



# Terracotta 3.7.4 Documentation

|   |        |
|---|--------|
| Welcome to BigMemory Go.....                            | 1/453  |
| BigMemory Go lets you put ALL your data in memory,..... | 1/453  |
| limited only by how much RAM you have.....              | 1/453  |
| Getting Started With BigMemory Go.....                  | 2/453  |
| Installing BigMemory Go.....                            | 2/453  |
| Configuring BigMemory Go.....                           | 3/453  |
| Automatic Resource Control.....                         | 3/453  |
| Fast Restartability.....                                | 4/453  |
| Using the BigMemory Go API.....                         | 4/453  |
| Search.....   | 4/453  |
| Transactional Caching.....                              | 4/453  |
| Administration and Monitoring.....                      | 4/453  |
| Learn More About How BigMemory Go Works.....            | 4/453  |
| Code Samples.....                                       | 5/453  |
| Introduction.....                                       | 5/453  |
| Example 1: Declarative Configuration via XML.....       | 5/453  |
| Example 2: Programmatic Configuration.....              | 6/453  |
| Example 3: Create, Read, Update and Delete (CRUD).....  | 6/453  |
| Example 4: Search.....                                  | 8/453  |
| Example 5: Automatic Resource Control (ARC).....        | 10/453 |
| Example 6: Using BigMemory As a Cache.....              | 10/453 |
| Configuration Overview.....                             | 12/453 |
| Introduction.....                                       | 12/453 |
| XML Configuration.....                                  | 12/453 |
| Dynamically Changing Cache Configuration.....           | 12/453 |
| Passing Copies Instead of References.....               | 13/453 |
| Special System Properties.....                          | 14/453 |
| net.sf.ehcache.disabled.....                            | 14/453 |
| net.sf.ehcache.use.classic.lru.....                     | 14/453 |
| ehcache.xsd.....  | 14/453 |
| ehcache-failsafe.xml.....                               | 14/453 |
| About Default Cache.....                                | 15/453 |
| More Information on Configuration Topics.....           | 15/453 |
| Storage Tiers Basics.....                               | 16/453 |
| Introduction.....                                       | 16/453 |
| Memory Store.....                                       | 16/453 |
| Off-Heap Store.....                                     | 16/453 |
| Allocating Direct Memory in the JVM.....                | 16/453 |
| Tuning Off-Heap Store Performance.....                  | 18/453 |
| Swapiness and Huge Pages.....                           | 19/453 |
| Disk Store.....   | 20/453 |
| Serialization.....                                      | 20/453 |
| Storage Options.....                                    | 20/453 |

# Terracotta 3.7.4 Documentation

|  |               |
|--|---------------|
| <b>Storage Tiers Advanced.....</b>                                 | <b>21/453</b> |
| Introduction.....  | 21/453        |
| Configuring the Memory Store.....                                  | 21/453        |
| Memory Use, Spooling, and Expiry Strategy in the Memory Store..... | 21/453        |
| Configuring the Off-Heap Store.....                                | 22/453        |
| Declarative Configuration.....                                     | 22/453        |
| Programmatic Configuration.....                                    | 23/453        |
| Configuring the Disk Store.....                                    | 23/453        |
| Disk-Store Storage Options.....                                    | 23/453        |
| Disk Store Configuration Element.....                              | 24/453        |
| Disk Store Expiry and Eviction.....                                | 24/453        |
| Turning off Disk Stores.....                                       | 25/453        |
| Configuration Examples.....  | 25/453        |
| <b>Sizing Storage Tiers.....</b>                                   | <b>27/453</b> |
| Introduction.....  | 27/453        |
| Sizing Attributes.....   | 27/453        |
| Pooling Resources Versus Sizing Individual Data Sets.....          | 27/453        |
| Memory Store (Heap).....   | 28/453        |
| Off-Heap Store.....  | 28/453        |
| Disk Store.....  | 28/453        |
| Sizing Examples.....   | 29/453        |
| Pooled Resources.....  | 29/453        |
| Explicitly Sizing Data Sets.....                                   | 29/453        |
| Mixed Sizing Configurations.....                                   | 30/453        |
| Using Percents.....  | 30/453        |
| Sizing Data Sets Without a Pool.....                               | 31/453        |
| Overflows.....   | 31/453        |
| Overriding Size Limitations.....                                   | 32/453        |
| Built-In Sizing Computation and Enforcement.....                   | 32/453        |
| Sizing of Data Set Entries.....                                    | 32/453        |
| Eviction When Using CacheManager-Level Storage.....                | 34/453        |
| <b>Pinning, Expiration, and Eviction.....</b>                      | <b>36/453</b> |
| Introduction.....  | 36/453        |
| Setting Expiration.....  | 36/453        |
| Pinning Data.....  | 36/453        |
| Pinning Individual Cache Entries.....                              | 37/453        |
| Pinning a Cache.....   | 37/453        |
| Scope of Pinning.....  | 38/453        |
| How Configuration Affects Element Flushing and Eviction.....       | 38/453        |
| Pinning Overrides Cache Sizing.....                                | 38/453        |
| Data Freshness and Expiration.....                                 | 39/453        |
| <b>Fast Restartability.....</b>                                    | <b>40/453</b> |
| Introduction.....  | 40/453        |
| Data Persistence Implementation.....                               | 40/453        |
| Strategy Options.....  | 40/453        |

# Terracotta 3.7.4 Documentation

|   |               |
|---|---------------|
| <b>Fast Restartability</b>                                  |               |
| Synchronous Writes Options.....                             | 41/453        |
| Disk Store Path.....  | 41/453        |
| Configuration Examples.....                                 | 41/453        |
| Options for Crash Resilience.....                           | 41/453        |
| Temporary Disk Storage.....                                 | 42/453        |
| In-memory Only.....   | 42/453        |
| Programmatic Configuration Example.....                     | 42/453        |
| Fast Restart Performance.....                               | 43/453        |
| Fast Restart Limitations.....                               | 43/453        |
| <br>  |               |
| <b>Key Classes and Methods of the BigMemory Go API.....</b> | <b>44/453</b> |
| Introduction.....   | 44/453        |
| CacheManager.....   | 44/453        |
| CacheManager Creation Modes.....                            | 44/453        |
| Cache.....  | 45/453        |
| Element.....  | 45/453        |
| <br>  |               |
| <b>BigMemory Go Search API.....</b>                         | <b>47/453</b> |
| Introduction.....   | 47/453        |
| What is Searchable?.....                                    | 47/453        |
| Making a Cache Searchable.....                              | 47/453        |
| By Configuration.....                                       | 47/453        |
| Defining Attributes.....                                    | 48/453        |
| Well-known Attributes.....                                  | 49/453        |
| Creating a Query.....                                       | 51/453        |
| Using Attributes in Queries.....                            | 52/453        |
| Expressions.....  | 52/453        |
| List of Operators.....                                      | 52/453        |
| Making Queries Immutable.....                               | 53/453        |
| Obtaining and Organizing Query Results.....                 | 53/453        |
| Aggregators.....  | 53/453        |
| Ordering Results.....                                       | 53/453        |
| Grouping Results.....                                       | 54/453        |
| Limiting the Size of Results.....                           | 54/453        |
| Interrogating Results.....                                  | 55/453        |
| Finding Null (or Not Null) Values.....                      | 55/453        |
| Sample Application.....                                     | 55/453        |
| Scripting Environments.....                                 | 55/453        |
| Implementation and Performance.....                         | 56/453        |
| Best Practices for Optimizing Searches.....                 | 56/453        |
| Concurrency Notes.....                                      | 57/453        |
| <br>  |               |
| <b>Transactions in Ehcache.....</b>                         | <b>59/453</b> |
| Introduction.....   | 59/453        |
| All or nothing.....   | 59/453        |
| Transactional Methods.....                                  | 59/453        |
| Change Visibility.....                                      | 59/453        |

