomplish this feature, some of your application classes are adapted into new classes that are cluster-aware. Snapshots of the set of all such adapted classes known to the server are displayed in the **Instance counts** panel. The panel has the following tabs:

- **Tabular** - Lists all the adapted classes in a spreadsheet view, including the class name and a count of the number of instances of the class that have been created since the server started. Click the column title to sort along that column's contents.
- **Hierarchical** - Presents an expandable/collapsible Java package view of the adapted classes, along with a count of the number of instances of the class that have been created since the server started.
- **Map** - Displays an area map distinguishing the most (and least) heavily used adapted classes.
- **Config snippet** - A snippet from the <application> section of the Terracotta configuration file showing the how the instrumented classes are configured.

To refresh the values in the classes browser, select **Refresh** from the **Instance counts** context menu.



Runtime Logging of New Shared Objects

You can log the creation of all new shared objects by following these steps:

1. Select the target client in the cluster list.
2. Click the **Logging Settings** tab.
3. Enable **NewObjectDebug** from the Runtime list.

During development or debugging operations, logging new objects can reveal patterns that introduce inefficiencies or errors into your clustered application. However, during production it is recommended that this type of intensive logging be disabled.

See the [Configuration Guide and Reference](#) for details on the various debug logging options.

8.1.10b Lock Profiler

The Terracotta runtime system can gather statistics about the distributed locks set by the Terracotta configuration. These statistics provide insight into the workings of a distributed application and aid the discovery of highly-contented access to shared state. Statistics are displayed in a table (see [The Statistics Table](#)).

Enable lock profiling: Trace depth:

Lock	Times Requested	Times Hopped	Average Contenders	Average Acquire Time	Average Held Time	Average Nested
▼ @DistributedMethodCall	4	0	0	0	1	0
▼ demo.chatter.Cl	4	0	0	0	1	0
▼ demo.chatter	4	0	0	0	1	0
demo.chatter	4	0	0	0	1	0

Find:

Trace

```
demo.chatter.ChatManager.sendNewM
demo.chatter.ChatManager.send(Unkn
```

@DistributedMethodCall

Distributed Invoke

Using the Lock Profiler

To enable or disable lock-statistics gathering, follow these steps:

1. Expand the **Diagnostics** node, then click the **Lock profiler** node.
2. Click **On** to enable statistics gathering.
3. Click **Off** to disable statistics gathering.
4. Specify the **Trace depth** (lock code-path trace depth) to set the number of client call-stack frames to analyze.
See [Trace Depth](#) for more information.
5. Click **Clients** to view lock statistics for Terracotta clients, or click **Servers** to view lock statistics for Terracotta servers.
Client-view lock statistics are based on the code paths in the clustered application that result in a lock being taken out. Server-view lock statistics concern the cluster-wide nature of the distributed locks.
See [Lock Element Details](#) for more information.
6. Click **Refresh** to display the latest statistics.

NOTE: Gathering Statistics Impacts Performance

Gathering and recording statistics can impact a cluster's performance. If statistics are being gathered, you are alerted in the cluster list by a flashing icon next to the affected cluster.

Lock Names

Each lock has a corresponding identifier, the lock-id. For a named lock the lock-id is the lock name. For an autolock the lock-id is the server-generated id of the object on which that lock was taken out. An example of an autolock id is @1001. That autolock id corresponds to the shared object upon which distributed synchronization was carried out. You can use the object browser (see [Object Browser](#)) to view the state of shared object @1001.

Searching for Specific Locks

You can search for specific locks listed in the **Lock** column. Enter a string in **Find** , then click **Next** or **Previous**

to move through matching entries.

Trace Depth

A single lock-expression in the configuration can result in the creation of multiple locks by the use of wildcard patterns. A single lock can be arrived at through any number of different code paths. For example, there could be 3 different call sequences that result in a particular lock being granted, with one of the paths rarely entered and another responsible for the majority of those lock grants. By setting the trace depth appropriately you can gain insight into the behavior of your application and how it can affect the performance of your clustered system.

The trace depth control sets the number of client call-stack frames that are analyzed per lock event to record lock statistics. A depth of 0 gathers lock statistics without regard to how the lock event was arrived at. A lock depth of 1 means that one call-stack frame will be used to disambiguate different code paths when the lock event occurred. A lock depth of 2 will use two frames, and so on.

TIP: Generating Line Numbers in Lock Traces

Trace stack frames can include Java source line numbers if the code is compiled with debugging enabled. This can be done by passing the `-g` flag to the `javac` command, or in Ant by defining the `javac` task with the `debug="true"` attribute.

With a trace-depth setting of 1 all locks are recorded together, regardless of the call path. This is because the stack depth analyzed will always be just the method that resulted in the lock event (in other words the surrounding method). For example, a lock event that occurs within method `Foo()` records all lock events occurring within `Foo()` as one single statistic.

With a lock depth of 2, different call paths can be separated because both the surrounding method and the calling method are used to record different lock statistics. For example, the callers of `Foo()`, `Bar1()` and `Bar2()`, are also considered. A call path of `Bar1() -> Foo()` is recorded separately from `Bar2() -> Foo()`.

The Statistics Table

Lock Statistic	Description
Times Requested	Number of times this lock was requested by clients in the cluster.
Times Hopped	Times an acquired greedy lock was retracted from a holding client and granted to another client.
Average Contenders	Average number of threads wishing to acquire the lock at the time it was requested.
Average Acquire Time	Average time (in milliseconds) between lock request and grant.
Average Held Time	Average time (in milliseconds) grantee held this lock.
Average Nested	Average number of outstanding locks held by acquiring thread at grant time.

TIP: Greedy Locks

Terracotta employs the concept of *greedy locks* to improve performance by limiting unnecessary lock hops. Once a client has been awarded a lock, it is allowed to keep that lock until another client requests it. The assumption is that once a client obtains a lock it is likely to request that same lock again. For example, in a cluster with a single node repeatedly manipulating a single shared object, server lock requests should be 1 until another client enters the cluster and begins manipulating that same object. Server statistics showing "na" (undefined) are likely due to greedy locks.

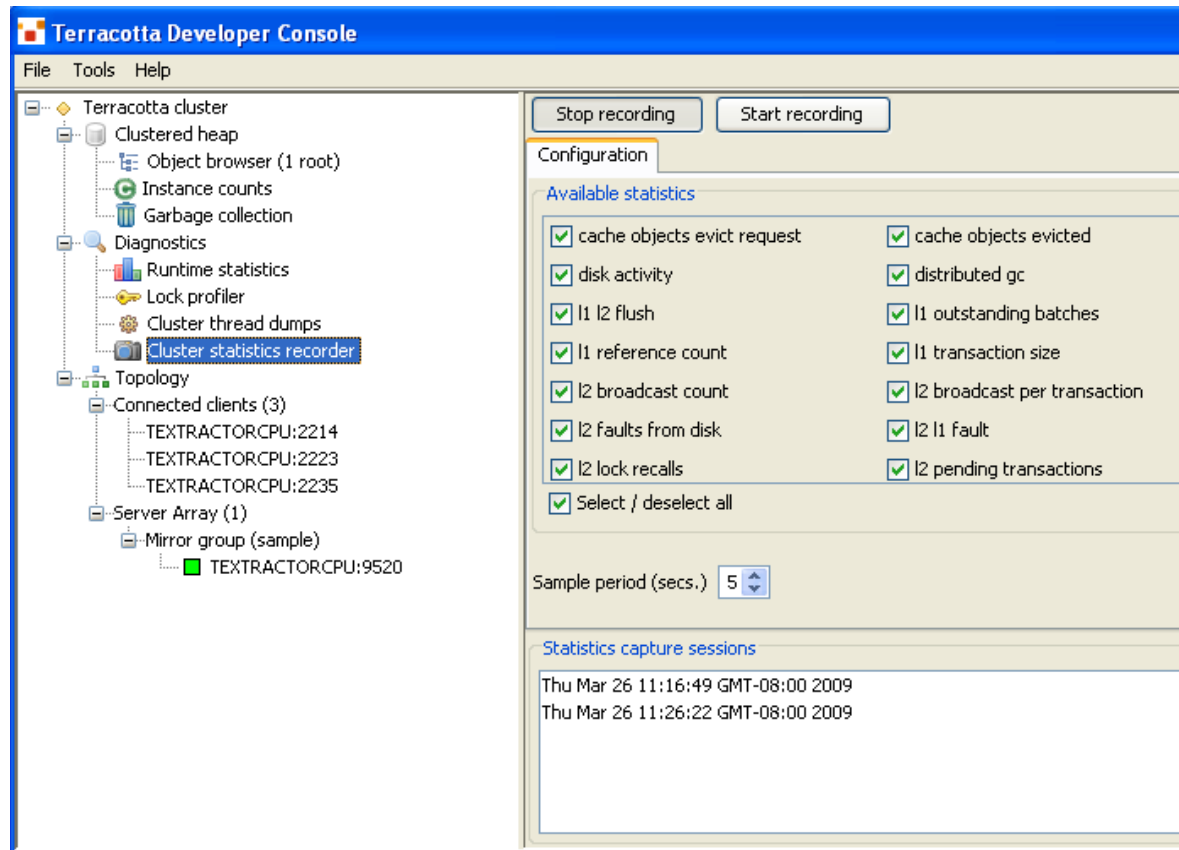
Lock Element Details

The bottom portion of the client's view displays details on the selected lock element. The currently selected lock trace is shown on the left and the configuration element responsible for the creation of the selected lock is shown on the right.