# Terracotta 3.7.4 Documentation

|   |               |
|---|---------------|
| <b>Transactions in Ehcache</b>  |               |
| When to use transactional modes.....  | 60/453        |
| Requirements.....   | 60/453        |
| Configuration.....  | 60/453        |
| Transactional Caches with Spring.....   | 61/453        |
| Global Transactions.....  | 61/453        |
| Implementation.....   | 61/453        |
| Failure Recovery.....   | 61/453        |
| Recovery.....   | 61/453        |
| Sample Apps.....  | 62/453        |
| XA Sample App.....  | 62/453        |
| XA Banking Application.....   | 62/453        |
| Transaction Managers.....   | 63/453        |
| Automatically Detected Transaction Managers.....                              | 63/453        |
| Configuring a Transaction Manager.....  | 63/453        |
| Local Transactions.....   | 63/453        |
| Introduction Video.....   | 64/453        |
| Configuration.....  | 64/453        |
| Isolation Level.....  | 64/453        |
| Transaction Timeouts.....   | 64/453        |
| Sample Code.....  | 65/453        |
| Performance.....  | 65/453        |
| Managing Contention.....  | 65/453        |
| What granularity of locking is used?.....                                     | 65/453        |
| Performance Comparisons.....  | 65/453        |
| FAQ.....  | 66/453        |
| Why do some threads regularly time out and throw an exception?.....           | 66/453        |
| Is IBM Websphere Transaction Manager supported?.....                          | 66/453        |
| How do transactions interact with Write-behind and Write-through caches?..... | 66/453        |
| Are Hibernate Transactions supported?.....                                    | 66/453        |
| How do I make WebLogic 10 work with transactional Ehcache?.....               | 66/453        |
| How do I make Atomikos work with the Ehcache "xa" mode?.....                  | 67/453        |
| <b>Explicit Locking.....</b>  | <b>68/453</b> |
| Introduction.....   | 68/453        |
| The API.....  | 68/453        |
| Example.....  | 69/453        |
| How it works.....   | 69/453        |
| <b>Blocking Cache and Self-Populating Cache.....</b>                          | <b>71/453</b> |
| Introduction.....   | 71/453        |
| Blocking Cache.....   | 71/453        |
| SelfPopulatingCache.....  | 71/453        |
| <b>Cache Decorators.....</b>  | <b>72/453</b> |
| Introduction.....   | 72/453        |
| Creating a Decorator.....   | 72/453        |
| Programmatically.....   | 72/453        |

# Terracotta 3.7.4 Documentation

|   |               |
|---|---------------|
| <b>Cache Decorators</b>   |               |
| By Configuration.....   | 72/453        |
| Adding decorated caches to the CacheManager.....  | 72/453        |
| Using CacheManager.replaceCacheWithDecoratedCache().....                                | 73/453        |
| Using CacheManager.addDecoratedCache().....   | 73/453        |
| Built-in Decorators.....  | 74/453        |
| BlockingCache.....  | 74/453        |
| SelfPopulatingCache.....  | 74/453        |
| Caches with Exception Handling.....   | 74/453        |
| <b>Event Listeners.....</b>   | <b>75/453</b> |
| CacheManager Event Listeners.....   | 75/453        |
| Configuration.....  | 75/453        |
| Implementing a CacheManager Event Listener Factory and CacheManager Event Listener..... | 75/453        |
| Cache Event Listeners.....  | 77/453        |
| Configuration.....  | 77/453        |
| Implementing a Cache Event Listener Factory and Cache Event Listener.....               | 78/453        |
| Adding a Listener Programmatically.....   | 80/453        |
| <b>Cache Exception Handlers.....</b>  | <b>81/453</b> |
| Introduction.....   | 81/453        |
| Declarative Configuration.....  | 81/453        |
| Implementing a Cache Exception Handler Factory and Cache Exception Handler.....         | 81/453        |
| Programmatic Configuration.....   | 82/453        |
| <b>Cache Extensions.....</b>  | <b>83/453</b> |
| Introduction.....   | 83/453        |
| Declarative Configuration.....  | 83/453        |
| Implementing a Cache Extension Factory and Cache Extension.....                         | 83/453        |
| Programmatic Configuration.....   | 85/453        |
| <b>Cache Eviction Algorithms.....</b>   | <b>86/453</b> |
| Introduction.....   | 86/453        |
| Provided MemoryStore Eviction Algorithms.....   | 86/453        |
| Least Recently Used (LRU).....  | 86/453        |
| Least Frequently Used (LFU).....  | 86/453        |
| First In First Out (FIFO).....  | 87/453        |
| Plugging in your own Eviction Algorithm.....  | 87/453        |
| Disk Store Eviction Algorithm.....  | 88/453        |
| <b>Class Loading and Class Loaders.....</b>   | <b>89/453</b> |
| Introduction.....   | 89/453        |
| Plugin Class Loading.....   | 89/453        |
| Loading of ehcache.xml resources.....   | 90/453        |

# Terracotta 3.7.4 Documentation

|  |                |
|--|----------------|
| <b>The Terracotta Management Console.....</b>                            | <b>91/453</b>  |
| Using Multiple Instances of BigMemory Go CacheManagers With the TMC..... | 91/453         |
| The TMC Update Checker.....  | 91/453         |
| <br>   |                |
| <b>Terracotta Management Console Security Setup.....</b>                 | <b>92/453</b>  |
| Introduction.....  | 92/453         |
| No Security.....   | 92/453         |
| Default Security.....  | 92/453         |
| Basic Connection Security.....   | 93/453         |
| Creating Individual Keychain Files.....                                  | 94/453         |
| Adding SSL.....  | 95/453         |
| Certificate-Based Client Authentication.....                             | 95/453         |
| Forcing SSL connections For TMC Clients.....                             | 97/453         |
| About the Default Keystore.....  | 97/453         |
| <br>   |                |
| <b>Management and Monitoring using JMX.....</b>                          | <b>98/453</b>  |
| Introduction.....  | 98/453         |
| JMX Overview.....  | 98/453         |
| MBeans.....  | 98/453         |
| JMX Remoting.....  | 99/453         |
| ObjectName naming scheme.....  | 99/453         |
| The Management Service.....  | 99/453         |
| JConsole Example.....  | 101/453        |
| Performance.....   | 101/453        |
| <br>   |                |
| <b>Logging.....</b>  | <b>102/453</b> |
| Introduction.....  | 102/453        |
| SLF4J Logging.....   | 102/453        |
| Concrete Logging Implementation Use in Maven.....                        | 102/453        |
| Concrete Logging Implementation Use in the Download Kit.....             | 102/453        |
| Recommended Logging Levels.....  | 102/453        |
| <br>   |                |
| <b>Shutting Down BigMemory.....</b>                                      | <b>103/453</b> |
| Introduction.....  | 103/453        |
| ServletContextListener.....  | 103/453        |
| The Shutdown Hook.....   | 103/453        |
| Dirty Shutdown.....  | 104/453        |
| <br>   |                |
| <b>Using Hibernate and BigMemory Go.....</b>                             | <b>105/453</b> |
| Introduction.....  | 105/453        |
| Download and Install.....  | 105/453        |
| Build with Maven.....  | 105/453        |
| Configure BigMemory Go as the Second-Level Cache Provider.....           | 106/453        |
| Hibernate 3.3.....   | 106/453        |
| Hibernate 4.x.....   | 106/453        |
| Enable Second-Level Cache and Query Cache Settings.....                  | 106/453        |
| Optional.....  | 107/453        |
| Configuration Resource Name.....   | 107/453        |

# Terracotta 3.7.4 Documentation

|   |                |
|---|----------------|
| <b>Using Hibernate and BigMemory Go</b>   |                |
| Set the Hibernate cache provider programmatically.....  | 107/453        |
| Putting it all together.....  | 107/453        |
| Configure Hibernate Entities to use Second-Level Caching.....   | 107/453        |
| Definition of the different cache strategies.....   | 108/453        |
| Configure.....  | 109/453        |
| Domain Objects.....   | 109/453        |
| Collections.....  | 109/453        |
| Queries.....  | 110/453        |
| Demo App.....   | 111/453        |
| Hibernate Tutorial.....   | 111/453        |
| Performance Tips.....   | 111/453        |
| Session.load.....   | 111/453        |
| Session.find and Query.find.....  | 111/453        |
| Session.iterate and Query.iterate.....  | 112/453        |
| FAQ.....  | 112/453        |
| If I'm using BigMemory Go with my app and with Hibernate for second-level caching,<br>should I try to use the CacheManager created by Hibernate for my app's caches?..... | 112/453        |
| Should I use the provider in the Hibernate distribution or in BigMemory Go's Ehcache?...<br>112/453   | 112/453        |
| What is the relationship between the Hibernate and Ehcache projects?.....   | 112/453        |
| Does BigMemory Go support the transactional strategy?.....  | 112/453        |
| Why do certain caches sometimes get automatically cleared by Hibernate?.....  | 112/453        |
| How are Hibernate Entities keyed?.....  | 113/453        |
| Are compound keys supported?.....   | 113/453        |
| I am getting this error message: An item was expired by the cache while it was locked.<br>What is it?.....  | 113/453        |
| <b>Using Coldfusion and BigMemory Go.....</b>   | <b>114/453</b> |
| Introduction.....   | 114/453        |
| Example Integration.....  | 114/453        |
| <b>Using Spring and BigMemory Go.....</b>   | <b>115/453</b> |
| Introduction.....   | 115/453        |
| Spring 3.1.....   | 115/453        |
| @Cacheable.....   | 115/453        |
| @CacheEvict.....  | 115/453        |
| Spring 2.5 - 3.1: Annotations For Spring.....   | 115/453        |
| @Cacheable.....   | 115/453        |
| @TriggersRemove.....  | 115/453        |
| The Annotations for Spring Project.....   | 116/453        |
| <b>JSR107 Support.....</b>  | <b>118/453</b> |
| <b>BigMemory Go FAQ.....</b>  | <b>119/453</b> |
| Getting Started.....  | 119/453        |
| Is BigMemory Go really free for 32GB?.....  | 119/453        |
| Configuration.....  | 119/453        |
| Where is the source code?.....  | 119/453        |

# Terracotta 3.7.4 Documentation

|   |                |
|---|----------------|
| <b>BigMemory Go FAQ</b>   |                |
| Operations.....   | 123/453        |
| How do you get an element without affecting statistics?.....                                    | 123/453        |
| Troubleshooting.....  | 124/453        |
| I have created a new cache and its status is STATUS_UNINITIALISED. How do I initialise it?..... | 124/453        |
| <b>BigMemory Max Tutorial.....</b>  | <b>126/453</b> |
| Download and Install.....   | 126/453        |
| Choose a Sample Template.....   | 127/453        |
| <b>Working with Terracotta License Files.....</b>   | <b>132/453</b> |
| Explicitly Specifying the Location of the License File.....                                     | 132/453        |
| Verifying Products and Features.....  | 132/453        |
| <b>Configuring BigMemory.....</b>   | <b>134/453</b> |
| Configuring Direct Memory Space.....  | 134/453        |
| Configuring Off-Heap.....   | 134/453        |
| Maximum, Minimum, and Default Values.....   | 135/453        |
| Notes for 32-Bit Systems.....   | 135/453        |
| Default Value of Direct Memory Space.....   | 135/453        |
| Optimizing BigMemory.....   | 135/453        |
| General Memory allocation.....  | 136/453        |
| Compressed References.....  | 136/453        |
| Swapiness and Huge Pages.....   | 136/453        |
| Maximum Serialized Size of an Element.....  | 137/453        |
| <b>The Terracotta Server Array.....</b>   | <b>138/453</b> |
| <b>Terracotta Server Arrays Architecture.....</b>   | <b>139/453</b> |
| Introduction.....   | 139/453        |
| Definitions and Functional Characteristics.....   | 139/453        |
| Server Array Configuration Tips.....  | 141/453        |
| Backing Up Persisted Shared Data.....   | 141/453        |
| Client Disconnection.....   | 142/453        |
| Cluster Structure and Behavior.....   | 142/453        |
| Terracotta Cluster in Development.....  | 142/453        |
| Terracotta Cluster With Reliability.....  | 143/453        |
| Terracotta Server Array with High Availability.....   | 144/453        |
| Scaling the Terracotta Server Array.....  | 149/453        |
| About Distributed Garbage Collection.....   | 151/453        |
| Types of DGC.....   | 152/453        |
| Running the Periodic DGC.....   | 152/453        |
| Monitoring and Troubleshooting the DGC.....   | 152/453        |
| <b>Working with Terracotta Configuration Files.....</b>   | <b>153/453</b> |
| Introduction.....   | 153/453        |
| How Terracotta Servers Get Configured.....  | 153/453        |