8.1.11 Recording and Viewing Statistics

8.1.11a Cluster Statistics Recorder

The **Cluster Statistics Recorder** panel can generate recordings of selected cluster-wide statistics. This panel has controls to start, stop, view, and export recording sessions. You can use the [Snapshot Visualization Tool](#) to view the recorded information.



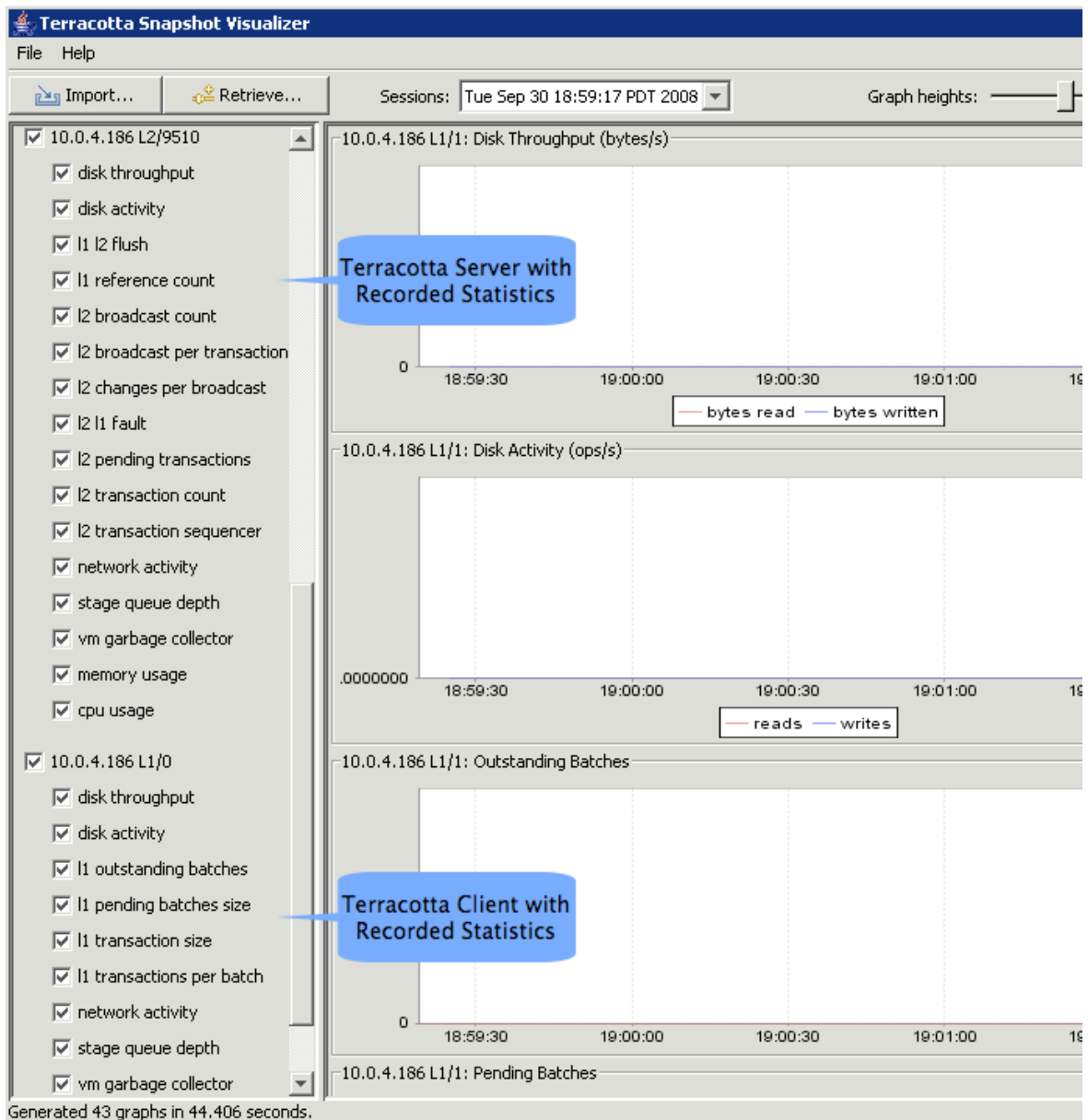
For definitions of available statistics, see [Definitions of Cluster Statistics](#). To learn about configuring the Terracotta Cluster Statistics Recorder, using its command-line interface, and more, see the [Platform Statistics Recorder Guide](#).

WARNING: Potentially Severe Performance Impact

Gathering and recording statistics can significantly impact a cluster's performance. If statistics are being gathered, you are alerted in the cluster list by a flashing icon next to the affected cluster. In a production environment or if testing performance, the impact of recording statistics should be well understood.

8.1.11b Snapshot Visualization Tool

The **Snapshot Visualization Tool (SVT)** provides a graphical view of cluster information and statistics. The view is created using data recorded with the Statistics Recorder.



The SVT is provided as a TIM, called `tim-svt`, which you can install using [tim-get](#). Once the SVT is installed, start (or restart) the Terracotta Developer Console and confirm that the **View** button on the Cluster Statistics Recorder is enabled. No changes to the Terracotta configuration file are necessary when you install the SVT.

TIP: Terracotta in Eclipse

In Eclipse with the Terracotta Eclipse plug-in, use the **Terracotta | Update modules...** menu to install SVT.

SVT controls are defined below.

Import...

Load a saved cluster statistics recording that was saved to file. Clicking **Import...** opens a standard file-selection window for locating the file.

Retrieve...

Find recorded sessions on active Terracotta servers. Clicking **Retrieve...** opens a dialog to enter the server address in the format `<server_ip_address:JMX_port>` or `<server_name:JMX_port>` as defined in `tc-config.xml`. Once connected to the server, available recorded sessions appear in the SVT **Sessions** menu.

Sessions

Select a recorded session from the **Sessions** drop-down menu to load it. **Sessions** lists the recorded sessions last retrieved from a Terracotta server.

Graph heights

Scale the y-axis on the graphs displayed in the SVT. Moving the slider to the left shrinks the graph height, while moving it to the right grows the graph height.

Server, Client, and Statistics checkboxes

Choose which servers and clients have their recorded statistics displayed in the graphs. For each server and client, choose which statistics are displayed in the graphs.

Generate graphs

Click **Generate graphs** to create the graphs from the selected statistics.

8.1.12 Troubleshooting the Console

This section provides solutions for common issues that can affect both the Terracotta Developer Console and Operations Center.

8.1.12a Cannot Connect to Cluster (Console Times Out)

If you've verified that your Terracotta cluster is up and running, but your attempt to monitor it remotely using a Terracotta console is unsuccessful, a firewall may be the cause. Firewalls that block traffic from Terracotta servers' JMX ports prevent monitoring tools from seeing those servers. To avoid this and other connection issues that may also be attributable to firewalls, ensure that the JMX and DSO ports configured in Terracotta are unblocked on your network.

If it is certain that no firewall is blocking the connection, network latencies may be causing the console to time out before it can connect to the cluster. In this case, you may need to adjust the console timeout setting using the following property:

```
-Dcom.tc.admin.connect-timeout=100000
```

where the timeout value is given in milliseconds.

8.1.12b Failure to Display Certain Metrics Hyperic (Sigar) Exception

The Terracotta Developer Console (or Terracotta Operations Center) may fail to display (or graph) certain metrics, while at the same time a certain "Hyperic" (or "Sigar") exception is reported in the logs or in the console itself.

These two problems are related to starting Java from a location different than the value of JAVA_HOME. To avoid the Hyperic error and restore metrics to the Terracotta consoles, invoke Java from the location specified by JAVA_HOME.

NOTE: Segfaults and Hyperic (Sigar) Libraries

If Terracotta clients or servers are failing with exceptions related to Hyperic (Sigar) resource monitoring, see [this Technical FAQ item](#).