# Terracotta 3.7.4 Documentation

|   |                |
|---|----------------|
| <b>Working with Terracotta Configuration Files</b>                |                |
| Default Configuration.....  | 153/453        |
| Local XML File (Default).....                                     | 153/453        |
| Local or Remote Configuration File.....                           | 153/453        |
| How Terracotta Clients Get Configured.....                        | 154/453        |
| Local or Remote XML File.....                                     | 154/453        |
| Terracotta Server.....  | 154/453        |
| Configuration in a Development Environment.....                   | 155/453        |
| One-Server Setup in Development.....                              | 155/453        |
| Two-Server Setup in Development.....                              | 155/453        |
| Clients in Development.....                                       | 157/453        |
| Configuration in a Production Environment.....                    | 157/453        |
| Clients in Production.....  | 158/453        |
| Binding Ports to Interfaces.....                                  | 159/453        |
| terracotta.xml (DSO only).....                                    | 160/453        |
| Which Configuration?.....   | 160/453        |
| <br>  |                |
| <b>Configuring Terracotta Clusters For High Availability.....</b> | <b>161/453</b> |
| Introduction.....   | 161/453        |
| Common Causes of Failures in a Cluster.....                       | 161/453        |
| Using BigMemory to Alleviate GC Slowdowns.....                    | 162/453        |
| Basic High-Availability Configuration.....                        | 162/453        |
| High-Availability Features.....                                   | 163/453        |
| HealthChecker.....  | 163/453        |
| Calculating HealthChecker Maximum.....                            | 166/453        |
| Automatic Server Instance Reconnect.....                          | 168/453        |
| Automatic Client Reconnect.....                                   | 168/453        |
| Special Client Connection Properties.....                         | 169/453        |
| Effective Client-Server Reconnection Settings: An Example.....    | 170/453        |
| Testing High-Availability Deployments.....                        | 171/453        |
| High-Availability Network Architecture And Testing.....           | 171/453        |
| Deployment Configuration: Simple (no network redundancy).....     | 171/453        |
| Deployment Configuration: Fully Redundant.....                    | 172/453        |
| Terracotta Cluster Tests.....                                     | 175/453        |
| <br>  |                |
| <b>Cluster Security.....</b>                                      | <b>178/453</b> |
| Introduction.....   | 178/453        |
| Configure SSL-based Security.....                                 | 178/453        |
| Configure Security Using LDAP (via JAAS).....                     | 178/453        |
| User Roles.....   | 179/453        |
| Extending Server Security.....                                    | 179/453        |
| <br>  |                |
| <b>Securing Terracotta Clusters with SSL.....</b>                 | <b>180/453</b> |
| Introduction.....   | 180/453        |
| Overview.....   | 180/453        |
| Security-Related Files.....                                       | 180/453        |
| Setting Up Server Security.....                                   | 182/453        |
| Create the Server Certificate and Add It to the Keystore.....     | 183/453        |

# Terracotta 3.7.4 Documentation

|   |                |
|---|----------------|
| <b>Securing Terracotta Clusters with SSL</b>                          |                |
| Set Up the Server Keychain.....                                       | 184/453        |
| Set Up Authentication.....  | 186/453        |
| Configure Server Security.....  | 187/453        |
| Enabling SSL on Terracotta Clients.....                               | 188/453        |
| Create a Keychain Entry.....  | 188/453        |
| Using a Client Truststore.....  | 189/453        |
| Restricting Clients to Specified Servers (Optional).....              | 189/453        |
| Running a Secured Server.....   | 190/453        |
| Confirm Security Enabled.....   | 190/453        |
| Stopping a Secured Server.....  | 190/453        |
| Troubleshooting.....  | 190/453        |
| <b>Changing Cluster Topology in a Live Cluster.....</b>               | <b>192/453</b> |
| Introduction.....   | 192/453        |
| Adding a New Server.....  | 192/453        |
| Removing an Existing Server.....                                      | 193/453        |
| Editing the Configuration of an Existing Server.....                  | 194/453        |
| <b>Terracotta Configuration Reference.....</b>                        | <b>195/453</b> |
| Introduction.....   | 195/453        |
| Configuration Schema.....   | 195/453        |
| Reference and Sample Configuration Files.....                         | 195/453        |
| Configuration Structure.....  | 195/453        |
| Configuration Variables.....  | 195/453        |
| Overriding tc.properties.....   | 196/453        |
| Setting System Properties in tc-config.....                           | 196/453        |
| Override Priority.....  | 196/453        |
| Failure to Override.....  | 197/453        |
| System Configuration Section.....                                     | 197/453        |
| /tc:tc-config/system/configuration-model.....                         | 197/453        |
| Servers Configuration Section.....                                    | 197/453        |
| /tc:tc-config/servers.....  | 197/453        |
| /tc:tc-config/servers/server.....                                     | 198/453        |
| /tc:tc-config/servers/server/data.....                                | 198/453        |
| /tc:tc-config/servers/server/logs.....                                | 199/453        |
| /tc:tc-config/servers/server/statistics.....                          | 199/453        |
| /tc:tc-config/servers/server/dso-port.....                            | 200/453        |
| /tc:tc-config/servers/server/jmx-port.....                            | 200/453        |
| /tc:tc-config/servers/server/l2-group-port.....                       | 200/453        |
| /tc:tc-config/servers/server/security.....                            | 200/453        |
| /tc:tc-config/servers/server/authentication.....                      | 201/453        |
| /tc:tc-config/servers/server/http-authentication/user-realm-file..... | 201/453        |
| /tc:tc-config/servers/server/dso.....                                 | 202/453        |
| /tc:tc-config/servers/server/dso/client-reconnect-window.....         | 202/453        |
| /tc:tc-config/servers/server/dso/persistence.....                     | 202/453        |
| /tc:tc-config/servers/server/dso/garbage-collection.....              | 202/453        |
| /tc:tc-config/servers/ha.....   | 203/453        |

# Terracotta 3.7.4 Documentation

|  |                |
|--|----------------|
| <b>Terracotta Configuration Reference</b>                                  |                |
| /tc:tc-config/servers/ha	mode.....   | 204/453        |
| /tc:tc-config/servers/ha/networked-active-passive.....                     | 204/453        |
| /tc:tc-config/servers/mirror-groups.....                                   | 204/453        |
| <b>Clients Configuration Section.....</b>                                  | <b>205/453</b> |
| /tc:tc-config/clients/logs.....  | 205/453        |
| /tc:tc-config/clients/dso/fault-count.....                                 | 206/453        |
| /tc:tc-config/clients/dso/debugging.....                                   | 206/453        |
| /tc:tc-config/clients/dso/debugging/runtime-logging/new-object-debug.....  | 206/453        |
| /tc:tc-config/clients/dso/debugging/runtime-output-options.....            | 206/453        |
| /tc:tc-config/clients/dso/debugging/runtime-output-options/caller.....     | 206/453        |
| /tc:tc-config/clients/dso/debugging/runtime-output-options/full-stack..... | 206/453        |
| <b>Terracotta Developer Console.....</b>                                   | <b>207/453</b> |
| Introduction.....  | 207/453        |
| Launching the Terracotta Developer Console.....                            | 207/453        |
| The Console Interface.....   | 208/453        |
| Console Messages.....  | 209/453        |
| Menus.....   | 209/453        |
| Context-Sensitive Help.....  | 209/453        |
| Context Menus.....   | 209/453        |
| Working with Clusters.....   | 209/453        |
| Adding and Removing Clusters.....  | 210/453        |
| Connecting to a cluster.....   | 210/453        |
| Connecting to a Secured Cluster.....                                       | 211/453        |
| Disconnecting from a Cluster.....  | 212/453        |
| Enterprise Ehcache Applications.....                                       | 212/453        |
| Overview Panel.....  | 212/453        |
| Performance Panel.....   | 215/453        |
| Statistics Panel.....  | 216/453        |
| Sizing Panel.....  | 220/453        |
| Editing Cache Configuration.....   | 221/453        |
| Enterprise Ehcache for Hibernate Applications.....                         | 222/453        |
| Hibernate View.....  | 223/453        |
| Second-Level Cache View.....   | 224/453        |
| Clustered Quartz Scheduler Applications.....                               | 229/453        |
| Overview.....  | 229/453        |
| Clustered HTTP Sessions Applications.....                                  | 231/453        |
| Overview.....  | 232/453        |
| Working with Terracotta Server Arrays.....                                 | 235/453        |
| Server Panel.....  | 236/453        |
| Connecting and Disconnecting from a Server.....                            | 238/453        |
| Server Connection Status.....  | 238/453        |
| Working with Clients.....  | 239/453        |
| Client Panel.....  | 239/453        |
| Connecting and Disconnecting Clients.....                                  | 240/453        |
| Monitoring Clusters, Servers, and Clients.....                             | 240/453        |
| Client Flush and Fault Rate Graphs.....                                    | 240/453        |