8.1.12c Console Runs Very Slowly

If you are using the Terracotta Developer Console to monitor a remote cluster, especially in an X11 environment, issues with Java GUI rendering may arise that slow the display. You may be able to improve performance simply by changing the rendering setup.

If you are using Java 1.7, set the property `sun.java2d.xrender` to "true" to enable the latest rendering technology:

```
-Dsun.java2d.xrender=true
```

For Java 1.5 and 1.6, be sure to set property `sun.java2d.pmoscreeen` to "false" to allow Swing buffers to reside in memory:

```
-Dsun.java2d.pmoscreeen=false
```

For information about this Java system property, see <http://download.oracle.com/javase/1.5.0/docs/guide/2d/flags.html#pmoscreen>.

You can add these properties to the Developer Console start-up script (`dev-console.sh` or `dev-console.bat`).

8.1.12d Console Logs and Configuration File

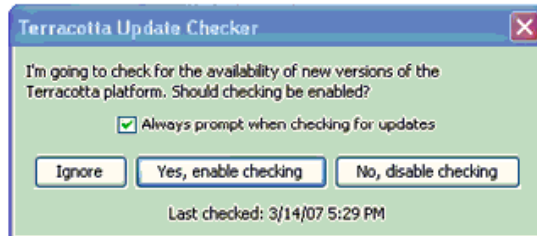
The Terracotta Developer Console stores a log file (`devconsole.log.<number>`) and a configuration file (`.AdminClient.xml`) in the home directory of the user who ran the console. The log file contains a record of errors encountered by the console.

8.1.13 Backing Up Shared Data

The [Terracotta Operations Center](#) provides console-based backup configuration. Enterprise versions of Terracotta also include a backup script ([Database Backup Utility \(backup-data\)](#)).

8.1.14 Update Checker

On a bi-weekly basis the Terracotta Developer Console will check, when first started, on updates to the Terracotta platform. By default, a notice informing you that an update check is about to be performed is displayed, allowing to ignore the immediate check, acknowledge and allow the check, or to disable further checking.



Should the update check be allowed, the Terracotta Developer Console will query the OpenTerracotta website (www.terracotta.org) and report on any new updates. Should the update checker feature be disabled, it can always be re-enabled via the **Help > Update Checker...** menu.

8.1.15 Definitions of Cluster Statistics

The following categories of cluster information and statistics are available for viewing and recording in the [Cluster Statistics Recorder](#).

TIP: Terracotta Cluster Nomenclature

l1 = Terracotta client
l2 = Terracotta server instance

8.1.15a cache objects evict request

The total number of objects marked for eviction from the l1, or from the l2 to disk. Evicted objects are still referenced, and can be faulted back to the l1 from the l2 or from disk to l2. The SVT graphs this metric for each l1 and l2 separately. High counts imply that free memory could be low.

8.1.15b cache objects evicted

The number of objects actually evicted. If this metric is not close in value to cache objects evict request_, then memory may not be getting freed quickly enough.

8.1.15c l1 l2 flush

The object flush rate when the l1 flushes objects to the l2 to free up memory or as a result of GC activity.

8.1.15d l2 faults from disk

The number of times the l2 has to load objects from disk to serve l1 object demand. A high faulting rate could indicate an overburdened l2.

8.1.15e l2 l1 fault

The number of times an l2 has to send objects to an l1 because the objects do not exist in the l1 local heap due to memory constraints. Better scalability is achieved when this number is lowered through improved locality and usage of an optimal number of JVMs.

8.1.15f memory (usage)

The amount of memory (heap) usage over time.

8.1.15g vm garbage collector

The standard Java garbage collector's behavior, tracked on all JVMs in the cluster.

8.1.15h distributed gc (distributed garbage collection, or DGC)

The behavior of the Terracotta tool that collects distributed garbage on an l2. The DGC can pause all other work on the l2 to ensure that no referenced objects are flagged for garbage collection.

8.1.15i l2 pending transactions

The number of [Terracotta transactions](#) held in memory by a Terracotta server instance for the purpose of minimizing disk writes. Before writing the pending transactions, the Terracotta server instance optimizes them by folding in redundant changes, thus reducing its disk access time. Any object that is part of a pending transaction cannot be changed until the transaction is complete.

8.1.15j stage queue depth

The depth to which visibility into Terracotta server-instance work-queues is available. A larger depth value allows more detail to emerge on pending task-completion processes, bottlenecks due to application requests or behavior, types of work being done, and load conditions. Rising counts (of items in these processing queues) indicate backlogs and could indicate performance degradation.

8.1.15k server transaction sequencer stats

Statistics on the Terracotta server-instance transaction sequencer, which sequences transactions as resource become available while maintaining transaction order.

8.1.15l network activity

The amount of data transmitted and received by a Terracotta server instance in bytes per second.

8.1.15m l2 changes per broadcast

The number of updates to objects on disk per broadcast message (see [l2 broadcast count](#)).

8.1.15n message monitor

The network message count flowing over TCP from Terracotta clients to the Terracotta server.

8.1.15o l2 broadcast count

The number of times that a Terracotta server instance has transmitted changes to objects. This "broadcast" occurs any time the changed object is resident in more than one Terracotta client JVM. This is not a true broadcast since messages are sent only to clients where the changed objects are resident.

8.1.15p l2 transaction count

The number of [Terracotta transactions](#) being processed (per second) by a Terracotta server instance.

8.1.15q l2 broadcast per transaction

The ratio of broadcasts to [Terracotta transactions](#). A high ratio (close to 1) means that each broadcast is reporting few transactions, and implies a high co-residency of objects and inefficient distribution of application data. A low ratio (close to 0) reflects high locality of reference and better options for linear scalability.

8.1.15r system properties

Snapshot of all Java properties passed in and set at startup for each JVM. Used to determine configuration states at the time of data capture, and for comparison of configuration across JVMs.

8.1.15s thread dump

Displays a marker on all statistics graphs in the SVT at the point when a thread dump was taken using the Cluster statistics recorder_. Clicking the marker displays the thread dump.

8.1.15t disk activity

The number of operations (reads and writes) per second, and the number of bytes per second (throughput). The number of reads corresponds to l2 faulting objects from disk, while writes corresponds to l2 flushing objects to disk. The SVT graphs the two aspects separately.

8.1.15u cpu (usage)

The percent of CPU resources being used. Dual-core processors are broken out into CPU0 and CPU1.

8.2 Terracotta Tools Catalog

A number of useful tools are available to help you get the most out of installing, testing, and maintaining

Terracotta. Many of these tools are included with the Terracotta kit, in the `bin` directory (unless otherwise noted). Some tools are found only in an enterprise version of Terracotta. To learn more about the many benefits of an enterprise version of Terracotta, see [Enterprise Products](#).

If a tool has a script associated with it, the name of the script appears in parentheses in the title for that tool section. The script file extension is `.sh` for UNIX/Linux and `.bat` for Microsoft Windows.

Detailed guides exist for some of the tools. Check the entry for a specific tool to see if more documentation is available.

8.2.1 Terracotta Maven Plugin

The Terracotta Maven Plugin allows you to use Maven to install, integrate, update, run, and test your application with Terracotta.

The Terracotta Maven Plugin, along with more documentation, is available from the [Terracotta Forge](#).

8.2.2 TIM Management (`tim-get`)

The `tim-get` script provides a simple way to update the JARs in the Terracotta kit as well as manage the catalog of available Terracotta integration modules (TIMs) and other JARs.

TIP: `tim-get` Documentation

See the [tim-get guide](#) for detailed usage information

8.2.3 Sessions Configurator (`sessions-configurator`)

The Terracotta Sessions Configurator is a graphical tool that assists you in clustering your web application's session data.

See the [Terracotta Sessions Configurator Guide](#) for detailed installation and feature information. See [Clustering a Spring Web Application](#) for a tutorial on clustering a Spring web application using the Terracotta Sessions Configurator.

8.2.4 Developer Console (`dev-console`)

The Terracotta Developer Console is a graphical tool for monitoring various aspects of your Terracotta cluster for testing and development purposes. The console can record a number of statistics that can be viewed over time using the Snapshot Visualization Tool (SVT).

See the [Terracotta Developer Console](#) for detailed information on the features of the console and SVT.

8.2.5 Operations Center (`ops-center`)

TIP: Enterprise Feature

Available in Terracotta enterprise editions.

The Terracotta Operations Center is a console for monitoring and managing various aspects of your Terracotta cluster in production.

See the [Terracotta Operations Center](#) for detailed information on the features and use of this graphical tool

8.2.6 Archive Utility (`archive-tool`)

`archive-tool` is used to gather filesystem artifacts generated by a Terracotta Server or DSO client application for the purpose of contacting Terracotta with a support query.

Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\archive-tool.bat <args>
```

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/archive-tool.sh <args>
```

where `<args>` are:

- `[-n]` (No Data - excludes data files)
- `[-c]` (Client - include files from the dso client)
- `<path to terracotta config xml file (tc-config.xml)> | <path to data and/or logs directory>`
- `[<output filename in .zip format>]`

8.2.7 Database Backup Utility (backup-data)

TIP: Enterprise Feature

Available in Terracotta enterprise editions.

The Terracotta backup utility creates a backup of the data being shared by your application. Backups are saved to the default directory `data-backup`. Terracotta automatically creates `data-backup` in the directory containing the Terracotta server's configuration file (`tc-config.xml` by default).

However, you can override this default behavior by specifying a different backup directory in the server's configuration file using the `< data-backup >` property:

```
<servers>
  <server host="%i" name="myServer">
    <data-backup>/Users/myBackups</data-backup>
    <statistics>terracotta/-server/server-statistics</statistics>
    <dso>
      <persistence>
        <mode>permanent-store</mode>
      </persistence>
    </dso>
  </server>
</servers>
```

8.2.7a Using the Terracotta Operations Center

NOTE:

In the example above, persistence mode is configured for permanent-store, which is required to enable backups.

Backups can be performed from the Terracotta [Operations Center \(ops-center\)](#) using the **Backup** feature.

Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\backup-data.bat <hostname> <jmx port>
```

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/backup-data.sh <hostname> <jmx port>
```

8.2.7b Example (UNIX/Linux)

```
${TERRACOTTA_HOME}/bin/backup-data.sh localhost 9520
```

To restore a backup, see the section *Restoring a Backup* in [Terracotta Operations Center](#).

8.2.8 Distributed Garbage Collector (run-dgc)

`run-dgc` is a utility that causes the specified Terracotta Server to perform distributed garbage collection (DGC). Use `run-dgc` to force a DGC cycle in addition to or instead of automated DGC cycles. Forced DGC cycle can also be initiated from the Terracotta Developer Console and the Terracotta Operations Center.

NOTE: Running Concurrent DGC Cycles

Two DGC cycles cannot run at the same time. Attempting to run a DGC cycle on a server while another DGC cycle is in progress generates an error.

Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\run-dgc.bat <hostname> <jmx-port>
```

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/run-dgc.sh <hostname> <jmx-port>
```

8.2.8a Further Reading

For more information on distributed garbage collection, see the [Concept and Architecture Guide](#) and the [Tuning Guide](#).

For information on monitoring the Terracotta Server's garbage collection, see [8.1 Terracotta Developer Console](#).

8.2.9 Start and Stop Server Scripts (start-tc-server, stop-tc-server)

Use the `start-tc-server` script to run the Terracotta Server, optionally specifying a configuration file:

Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\start-tc-server.bat [-f <config specification>]
```

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh [-f <config specification>]
```

<config specification> can be one of:

- path to configuration file
- URL to configuration file
- <server host>:<dso-port> of another running Terracotta Server

If no configuration is specified, a file named `tc-config.xml` in the current working directory will be used. If no configuration is specified and no file named `tc-config.xml` is found in the current working directory, a default configuration will be used. The default configuration includes no DSO application element and is therefor useful only in development mode, where each DSO client provides it's own configuration.

For production purposes, DSO clients should obtain their configuration from a Terracotta Server using the `tc.config` system property.

Use the `stop-tc-server` script to cause the Terracotta Server to gracefully terminate:

Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\stop-tc-server.bat [<server-host> <jmx-port>]
```

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/stop-tc-server.sh [<server-host> <jmx-port>]
```

`stop-tc-server` uses JMX to ask the server to terminate. If you have secured your server, requiring authenticated access, you will be prompted for a password.

8.2.9a Further Reading

For more information on securing your server for JMX access see the section `/tc:tc-config/servers/server/authentication` in [Configuration Guide and Reference](#).

8.2.10 Version Utility (version)

Terracotta Version Tool is a utility script that outputs information about the Terracotta installation, including the version, date, and version-control change number from which the installation was created. When contacting Terracotta with a support query, please include the output from Version Tool to expedite the resolution of your issue.

Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\version.bat
```

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/version.sh&
```

8.2.11 Server Status (server-stat)

The Server Status tool is a command-line utility for checking the current status of one or more Terracotta servers instances.

Server Status returns the following data on each server it queries:

- Health - OK (server responding normally) or FAILED (connection failed or server not responding correctly).
- Role - The server's position in an active-passive group. Single servers always show ACTIVE. "Hot standbys" are shown as PASSIVE.
- State - The work state that the server is in.
- JMX port - The TCP port the server is using to listen for JMX events.
- Error - If the Server Status tool fails, the type of error.
- Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\server-stat.bat <args>
```

where <args> are:

- [-s] host1,host2,... - Check one or more servers using the given hostnames or IP addresses using the default JMX port (9520).
- [-s] host1:9520,host2:9521,... - Check one or more servers using the given hostnames or IP addresses

with JMX port specified.

- [-f] <path>/tc-config.xml - Check the servers defined in the current Terracotta configuration file.
- [-h] - Display help on Server Status.

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/platform/bin/server-stat.sh <args>
```

<args> are the same as shown for Microsoft Windows.

8.2.11a Example

The following example shows usage of and output from the Server Status tool.

```
[PROMPT] server-stat.sh -s 0.0.0.0:9520
0.0.0.0.health: OK
0.0.0.0.role: ACTIVE
0.0.0.0.state: ACTIVE-COORDINATOR
0.0.0.0.jmxport: 9520
```

If no server is specified, by default the Server Status checks the status of localhost at JMX port 9520.

8.2.12 Cluster Statistics Recorder (tc-stats)

The Terracotta Cluster Statistics Recorder allows you to configure and manage the recording of statistics for your entire cluster. The Cluster Statistics Recorder has a command-line interface (CLI) useful for scripting statistics-gathering operations. For more information, see the section [Command-Line Interface in the Platform Statistics Recorder Guide](#).

8.2.13 DSO Tools

NOTE:

The following subject matter covers aspects of core Terracotta DSO technology. DSO is recommended for *advanced users only*.

8.2.13a Sample Launcher (samples)

Terracotta Sample Launcher is a graphical tool that provides an easy way to run the Terracotta for POJO samples in a point-and-click manner. When run, Sample Launcher automatically starts up the demo Terracotta Server, which it also shuts down upon termination. A selection of samples demonstrating POJO clustering are listed and can be launched. Descriptions of each sample, including information about how to run the sample from the command-line, as well as sample code and configuration can be browsed.

Run Sample Launcher from the command line.

Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\platform\tools\pojo\samples.bat
```

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/platform/tools/pojo/samples.sh&
```

8.2.13b Make Boot Jar Utility (make-boot-jar)

The `make-boot-jar` script generates a boot jar file based on the contents of the Terracotta configuration file, determined in the following order:

- The configuration file specified by `-f`. For example:

```
[PROMPT] ./make-boot-jar.sh -f ../tc-config.xml
```

- The configuration file found in the current directory.
- The default Terracotta configuration file.

If the boot jar exists, `make-boot-jar` re-creates the boot jar only if it needs to be re-created. It can be forced to create one by passing the `-w` option. It returns with exit code 1 if the boot jar file is incomplete, otherwise the exit code is 0.

8.2.13c Scan Boot Jar Utility (scan-boot-jar)

The `scan-boot-jar` script verifies the contents of the boot jar file against an L1 configuration. It will list all of the classes declared in the `<additional-boot-jar-classes/>` section that is not included in the boot jar, as well as classes in the boot jar that is not listed in the `<additional-boot-jar-classes/>` section. It returns with exit code 1 if the boot jar file is incomplete, otherwise the exit code is

0.

8.2.13d Boot Jar Path Utility (boot-jar-path)

`boot-jar-path` is a helper utility used by the `dso-env` script for determining the full path to the JVM-specific DSO bootjar. This script is not meant to be used directly.

8.2.13e DSO Environment Setter (dso-env)

The `dso-env` script helps you set up your environment to run a DSO client application, using existing environment variables and setting `TC_JAVA_OPTS` to a value you can pass to java. It expects `JAVA_HOME`, `TC_INSTALL_DIR`, and `TC_CONFIG_PATH` to be set prior to invocation. `dso-env` is meant to be executed by your custom startup scripts, and is also used by each Terracotta demo script.

Microsoft Windows

```
set TC_INSTALL_DIR=%TERRACOTTA_HOME%
set TC_CONFIG_PATH=<config specification>
call "%TC_INSTALL_DIR%\bin\dso-env.bat" -q
set JAVA_OPTS=%TC_JAVA_OPTS% %JAVA_OPTS%
call "%JAVA_HOME%\bin\java" %JAVA_OPTS% ...
```

UNIX/Linux

```
TC_INSTALL_DIR=${TERRACOTTA_HOME}
TC_CONFIG_PATH=<config specification>
. ${TC_INSTALL_DIR}/platform/bin/dso-env.sh -q
JAVA_OPTS="${TC_JAVA_OPTS} ${JAVA_OPTS}"
${JAVA_HOME}/bin/java ${JAVA_OPTS} ...
```

<config specification> above is either the path to a local config file or a <server>:
<dso-port> tuple specifying the configuration of a running Terracotta Server. If the config specification is not set, an existing file in the current working directory named `tc-config.xml` will be used. If no config is specified and no local `tc-config.xml` is found, the Terracotta runtime will fail to start.

8.2.13f Java Wrapper (dso-java)

`dso-java` is a script that can be used to run a DSO client application in a manner similar to running a standard java application. For instance, one way to run the `jtable` POJO sample is to first run the demo server:

```
[PROMPT] ${TERRACOTTA_HOME}/samples/start-demo-server.sh&
```

... and to change into the `jtable` directory and invoke `dso-java` in the following way:

```
[PROMPT] cd ${TERRACOTTA_HOME}/samples/pojo/jtable
[PROMPT] ${TERRACOTTA_HOME}/platform/bin/dso-java -cp classes demo.jtable.Main
```

`dso-java` uses the [DSO Environment Setter \(dso-env\)](#) helper script to specify the Java runtime options needed to activate the Terracotta runtime. The configuration file, `tc-config.xml`, is located in the current working directory. If the configuration file was in a different location, specify that location using the `tc.config` Java system property:

```
-Dtc.config=<config specification>
```

<config specification> is a comma-separated list of:

- path to configuration file
- URL to configuration file
- <server host>:<dso-port> of running Terracotta Server

When a <config specification> is comprised of a list of configuration sources, the first configuration successfully obtained is used.

9 Terracotta DSO Installation

Terracotta Distributed Shared Objects (DSO) clusters require a special installation. DSO uses object identity, instrumented classes (byte-code instrumentation), object-graph roots, and cluster-wide locks to maintain data coherence.

Terracotta DSO clusters differ from standard (non-DSO) clusters in certain important ways. With DSO:

- Objects are not serialized.
If your shared classes must be serialized, do not use DSO.
- All shared classes must meet portability requirements.
Non-portable classes cannot be shared and must be excluded using configuration.

- Clustered applications require a boot JAR to pre-instrument certain classes. The boot JAR file is platform-specific.
- Special integration files, called Terracotta Integration Modules (TIMs), are required to integrate with other technologies.
- Cluster-wide locking requirements are stricter and more extensive.
- A limited number of platforms are supported.

The threshold for successfully setting up a DSO cluster can be substantially higher than for a non-DSO cluster due to DSO's stricter code and configuration requirements. It is recommended that if possible you use the standard installation (also called *express* installation) to set up a non-DSO cluster. Use the DSO installation only if your deployment requires the features of DSO.

WARNING: Do Not Combine Installation Methods

You cannot combine the standard ("express" or non-DSO) and the DSO ("custom") installations. These two installation methods are incompatible and if combined cause errors at startup.

If you began with a standard install, then you *cannot* continue with the DSO install. If you began with the DSO install, then you *cannot* continue with the standard install. You must start with a fresh installation if switching between installation methods.

If you are new to Terracotta, see this [introduction to the Terracotta platform](#) before proceeding with the DSO installation. For more information on comparing standard and DSO installation methods, see [Standard Versus DSO Installations](#).

9.0.1 Standard Versus DSO Installations

There are two ways to install the Terracotta products: The standard installation, also called *express*, and the DSO installation, also called *custom*. Clusters based on the standard installation are much simpler and more flexible than those based on the DSO installation. The custom installation is for users who require DSO features such as Terracotta roots, preservation of object identity, or integration of other technologies using Terracotta Integration Modules (TIMs).

If you are using Ehcache on a single JVM, for example, or used cache replication for clustering, consider the standard installation (see [Enterprise Ehcache Installation](#)). If you are a current Terracotta user who requires DSO and distributed caching, it is recommended that you verify the need for DSO before continuing with the DSO installation given in this document.

If you are unsure about which installation path to choose, read both installation documents to find the one that meets your requirements. These installation paths are **not compatible and cannot be used in combination**.

9.0.2 Overview of Installation

This installation procedure is for users intending to install Enterprise Ehcache, Quartz Scheduler, or Terracotta Web Sessions. Instructions on integrating an application server (container) are also included. These products are independent of each other, but can be installed and run with clients using the same Terracotta Server Array.

The installation process involves these major tasks:

1. Have the JARs for the products you intend to install.
2. Edit the Terracotta configuration and the configuration (or script) files for products and containers.
3. Use `tim-get` to install the TIMs listed in the Terracotta configuration file.

Details on completing these tasks are provided in [9.1 Performing a DSO Installation](#).

9.1 Performing a DSO Installation

This document shows you how to perform a custom installation for clustering the following Terracotta products:

- Enterprise Ehcache
Includes Enterprise Ehcache for Hibernate (second-level cache for Hibernate)
- Quartz Scheduler
- Web Sessions

9.1.1 Prerequisites

- JDK 1.5 or higher.

See the [Certified Platforms page](#) for certified JVMs.

- [Terracotta 3.5.0 or higher](#)
Download the kit and run the installer on the machine that will host the Terracotta server, and on each application server (also called a Terracotta client). The kit contains compatible versions of Ehcache and Quartz.
- If you are using an application server, choose a certified server (see the [Certified Platforms page](#)).

For guaranteed compatibility, use the JAR files included with the Terracotta kit you are installing. Mixing with older components may cause errors or unexpected behavior. If you are using an Enterprise Edition kit, some JAR files will have "-ee-" as part of their name.

9.1.1a Enterprise Ehcache Users

Ehcache must be installed both for Enterprise Ehcache and Enterprise Ehcache for Hibernate (second-level cache for Hibernate). If you do not have Ehcache installed, a compatible version of Ehcache is available in the Terracotta kit. To install Ehcache, add the following JAR files to your application's classpath (or `WEB-INF/lib` directory if using a WAR file):

```
${TERRACOTTA_HOME}/ehcache/lib/ehcache-core-<ehcache-version>.jar
```

The Ehcache core libraries, where `<ehcache-version>` is the version of Ehcache (2.4.1 or higher).

- `${TERRACOTTA_HOME}/ehcache/lib/slf4j-api-<slf4j-version>.jar`
The SLF4J logging facade allows Ehcache to bind to any supported logger used by your application. Binding JARs for popular logging options are available from the [SLF4J project](#). For convenience, the binding JAR for `java.util.logging` is provided in `${TERRACOTTA_HOME}/ehcache` (see below).
- `${TERRACOTTA_HOME}/ehcache/lib/slf4j-jdk14-<slf4j-version>.jar`
An SLF4J binding JAR for use with the standard `java.util.logging`, also known as JDK 1.4 logging.
- You will also need to install Terracotta Integration Modules (TIMs) to allow Ehcache to run clustered. The required TIMs are described later in this procedure.
- Hibernate 3.2.5, 3.2.6, 3.2.7, 3.3.1, or 3.3.2 (Enterprise Ehcache for Hibernate only)
If you are clustering Enterprise Ehcache for Hibernate (second-level cache), be sure to use a compatible version of Hibernate in your application. Because sharing of Hibernate regions between different versions of Hibernate is not supported, be sure to use the same version of Hibernate throughout the cluster.

9.1.1b Quartz Scheduler Users

If you do not have Quartz installed, a compatible version of Quartz is available in the Terracotta kit. To install Quartz, add the following JAR file to your application's classpath (or `WEB-INF/lib` directory if using a WAR file):

- `${TERRACOTTA_HOME}/quartz/quartz-<quartz-version>.jar`
- The Quartz core libraries, where `<quartz-version>` is the version of quartz.
- You will also need to install Terracotta Integration Modules (TIMs) to allow Ehcache to run clustered. The required TIMs are described later in this procedure.