# Terracotta 3.7.4 Documentation

|   |                |
|---|----------------|
| <b>Terracotta Developer Console</b>                               |                |
| Real-Time Performance Monitoring.....                             | 240/453        |
| Logs and Status Messages.....                                     | 247/453        |
| Operator Events.....  | 248/453        |
| Advanced Monitoring and Diagnostics.....                          | 248/453        |
| Shared Objects.....   | 248/453        |
| Lock Profiler.....  | 252/453        |
| Recording and Viewing Statistics.....                             | 255/453        |
| Cluster Statistics Recorder.....                                  | 255/453        |
| Troubleshooting the Console.....                                  | 256/453        |
| Cannot Connect to Cluster (Console Times Out).....                | 256/453        |
| Failure to Display Certain Metrics Hyperic (Sigar) Exception..... | 256/453        |
| Console Runs Very Slowly.....                                     | 256/453        |
| Console Logs and Configuration File.....                          | 257/453        |
| Backing Up Shared Data.....                                       | 257/453        |
| Update Checker.....   | 257/453        |
| Definitions of Cluster Statistics.....                            | 258/453        |
| cache objects evict request.....                                  | 258/453        |
| cache objects evicted.....  | 258/453        |
| I1 I2 flush.....  | 258/453        |
| I2 faults from disk.....  | 258/453        |
| I2 I1 fault.....  | 258/453        |
| memory (usage).....   | 258/453        |
| vm garbage collector.....   | 258/453        |
| distributed gc (distributed garbage collection, or DGC).....      | 258/453        |
| I2 pending transactions.....                                      | 259/453        |
| stage queue depth.....  | 259/453        |
| server transaction sequencer stats.....                           | 259/453        |
| network activity.....   | 259/453        |
| I2 changes per broadcast.....                                     | 259/453        |
| message monitor.....  | 259/453        |
| I2 broadcast count.....   | 259/453        |
| I2 transaction count.....   | 259/453        |
| I2 broadcast per transaction.....                                 | 259/453        |
| system properties.....  | 260/453        |
| disk activity.....  | 260/453        |
| cpu (usage).....  | 260/453        |
| <b>The Terracotta Operations Console.....</b>                     | <b>261/453</b> |
| <b>JMX Management and Monitoring.....</b>                         | <b>262/453</b> |
| Introduction.....   | 262/453        |
| JMX Overview.....   | 262/453        |
| MBeans.....   | 262/453        |
| JMX Remoting.....   | 263/453        |
| ObjectName naming scheme.....                                     | 263/453        |
| The Management Service.....                                       | 263/453        |
| JConsole Example.....   | 265/453        |

# Terracotta 3.7.4 Documentation

|  |                |
|--|----------------|
| <b>JMX Management and Monitoring</b>                                 |                |
| Hibernate statistics.....  | 265/453        |
| Performance.....   | 266/453        |
| SSL-Secured JMX Monitoring.....                                      | 266/453        |
| Compile the Client.....  | 266/453        |
| Run the Client.....  | 267/453        |
| About the Credentials.....   | 267/453        |
| Troubleshooting.....   | 267/453        |
| <b>Terracotta Tools Catalog.....</b>                                 | <b>271/453</b> |
| Introduction.....  | 271/453        |
| Terracotta Maven Plugin.....   | 271/453        |
| Cluster Thread and State Dumps (debug-tool).....                     | 271/453        |
| TIM Management (tim-get).....  | 272/453        |
| Sessions Configurator (sessions-configurator).....                   | 272/453        |
| Developer Console (dev-console).....                                 | 272/453        |
| Operations Center (ops-center).....                                  | 272/453        |
| Archive Utility (archive-tool).....                                  | 272/453        |
| Database Backup Utility (backup-data).....                           | 273/453        |
| Using the Terracotta Operations Center.....                          | 273/453        |
| Distributed Garbage Collector (run-dgc).....                         | 274/453        |
| Further Reading.....   | 274/453        |
| Start and Stop Server Scripts (start-tc-server, stop-tc-server)..... | 274/453        |
| Stopping an SSL-Secured Server.....                                  | 275/453        |
| Further Reading.....   | 275/453        |
| Version Utility (version).....                                       | 276/453        |
| Server Status (server-stat).....                                     | 276/453        |
| Example.....   | 277/453        |
| Cluster Statistics Recorder (tc-stats).....                          | 277/453        |
| DSO Tools.....   | 277/453        |
| Sample Launcher (samples).....                                       | 277/453        |
| Make Boot Jar Utility (make-boot-jar).....                           | 277/453        |
| Scan Boot Jar Utility (scan-boot-jar).....                           | 278/453        |
| Boot Jar Path Utility (boot-jar-path).....                           | 278/453        |
| DSO Environment Setter (dso-env).....                                | 278/453        |
| Java Wrapper (dso-java).....   | 279/453        |
| <b>Introduction.....</b>   | <b>280/453</b> |
| Intended Audience.....   | 280/453        |
| <b>SPOF Analysis Diagram.....</b>                                    | <b>281/453</b> |
| <b>Failure Scenarios.....</b>  | <b>282/453</b> |
| Client (L1) Failures.....  | 282/453        |
| Loss of Terracotta-Client Java PID.....                              | 282/453        |
| Terracotta-Client Host Reboot.....                                   | 282/453        |
| Terracotta-Client Host Extended Power Outage.....                    | 284/453        |
| L1 Local Disk Full.....  | 284/453        |

# Terracotta 3.7.4 Documentation

|  |                |
|--|----------------|
| <b>Failure Scenarios</b>   |                |
| L1 CPU Pegged.....   | 284/453        |
| L1 Memory Pegged (Constant GC).....  | 285/453        |
| L1-L2 Connectivity Failure.....  | 286/453        |
| L1 NIC Failure - Dual NIC Client Host.....   | 286/453        |
| Primary Switch Failure.....  | 287/453        |
| Primary L2 NIC Failure - Dual NIC Terracotta Server Host.....                                | 288/453        |
| Terracotta Server (L2) Subsystem failure.....  | 289/453        |
| Hot-standby L2 Available - Primary L2 Java PID Exits.....                                    | 289/453        |
| Hot-standby L2 Available - Primary L2 Host Reboot.....                                       | 290/453        |
| Hot-standby L2 Available - Primary Host Unavailable: Extended Power Outage.....              | 290/453        |
| Primary L2 Local Disk Full.....  | 291/453        |
| Primary L2 - CPU Pegged at 100%.....   | 291/453        |
| Primary L2 Memory Pegged - Excessive GC or I/O (Host Reachable but PID<br>Unresponsive)..... | 292/453        |
| Primary L2 Available - Hot-standby L2 PID Unresponsive.....                                  | 293/453        |
| Primary L2 Available - Hot-standby L2 Host Failure.....                                      | 294/453        |
| Primary L2 Available - Hot-standby L2 NIC Failure (Dual NIC Host).....                       | 295/453        |
| Primary L2 Available - Hot-standby L2 "Gray" Issue (CPU High).....                           | 295/453        |
| Primary L2 Available - Hot-standby L2 "Gray" Issue (Memory Pegged).....                      | 296/453        |
| Primary L2 Available - Hot-standby L2 "Grey" Issue (Disk Full).....                          | 297/453        |
| Primary and Hot-standby L2 Failure.....  | 297/453        |
| Other Failures.....  | 298/453        |
| Data Center Failure.....   | 298/453        |
| <b>Recovery Scenarios.....</b>   | <b>299/453</b> |
| <b>A Brief on Terracotta Health Monitoring.....</b>  | <b>301/453</b> |
| Variables and Usage.....   | 301/453        |
| A Word about Reconnect Properties.....   | 302/453        |
| Tolerance Formula and Flowchart.....   | 302/453        |
| Assumptions.....   | 303/453        |
| <b>How to Report an Issue to Terracotta.....</b>   | <b>304/453</b> |
| Artifacts List.....  | 304/453        |
| Delivering Artifacts to Terracotta.....  | 304/453        |
| <b>Developing Applications With the Terracotta Toolkit.....</b>                              | <b>306/453</b> |
| Introduction.....  | 306/453        |
| Installing the Terracotta Toolkit.....   | 306/453        |
| Understanding Versions.....  | 307/453        |
| <b>Working With the Terracotta Toolkit.....</b>  | <b>308/453</b> |
| Initializing the Toolkit.....  | 308/453        |
| Using Toolkit Tools.....   | 308/453        |
| Toolkit Data Structures and Serialization.....   | 308/453        |
| Maps.....  | 309/453        |
| Queues.....  | 309/453        |

# Terracotta 3.7.4 Documentation

|  |                |
|--|----------------|
| <b>Working With the Terracotta Toolkit</b>   |                |
| Cluster Information.....   | 309/453        |
| Locks.....   | 310/453        |
| Clustered Barriers.....  | 311/453        |
| Utilities.....   | 312/453        |
| Destroying Clustered Terracotta Toolkit Objects.....   | 312/453        |
| <br>   |                |
| <b>Terracotta Toolkit Reference.....</b>   | <b>313/453</b> |
| Client Failures.....   | 313/453        |
| Connection Issues.....   | 313/453        |
| Multiple Terracotta Clients in a Single JVM.....   | 313/453        |
| Multiple Clients With a Single Web Application.....  | 313/453        |
| Clients Sharing a Node ID.....   | 313/453        |
| <br>   |                |
| <b>Working with Apache Maven.....</b>  | <b>315/453</b> |
| UNIX/LINUX.....  | 315/453        |
| MICROSOFT WINDOWS.....   | 315/453        |
| Creating Enterprise Edition Clients.....   | 316/453        |
| Using the tc-maven Plugin.....   | 316/453        |
| Working With Terracotta SNAPSHOT Projects.....   | 317/453        |
| Terracotta Repositories.....   | 318/453        |
| <br>   |                |
| <b>Terracotta Clustering Best Practices.....</b>   | <b>319/453</b> |
| Analyze Java Garbage Collection (GC).....  | 319/453        |
| Printing and Analyzing GC Logs.....  | 319/453        |
| Observing GC Statistics With jstat.....  | 319/453        |
| Solutions to Problematic GC.....   | 319/453        |
| Detect Memory Pressure Using the Terracotta Logs.....  | 320/453        |
| Reduce Swapping.....   | 320/453        |
| Keep Disks Local.....  | 321/453        |
| Do Not Interrupt!.....   | 321/453        |
| Diagnose Client Disconnections.....  | 321/453        |
| Bring a Cluster Back Up in Order.....  | 321/453        |
| Manage Sessions in a Cluster.....  | 322/453        |
| <br>   |                |
| <b>BigMemory Max Explained.....</b>  | <b>324/453</b> |
| How BigMemory Improves Performance.....  | 324/453        |
| Using BigMemory With Terracotta Servers.....   | 325/453        |
| <br>   |                |
| <b>Technical FAQ.....</b>  | <b>326/453</b> |
| CONFIGURATION.....   | 326/453        |
| How do I enable persistent mode?.....  | 326/453        |
| How do I configure failover to work properly with two Terracotta servers?.....                       | 326/453        |
| DEVELOPMENT.....   | 326/453        |
| How do I know that my application has started up with a Terracotta client and are sharing data?..... | 327/453        |
| Is there a maximum number of objects that can be held by one Terracotta server instance?.....        | 327/453        |

# Terracotta 3.7.4 Documentation

## Technical FAQ

|  |         |
|--|---------|
| Why is it a bad idea to change shared data in a shutdown hook?.....  | 327/453 |
| What's the best way for my application to listen to Terracotta cluster events such as lost<br>application nodes?.....  | 327/453 |
| How can my application check that the Terracotta process is alive at runtime?.....   | 327/453 |
| What are some ways to externally monitor a cluster?.....   | 328/453 |
| ENVIRONMENT.....   | 328/453 |
| Where is there information on platform compatibility for my version of Terracotta<br>software?.....  | 328/453 |
| Can I run the Terracotta process as a Microsoft Windows service?.....  | 328/453 |
| How do I use my JRE instead of the one shipped with Terracotta?.....   | 328/453 |
| Do you have any advice for running Terracotta software on Ubuntu?.....   | 328/453 |
| Which Garbage Collector should I use with the Terracotta Server (L2) process?.....   | 328/453 |
| INTEROPERABILITY.....  | 329/453 |
| Can I substitute Terracotta for JMS? How do you do messaging in Terracotta clusters?.....  | 329/453 |
| Does Terracotta clustering work with Hibernate?.....   | 329/453 |
| What other technologies does Terracotta software work with?.....   | 329/453 |
| OPERATIONS.....  | 329/453 |
| How do I confirm that my Terracotta servers are up and running correctly?.....   | 329/453 |
| How can I control the logging level for Terracotta servers and clients?.....   | 330/453 |
| Are there ways I can monitor the cluster that don't involve using the Terracotta<br>Developer Console or Operations Center?.....                                 | 330/453 |
| How many Terracotta clients (L1s) can connect to the Terracotta Server Array (L2s) in<br>a cluster?.....   | 330/453 |
| TROUBLESHOOTING.....   | 331/453 |
| After my application interrupted a thread (or threw InterruptedException), why did the<br>Terracotta client die?.....  | 331/453 |
| Why does the cluster seem to be running more slowly?.....  | 331/453 |
| Why do all of my objects disappear when I restart the server?.....   | 331/453 |
| Why are old objects still there when I restart the server?.....  | 331/453 |
| Why is the Terracotta Developer Console or Terracotta Operations Center timing out<br>when it tries to connect to a Terracotta server?.....                      | 331/453 |
| Why does the Developer Console runs very slowly when I'm monitoring the cluster?.....  | 332/453 |
| Why can't certain nodes on my Terracotta cluster see each other on the network?.....   | 332/453 |
| Client and/or server nodes are exiting regularly without reason.....   | 332/453 |
| Why is more than one active server coming up when I configured only one?.....  | 332/453 |
| I have a cluster with more than one stripe (more than one active Terracotta server) but<br>data is distributed very unevenly between the two stripes.....        | 333/453 |
| Why is a crashed Terracotta server instance failing to come up when I restart it?.....   | 333/453 |
| I lost some data after my entire cluster lost power and went down. How can I ensure<br>that all data persists through a failure?.....                            | 333/453 |
| Why does the JVM on my SPARC machines crash regularly?.....  | 333/453 |
| SPECIFIC ERRORS AND WARNINGS.....  | 333/453 |
| Why, after restarting an application server, does the error Client Cannot Reconnect...<br>repeat endlessly until the Terracotta server's database is wiped?..... | 334/453 |
| What does the warning "WARN com.tc.bytes.TCByteBufferFactory - Asking for a<br>large amount of memory..." mean?.....   | 334/453 |
| What is causing regular segmentation faults (segfaults) with clients and/or servers  |         |