9.1.2 Step 1: Configure the Terracotta Platform

The Terracotta platform is the basis of the Terracotta cluster. You must configure the Terracotta servers and clients that form the cluster using the Terracotta configuration file (`tc-config.xml` by default). Start with a basic `tc-config.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- All content copyright Terracotta, Inc., unless otherwise indicated. All rights reserved. -->
<!-- This Terracotta configuration file is intended for use with Terracotta for Hibernate. -->
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd"
xmlns:tc="http://www.terracotta.org/config"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <servers>
    <!-- Shows where the Terracotta server can be found. -->
    <server host="localhost">
      <data>%(user.home)/terracotta/server-data</data>
      <logs>%(user.home)/terracotta/server-logs</logs>
    </server>
  </servers>
  <!-- Shows where to put the generated client logs -->
```

```

<clients>
  <logs>%(user.home)/terracotta/client-logs</logs>

  <!-- Names the Terracotta Integration Modules (TIM) needed for clustering specific technologies. -->
  <modules>
    <!-- Add TIMs here using <module name="tim-foo-<foo version>" /> elements. -->
  </modules>
</clients>
</tc:tc-config>

```

Save this file to `${TERRACOTTA_HOME}/tc-config.xml` on the host with the Terracotta server.

NOTE: Locating and Naming tc-config.xml

This procedure assumes you name the Terracotta configuration file `tc-config.xml` and save it to `${TERRACOTTA_HOME}`. If you give the file a different name and locate it elsewhere, you must adjust the name and paths shown in this procedure accordingly.

9.1.2a TIMs for Clustering Enterprise Ehcache

To cluster Enterprise Ehcache or Enterprise Ehcache for Hibernate, add the following element to the `<modules>` subsection of the `<clients>` section:

```
<module name="tim-ehcache-2.0" />
```

The module shown is for Ehcache 2.0. Note that the version shown at the end of the module name must match the version of Ehcache being used. For example, to integrate with Ehcache 1.7.2, add:

```
<module name="tim-ehcache-1.7" />
```

You must use Ehcache version 1.7.2 or higher. The Terracotta kit contains a compatible version of Ehcache and it is recommended that you use that version by adding the provided Ehcache JAR files to your application's classpath.

9.1.2b TIMs for Clustering Quartz Scheduler

To cluster Quartz Scheduler, add the following element to the `<modules>` subsection of the `<clients>` section

```
<module name="tim-quartz-1.7" />
```

The module shown is for Quartz 1.7.x. Note that the version shown at the end of the module name must match the version of Quartz being used. You must use Quartz version 1.5.1 or higher. The Terracotta kit contains a compatible version of Quartz and it is recommended that you use that version by adding the provided Quartz JAR file to your application's classpath.

9.1.2c TIMs for Integrating an Application Server

To integrate an application server, add the following element to the `<modules>` subsection of the `<clients>` section:

```
<module name="tim-<app-server>-<app-server-version>" />
```

For example, to use Tomcat 6.0, add:

```
<module name="tim-tomcat-6.0"/>
```

See the following table for a full list of certified application-server TIMs.

Container	Terracotta Integration Module Name
GlassFish v1	tim-glassfish-v1
GlassFish v2	tim-glassfish-v2
JBoss Application Server 4.0	tim-jboss-4.0
JBoss Application Server 4.2	tim-jboss-4.2
JBoss Application Server 5.1	tim-jboss-5.1
Jetty 6.1	tim-jetty-6.1
Tomcat 5.0	tim-tomcat-5.0
Tomcat 5.5	tim-tomcat-5.5

Tomcat 6.0	tim-tomcat-6.0
WebLogic 9	tim-weblogic-9
WebLogic 10	tim-weblogic-10

To integrate your chosen application server, see the following sections.

Tomcat, JBoss Application Server, Jetty, WebLogic

Integrate Terracotta by adding the following to the top of the appropriate startup script for the chosen container, or to a configuration file used by the startup script:

UNIX/Linux

```
TC_INSTALL_DIR=path/to/local/terracotta_home
TC_CONFIG_PATH=path/to/tc-config.xml
. ${TC_INSTALL_DIR}/platform/bin/dso-env.sh -q
export JAVA_OPTS="$JAVA_OPTS $TC_JAVA_OPTS"
```

Microsoft Windows

```
set TC_INSTALL_DIR=path\to\local\terracotta_home>
set TC_CONFIG_PATH=path\to\local\tc-config.xml
call %TC_INSTALL_DIR%\platform\bin\dso-env.bat -q
set JAVA_OPTS=%JAVA_OPTS% %TC_JAVA_OPTS%
```

The following table lists suggested container script files to use:

NOTE: JAVA_OPTIONS and JAVA_OPTS

Your container may use JAVA_OPTIONS instead of JAVA_OPTS.

For This Container	Add Terracotta Configuration to this File	Notes
JBoss Application Server	run.conf	run.conf is called by both run.sh and run.bat.
Jetty	jetty.sh	Jetty can be configured in a number of different ways. See the comments in the jetty.sh startup script for more information on how to set the Jetty environment.
Tomcat	setenv.sh or setenv.bat	The setenv script is called by catalina.sh or catalina.bat if it exists in the same directory.
WebLogic	setEnv.sh or setEnv.bat	The setEnv script is called by startWeblogic.sh or startWeblogic.bat. See the startWeblogic script to find or edit the location of the setEnv script.

GlassFish

GlassFish uses a multi-step process for starting the application server instances. To ensure that Terracotta runs in the same JVM as the application server, add these startup flags to the GlassFish domain.xml (found under the domains/<your_domain>/config directory):

```
<jvm-options>-Dcom.sun.enterprise.server.ss.ASQuickStartup=false</jvm-options>
<jvm-options>-Dtc.config=<path/to/Terracotta_configuration_file></jvm-options>
<jvm-options>-Dtc.install-root=<path/to/Terracotta_home></jvm-options>
<jvm-options>-Xbootclasspath/p:<path/to/DSO_boot_jar></jvm-options>
```

The last JVM option contains a boot-jar path. Run the following command from \${TERRACOTTA_HOME} on one of the application servers to find the boot-jar path:

UNIX/Linux

```
[PROMPT] platform/bin/make-boot-jar.sh
```

Microsoft Windows

```
[PROMPT] platform\bin\make-boot-jar.bat
```

You should see output similar to the following (shown for Linux):

```
2009-06-24 09:43:54,961 INFO - Configuration loaded from the file at '/Users/local/terracotta-3.0.0/tc-
config.xml'.
Creating boot JAR at '/Users/local/terracotta-3.0.0/platform/bin/./lib/dso-boot/dso-
boot-hotspot_linux_150_16.jar...
Successfully created boot JAR file at '/Users/local/terracotta-3.0.0/platform/bin/./lib/dso-boot/dso-
boot-hotspot_linux_150_16.jar'.
```

Note the relative path given, which in this example is `/Users/local/terracotta-3.0.0/platform/bin/./lib/dso-boot/dso-boot-hotspot_linux_150_16.jar`. The inferred path, `/Users/local/terracotta-3.0.0/lib/dso-boot/dso-boot-hotspot_linux_150_16.jar` is needed for the value of the `domain.xml` element `<jvm-options>-Xbootclasspath/p:<path/to/DSO_boot_jar></jvm-options>`.

TIP: Using Startup Flags in `domain.xml`

`domain.xml` uses `<jvm-options>` elements to pass the required flags. You can add other startup flags, such as `-Dcom.tc.session.cookie.domain`, to `domain.xml`.

If the setup on your application servers is the same, you can use the path output from one application server to configure the others. However, if the setup on your application servers varies, you may have to run `make-boot-jar` on each application server to find the appropriate path.

For example, an installation on Linux where clients received their configuration from a server could use startup flags similar to the following:

```
<jvm-options>-Dcom.sun.enterprise.server.ss.ASQuickStartup=false</jvm-options>
<jvm-options>-Dtc.config=server1:9510</jvm-options>
<jvm-options>-Dtc.install-root=/myHome/tc3.0</jvm-options>
<jvm-options>-Xbootclasspath/p:/myHome/tc3.0/lib/dso-boot/dso-boot-hotspot_linux_160_06.jar</jvm-options>
```

9.1.2d Clustering a Web Application with Terracotta Web Sessions

To cluster a web application, you must add the following `<web-applications>` subsection to the `<application>` section of `tc-config.xml`:

```
<!-- The application section is at the same level as the servers and clients sections. -->
<application>
...
  <web-applications>
    <web-application>myWebApp</web-application>
  </web-applications>
...
</application>
```

The value of `<web-application>` is the application context root or the name of the application's WAR file.

9.1.3 Step 2: Configure Terracotta Products

The following sections show you how to configure the following Terracotta products:

- [Enterprise Ehcache Configuration](#)
- [Enterprise Ehcache for Hibernate Configuration](#)
- [Quartz Scheduler Configuration](#)
- [Web Sessions Configuration](#)

9.1.3a Enterprise Ehcache Configuration

Each instance of the distributed cache must have an Ehcache configuration file. The Ehcache configuration file, `ehcache.xml` by default, should be on your application's classpath. If you are using a WAR file, add the Ehcache configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.

TIP: Distributed Ehcache for Hibernate

Terracotta Distributed Ehcache for Hibernate also uses `ehcache.xml`.

Sample Ehcache Configuration File

The Ehcache configuration file configures the caches that you want to cluster. The following is a sample `ehcache.xml` file:

```
<ehcache xsi:noNamespaceSchemaLocation="ehcache.xsd" name="myCacheMan">
  <defaultCache maxElementsInMemory="10000" eternal="false"
    timeToIdleSeconds="120" timeToLiveSeconds="120" overflowToDisk="true"
    diskSpoolBufferSizeMB="30" maxElementsOnDisk="10000000"
    diskPersistent="false" diskExpiryThreadIntervalSeconds="120"
    memoryStoreEvictionPolicy="LRU"/>
  <cache name="foo" maxElementsInMemory="1000"
    maxElementsOnDisk="10000" eternal="false" timeToIdleSeconds="3600"
    timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
    <!-- Adding the element <terracotta /> turns on Terracotta clustering for the cache "foo". -->
    <terracotta clustered="true" valueMode="identity"/>
  </cache>
</ehcache>
```

NOTE: Understanding the Cache Mode (valueMode)

The `<terracotta>` element's `valueMode` attribute sets the cache mode to serialization or identity. Before choosing a cache mode, be sure to understand the functions, effects, and requirements of serialization and identity modes. See [Comparing Serialization and Identity Modes](#) for more information.

Using the Cache in Your Application

In your application, the distributed cache is set up by creating the `CacheManager`, which references the Ehcache configuration file. There are a number of ways to have your application locate the Ehcache configuration file, some of which have been noted above.

The following example code shows how to use the cache configured in the `ehcache.xml` file shown above

```
import net.sf.ehcache.CacheManager;
import net.sf.ehcache.Cache;
import net.sf.ehcache.Element;
// Look up cache manager and cache. This assumes that the app can find the
// Ehcache configuration file. Note that "foo" in getEhcache() corresponds to
// name given to a cache block in the Ehcache configuration file.
CacheManager cacheManager = new CacheManager();
Cache cache = cacheManager.getEhcache("foo");
// Put element in cache
cache.put(new Element("key", "value"));
// Get element from cache
Element element = cache.get("key");
```

As an option to using the Ehcache configuration file, you can also create the cache programmatically:

```
public Cache(String name,
    int maxElementsInMemory,
    MemoryStoreEvictionPolicy memoryStoreEvictionPolicy,
    boolean eternal,
    long timeToLiveSeconds,
    long timeToIdleSeconds,
    int maxElementsOnDisk,
    boolean isTerracottaClustered,
    String terracotta ValueMode)
```

For more information on the Ehcache configuration file, instantiating the `CacheManager`, and programmatic approaches see the [Ehcache documentation](#).

Incompatible Configuration

Do not use the element `<terracottaConfig>` in `ehcache.xml`.

For any clustered cache, you cannot use configuration elements that are incompatible when clustering with Terracotta. Clustered caches have a `<terracotta>` element.

The following Ehcache configuration attributes or elements should not be used in clustered caches:

- DiskStore-related attributes `overflowToDisk`, `overflowToOffHeap`, and `diskPersistent`.
The Terracotta server automatically provides a disk store.
- Replication-related configuration elements, such as `<cacheManagerPeerProviderFactory>`, `<cacheManagerPeerListenerFactory>`, `<bootstrapCacheLoaderFactory>`, `<cacheEventListenerFactory>`.

When a change occurs in a Terracotta cluster, all nodes that have the changed element or object are automatically updated. Unlike the replication methods used to cluster Ehcache, cache event listeners are not (and do not need to be) notified of remote changes. Listeners are still aware of local changes

- Replication-related attributes such as `replicateAsynchronously` and `replicatePuts`.

If you use the attribute `MemoryStoreEvictionPolicy`, it must be set to either LRU or LRU. Setting `MemoryStoreEvictionPolicy` to FIFO causes the error `IllegalArgumentException`.

9.1.3b Enterprise Ehcache for Hibernate Configuration

Each instance of the distributed second-level cache for Hibernate must have an Ehcache configuration file. The Ehcache configuration file, `ehcache.xml` by default, should be on your application's classpath. If you are using a WAR file, add the Ehcache configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`.

TIP: Distributed Ehcache

Terracotta Distributed Ehcache also uses `ehcache.xml`.

See [Incompatible Configuration](#) for configuration elements that must be avoided.

Sample Ehcache Configuration File

Create a basic Ehcache configuration file, `ehcache.xml` by default:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache name="myCache"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="ehcache.xsd">
  <defaultCache
    maxElementsInMemory="0"
    eternal="false"
    timeToIdleSeconds="1200"
    timeToLiveSeconds="1200">
    <terracotta />
  </defaultCache>
</ehcache>
```

This defaultCache configuration includes Terracotta clustering. The Terracotta client must load the configuration from a file or a Terracotta server.

TIP: Terracotta Clients and Servers

In a Terracotta cluster, the application server is also known as the client.

The source of the Terracotta client configuration is specified in the application server (see [TIMs for Integrating an Application Server](#)). It can also be specified on the command line when the application is started using the `tc.Config` property. For example:

```
-Dtc.Config=localhost:9510
```

Cache-Specific Configuration

Using an Ehcache configuration file with only a defaultCache configuration means that every cached Hibernate entity is cached with the settings of that defaultCache. You can create specific cache configurations for Hibernate entities using `<cache>` elements.

For example, add the following `<cache>` block to `ehcache.xml` to cache a Hibernate entity that has been configured for caching (see [Step 3: Prepare Your Application for Caching](#)):

```
<cache name="com.my.package.Foo" maxElementsInMemory="1000"
maxElementsOnDisk="10000" eternal="false" timeToIdleSeconds="3600"
timeToLiveSeconds="0" memoryStoreEvictionPolicy="LRU">
  <!-- Adding the element <terracotta /> turns on Terracotta clustering for the cache Foo. -->
  <terracotta />
</cache>
```

You can edit the eviction settings in the defaultCache and any other caches that you configure in `ehcache.xml` to better fit your application's requirements.

Enabling Second-Level Cache in Hibernate

You must also enable the second-level cache and specify the provider in the Hibernate configuration. For more information, see [Hibernate Configuration File](#).

9.1.3c Quartz Scheduler Configuration

Quartz is configured programmatically or by a Quartz configuration file (`quartz.properties` by default). If no configuration is provided, a default configuration is loaded. The following shows the contents of the default configuration file:

```
# Default Properties file for use by StdSchedulerFactory
# to create a Quartz Scheduler Instance, if a different
# properties file is not explicitly specified.
#

org.quartz.scheduler.instanceName = DefaultQuartzScheduler
org.quartz.scheduler.rmi.export = false
org.quartz.scheduler.rmi.proxy = false
org.quartz.scheduler.wrapJobExecutionInUserTransaction = false

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 10
org.quartz.threadPool.threadPriority = 5
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread = true

org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

To cluster with Terracotta, you must edit the property `org.quartz.jobStore.class` to specify the Terracotta Job Store for Quartz instead of `org.quartz.simpl.RAMJobStore`:

```
org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore
```

The Quartz configuration file must be on your application's classpath. If you are using a WAR file, add the Quartz configuration file to `WEB-INF/classes` or to a JAR file that is included in `WEB-INF/lib`. For more information on configuring Quartz, including use of the Quartz API, see the Quartz documentation at <http://www.quartz-scheduler.org>.

9.1.3d Web Sessions Configuration

Clustered sessions are configured from `tc-config.xml`. For more on the <web-applications> section of `tc-config.xml`, see the [Terracotta Configuration Guide and Reference](#).

9.1.4 Step 3: Install the TIMs

The TIMs specified in `tc-config.xml` must be installed on each Terracotta client. If you are using Ehcache or Quartz, TIMs associated with Ehcache or Quartz must be added to your application's classpath. Install the TIM JAR files using the following command:

Unix/Linux

```
${TERRACOTTA_HOME}/bin/tim-get.sh install-for path/to/tc-config.xml
```

MICROSOFT WINDOWS

```
${TERRACOTTA_HOME}\bin\tim-get.bat install-for path\to\tc-config.xml
```

Be sure to target the Terracotta configuration file you modified with the TIM <module> elements. `tim-get` will print a status for each TIM it attempts to install as well as all dependencies.

For example, if you added TIMs for Ehcache 2.0.0 and Tomcat 6.0, output similar to the following should appear:

```
Parsing module: tim-ehcache-2.0:latest
Parsing module: tim-tomcat-6.0:latest
Installing tim-ehcache-2.0 1.5.1 and dependencies...
  INSTALLED: tim-ehcache-2.0 1.5.1 - Ok
  INSTALLED: terracotta-toolkit-1.0 1.0.0 - Ok
  INSTALLED: tim-ehcache-2.0-hibernate-ui 1.5.1 - Ok
Installing tim-tomcat-6.0 2.1.1 and dependencies...
  INSTALLED: tim-tomcat-6.0 2.1.1 - Ok
  INSTALLED: tim-tomcat-5.5 2.1.1 - Ok
  INSTALLED: tim-tomcat-common 2.1.1 - Ok
  SKIPPED: tim-session-common 2.1.1 - Already installed
  SKIPPED: terracotta-toolkit-1.0 1.0.0 - Already installed
```

Done.

Of the TIMs shown for the Ehcache 2.0 portion of the `tim-get` output, the following must be added to your

application's classpath:

- tim-ehcache-2.0
- terracotta-toolkit-<API version> or terracotta-toolkit-<API version>-ee

For Quartz Scheduler, the TIMs that must be added to the application's classpath are:

- tim-quartz-<version>
- terracotta-toolkit-<API version> or terracotta-toolkit-<API version>-ee

where <version> is the version of Quartz Scheduler.

If you are clustering sessions, there is no explicit requirement for placing container or sessions-related TIMs on the classpath.

If you install both open-source TIMs and Enterprise Edition TIMs, then you must specify both types of Terracotta Toolkit JARs in the Terracotta configuration file. For example, if you want to install

tim-tomcat-6.0 and tim-ehcache-2.x-ee, then specify the following:

```
<modules>
  <module group-id="org.terracotta.toolkit" name="terracotta-toolkit-1.2" />
  <module group-id="org.terracotta.toolkit" name="terracotta-toolkit-1.2-ee" />
  <module name="tim-tomcat-6.0" />
  <module name="tim-ehcache-2.x-ee" />
<!-- Other TIMs here. -->
</modules>
```

The Terracotta Toolkit API version available for your Terracotta kit may be different than the one shown in this example.

9.1.4a Location of TIMs

Generally, TIMs are found on the following path:

```
${TERRACOTTA_HOME}/platform/modules/org/terracotta/modules/tim-<name>-<version>/<TIM-version>
/tim-<name>-<version>-<TIM-version>.jar
```

where <name> is the name of the technology being integrated, and <version> is the version of that technology (if applicable). For example, the path to the TIM for Ehcache 2.0, which in this example has the TIM version 1.5.1, is as shown:

```
${TERRACOTTA_HOME}/platform/modules/org/terracotta/modules/tim-ehcache-2.0/1.5.1/tim-ehcache-2.0-1.5.1.jar
```

The Terracotta Toolkit (terracotta-toolkit) is found in:

```
${TERRACOTTA_HOME}/platform/modules/org/terracotta/toolkit/terracotta-toolkit-1.0/1.0.0/terracotta-toolkit-1.0-1.0.0.jar
```

9.1.5 Step 4: Start the Cluster

1. Start the Terracotta server:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/start-tc-server.sh &
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\start-tc-server.bat
```

2. Start the application servers.
3. Start the Terracotta Developer Console:

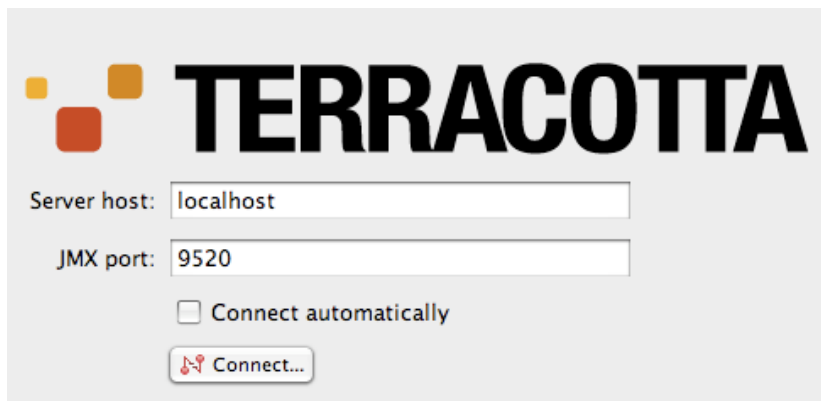
UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

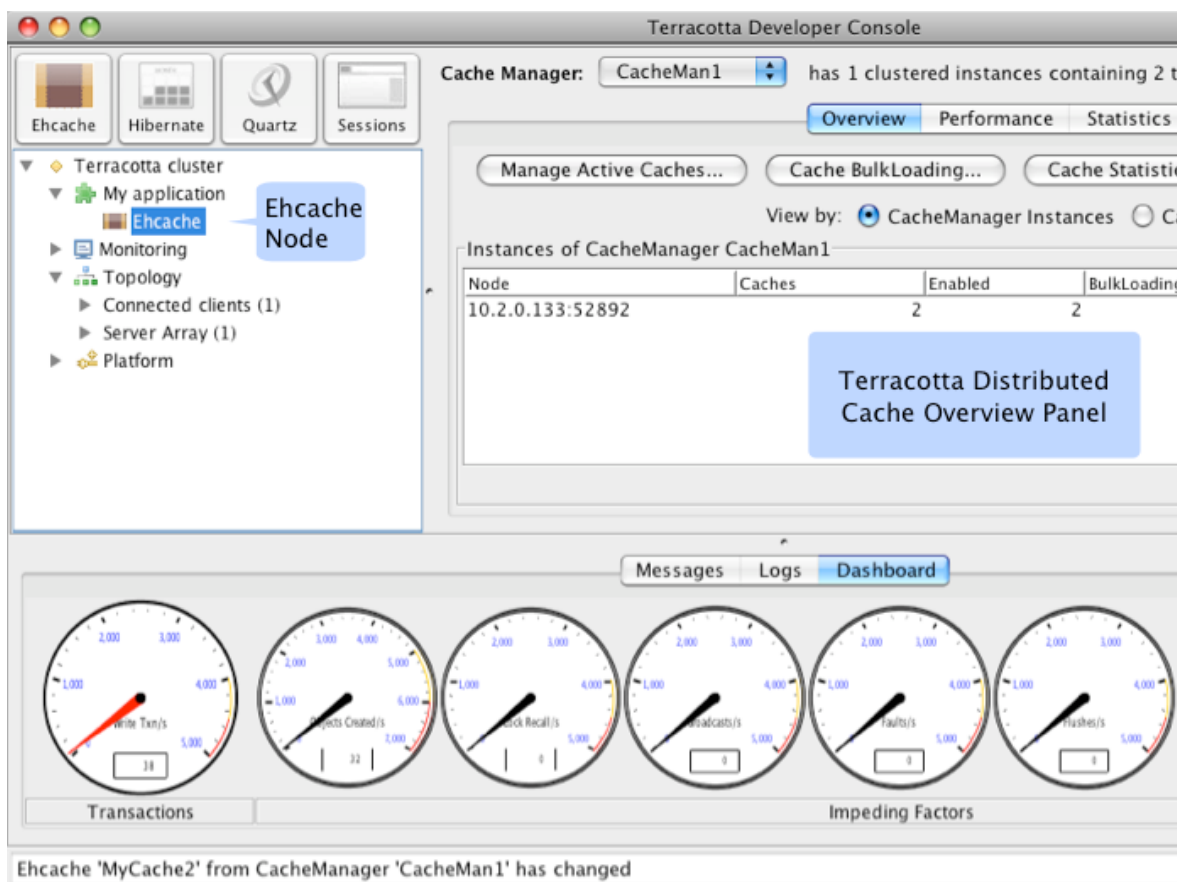
Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

4. Connect to the Terracotta cluster.
Click **Connect...** in the Terracotta Developer Console.



5. Click the **Topology** node in the cluster navigation window to see the Terracotta servers and clients (application servers) in the Terracotta cluster.
6. If you are clustering Ehcache or Ehcache for Hibernate, click the **My Application** node in the cluster navigation window to see panels for these products.
For example, if you are clustering Ehcache, click the Ehcache node in the cluster navigation window to see the caches in the Terracotta cluster.



9.1.6 Quartz Scheduler DSO Installation

Quartz Scheduler Where is an Enterprise feature that allows jobs and triggers to be run on specified Terracotta clients instead of randomly chosen ones. For more information on the Quartz Scheduler Where locality API, see [4.2.1 Quartz Scheduler Where \(Locality API\)](#).

DSO users must install tim-quartz-2.0-ee. First, add the TIM to your Terracotta configuration file (tc-config.xml by default):

```
...
<clients>
...
  <modules>
    <module name="tim-quartz-2.0-ee" />
  </modules>
</clients>
```

```
...  
</modules>  
...  
</clients>  
...
```

To install the TIMs declared in the Terracotta configuration file, use the following command:

UNIX/Linux

```
${TERRACOTTA_HOME}/bin/tim-get.sh install-for /path/to/tc-config.xml
```

Use `tim-get.bat` with Microsoft Windows.

[Top of Page](#)