# Terracotta 3.7.4 Documentation

|   |             |
|---|-------------|
| <b>Technical FAQ</b>  |             |
| failing?.....   | 334/453     |
| When starting a Terracotta server, why does it throw a DBVersionMismatchException?.....   | 334/453     |
| Why am I getting MethodNotFound and ClassNotFoundException exceptions?.....   | 334/453     |
| I'm getting a Hyperic (Sigar) exception, and the Terracotta Developer Console or<br>Terracotta Operations Center is not showing certain metrics?..... | 335/453     |
| When I start a Terracotta server, why does it fail with a schema error?.....  | 335/453     |
| Why is a newly started passive (backup) Terracotta server failing to join the cluster as it<br>tries to synchronize with the active server?.....      | 335/453     |
| The Terracotta servers crash regularly and I see a ChecksumException.....   | 335/453     |
| Why is java.net.UnknownHostException thrown when I try to run Terracotta sample<br>applications?.....   | 336/453     |
| On a node with plenty of RAM and disk space, why is there a failure with errors stating<br>that a "native thread" cannot be created?.....             | 336/453     |
| Why does the Terracotta server crash regularly with java.io.IOException: File exists?.....  | 336/453     |
| <br><b>Distributed Cache Tutorial</b> .....   | <br>337/453 |
| Download and Install the Terracotta Enterprise Suite.....   | 337/453     |
| Download.....   | 337/453     |
| <br><b>Hibernate Distributed Cache Tutorial</b> .....   | <br>344/453 |
| Download and Install the Terracotta Enterprise Suite.....   | 344/453     |
| Download.....   | 344/453     |
| <br><b>Enterprise Ehcache Installation</b> .....  | <br>350/453 |
| Introduction.....   | 350/453     |
| Step 1: Requirements.....   | 350/453     |
| Step 2: Install the Distributed Cache.....  | 350/453     |
| Step 3: Configure the Distributed Cache.....  | 351/453     |
| Add Terracotta to Specific Caches.....  | 351/453     |
| Edit Incompatible Configuration.....  | 352/453     |
| Step 4: Start the Cluster.....  | 352/453     |
| Step 5: Edit the Terracotta Configuration.....  | 353/453     |
| Procedure:.....   | 354/453     |
| Step 6: Learn More.....   | 355/453     |
| <br><b>Fast Restartability</b> .....  | <br>356/453 |
| Introduction.....   | 356/453     |
| Cache Persistence Implementation.....   | 356/453     |
| Strategy Options.....   | 356/453     |
| Synchronous Writes Options.....   | 357/453     |
| DiskStore Path.....   | 357/453     |
| Configuration Examples.....   | 357/453     |
| Options for Crash Resilience.....   | 358/453     |
| Clustered Caches.....   | 358/453     |
| Temporary Disk Storage.....   | 358/453     |
| In-memory Only Cache.....   | 358/453     |
| Programmatic Configuration Example.....   | 359/453     |

# Terracotta 3.7.4 Documentation

|   |                |
|---|----------------|
| <b>Fast Restartability</b>                                      |                |
| Fast Restart Performance.....                                   | 359/453        |
| Fast Restart Limitations.....                                   | 359/453        |
| Compatibility with Previous Versions.....                       | 359/453        |
| Comparison of Disk Usage Options.....                           | 359/453        |
| Upgrading to Ehcache 2.6.....                                   | 360/453        |
| <b>Enterprise Ehcache API Guide.....</b>                        | <b>361/453</b> |
| Enterprise Ehcache Search API For Clustered Caches.....         | 361/453        |
| Sample Code.....  | 361/453        |
| Dynamic Search.....   | 362/453        |
| Stored Search Indexes.....                                      | 364/453        |
| Best Practices for Optimizing Searches.....                     | 364/453        |
| Enterprise Ehcache Cluster Events.....                          | 365/453        |
| Cluster Topology.....   | 365/453        |
| Cluster Events.....   | 366/453        |
| Events API Example Code.....                                    | 367/453        |
| Bulk-Load API.....  | 367/453        |
| Bulk-Load API Example Code.....                                 | 368/453        |
| Unlocked Reads for Consistent Caches (UnlockedReadsView).....   | 369/453        |
| UnlockedReadsView and Data Freshness.....                       | 370/453        |
| Explicit Locking.....   | 370/453        |
| Configuration Using the Fluent Interface.....                   | 371/453        |
| Write-Behind Queue in Enterprise Ehcache.....                   | 371/453        |
| <b>Enterprise Ehcache Configuration Reference.....</b>          | <b>373/453</b> |
| Offloading Large Caches.....                                    | 373/453        |
| Tuning Concurrency.....   | 374/453        |
| Non-Blocking Disconnected (Nonstop) Cache.....                  | 374/453        |
| Configuring Nonstop.....  | 374/453        |
| Nonstop Timeouts and Behaviors.....                             | 375/453        |
| How Configuration Affects Element Eviction.....                 | 376/453        |
| Strict Consistency, UnlockedReadsView, and Bulk Loading.....    | 377/453        |
| Understanding Performance and Cache Consistency.....            | 378/453        |
| Cache Events in a Terracotta Cluster.....                       | 379/453        |
| Handling Cache Update Events.....                               | 379/453        |
| Configuring Caches for High Availability.....                   | 380/453        |
| Using Rejoin to Automatically Reconnect Terracotta Clients..... | 380/453        |
| Working With Transactional Caches.....                          | 381/453        |
| Strict XA (Support for All JTA Components).....                 | 382/453        |
| XA (Basic JTA Support).....                                     | 384/453        |
| Local Transactions.....   | 384/453        |
| Avoiding XA Commit Failures With Atomic Methods.....            | 385/453        |
| Implementing an Element Comparator.....                         | 385/453        |
| Working With OSGi.....  | 386/453        |

# Terracotta 3.7.4 Documentation

|  |                |
|--|----------------|
| <b>Enterprise Ehcache for Hibernate.....</b>             | <b>388/453</b> |
| Introduction.....  | 388/453        |
| Installing Enterprise Ehcache for Hibernate.....         | 388/453        |
| Step 1: Requirements.....                                | 388/453        |
| Step 2: Install and Update the JAR files.....            | 388/453        |
| Step 3: Prepare Your Application for Caching.....        | 389/453        |
| Step 4: Edit Configuration Files.....                    | 391/453        |
| Step 5: Start Your Application with the Cache.....       | 392/453        |
| Step 6: Edit the Terracotta Configuration.....           | 393/453        |
| Step 7: Learn More.....                                  | 394/453        |
| Testing and Tuning Enterprise Ehcache for Hibernate..... | 395/453        |
| Testing the Cache.....                                   | 395/453        |
| Optimizing the Cache Size.....                           | 396/453        |
| Optimizing for Read-Only Data.....                       | 399/453        |
| Reducing Unnecessary Database Connections.....           | 399/453        |
| Reducing Memory Usage with Batch Processing.....         | 401/453        |
| Other Important Tuning Factors.....                      | 402/453        |
| Enterprise Ehcache for Hibernate Reference.....          | 406/453        |
| Cache Configuration File.....                            | 406/453        |
| Migrating From an Existing Second-Level Cache.....       | 407/453        |
| Cache Concurrency Strategies.....                        | 407/453        |
| Setting Up Transactional Caches.....                     | 409/453        |
| Configuring Multiple Hibernate Applications.....         | 410/453        |
| Finding Cacheable Entities and Collections.....          | 410/453        |
| Cache Regions in the Object Browser.....                 | 410/453        |
| Hibernate Statistics Sampling Rate.....                  | 411/453        |
| Is a Cache Appropriate for Your Use Case?.....           | 411/453        |
| <b>Clustering Quartz Scheduler.....</b>                  | <b>413/453</b> |
| Step 1: Requirements.....                                | 413/453        |
| Step 2: Install Quartz Scheduler.....                    | 413/453        |
| Step 3: Configure Quartz Scheduler.....                  | 414/453        |
| Add Terracotta Configuration.....                        | 414/453        |
| Scheduler Instance Name.....                             | 414/453        |
| Step 4: Start the Cluster.....                           | 415/453        |
| Step 5: Edit the Terracotta Configuration.....           | 416/453        |
| Procedure:.....  | 416/453        |
| Step 6: Learn More.....                                  | 418/453        |
| <b>Quartz Scheduler Where (Locality API).....</b>        | <b>419/453</b> |
| Introduction.....  | 419/453        |
| Installing Quartz Scheduler Where.....                   | 419/453        |
| Configuring Quartz Scheduler Where.....                  | 419/453        |
| Understanding Generated Node IDs.....                    | 420/453        |
| Using SystemPropertyInstanceIdGenerator.....             | 420/453        |
| Available Constraints.....                               | 421/453        |
| Quartz Scheduler Where Code Sample.....                  | 421/453        |
| CPU-Based Constraints.....                               | 423/453        |

# Terracotta 3.7.4 Documentation

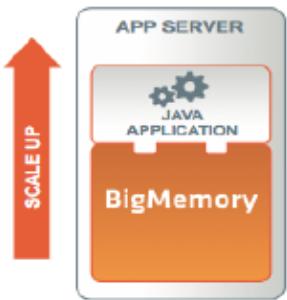
|  |                |
|--|----------------|
| <b>Quartz Scheduler Where (Locality API)</b>               |                |
| Failure Scenarios.....                                     | 424/453        |
| Locality With the Standard Quartz Scheduler API.....       | 424/453        |
| <br>   |                |
| <b>Quartz Scheduler Reference.....</b>                     | <b>425/453</b> |
| Execution of Jobs.....                                     | 425/453        |
| Working With JobDataMaps.....                              | 425/453        |
| Updating a JobDataMap.....                                 | 425/453        |
| Best Practices for Storing Objects in a JobDataMap.....    | 425/453        |
| Cluster Data Safety.....                                   | 425/453        |
| Effective Scaling Strategies.....                          | 426/453        |
| <br>   |                |
| <b>Web Sessions Installation.....</b>                      | <b>427/453</b> |
| Step 1: Requirements.....                                  | 427/453        |
| Step 2: Install the Terracotta Sessions JAR.....           | 427/453        |
| Step 3: Configure Web-Session Clustering.....              | 428/453        |
| Jetty, WebLogic, and WebSphere.....                        | 428/453        |
| Tomcat and JBoss AS 6.0 or Earlier.....                    | 429/453        |
| JBoss AS 7.0 or 7.1.....                                   | 430/453        |
| Step 4: Start the Cluster.....                             | 431/453        |
| Step 5: Edit the Terracotta Configuration.....             | 432/453        |
| Procedure:.....  | 432/453        |
| Step 6: Learn More.....                                    | 434/453        |
| <br>   |                |
| <b>Web Sessions Reference.....</b>                         | <b>435/453</b> |
| Architecture of a Terracotta Cluster.....                  | 435/453        |
| Additional Configuration Options.....                      | 436/453        |
| Session Locking.....                                       | 436/453        |
| Synchronous Writes.....                                    | 436/453        |
| Troubleshooting.....                                       | 437/453        |
| Sessions Time Out Unexpectedly.....                        | 437/453        |
| Changes Not Replicated.....                                | 437/453        |
| Tomcat 5.5 Messages Appear With Tomcat 6 Installation..... | 437/453        |
| Deadlocks When Session Locking Is Enabled.....             | 437/453        |
| Events Not Received on Node.....                           | 437/453        |
| <br>   |                |
| <b>Security Overview.....</b>                              | <b>438/453</b> |
| Introduction.....  | 438/453        |
| For Details.....   | 438/453        |
| <br>   |                |
| <b>About Terracotta DSO Installations.....</b>             | <b>439/453</b> |
| Standard Versus DSO Installations.....                     | 439/453        |
| Overview of Installation.....                              | 440/453        |
| <br>   |                |
| <b>Performing a DSO Installation.....</b>                  | <b>441/453</b> |
| Introduction.....  | 441/453        |
| Prerequisites.....   | 441/453        |
| Enterprise Ehcache Users.....                              | 441/453        |

# Terracotta 3.7.4 Documentation

## Performing a DSO Installation

|  |         |
|--|---------|
| Quartz Scheduler Users.....                                    | 442/453 |
| Step 1: Configure the Terracotta Platform.....                 | 442/453 |
| TIMs for Clustering Enterprise Ehcache.....                    | 442/453 |
| TIMs for Clustering Quartz Scheduler.....                      | 443/453 |
| TIMs for Integrating an Application Server.....                | 443/453 |
| Clustering a Web Application with Terracotta Web Sessions..... | 445/453 |
| Step 2: Configure Terracotta Products.....                     | 446/453 |
| Enterprise Ehcache Configuration.....                          | 446/453 |
| Enterprise Ehcache for Hibernate Configuration.....            | 447/453 |
| Quartz Scheduler Configuration.....                            | 449/453 |
| Web Sessions Configuration.....                                | 449/453 |
| Step 3: Install the TIMs.....                                  | 449/453 |
| Unix/linux.....  | 450/453 |
| Location of TIMs.....  | 451/453 |
| Step 4: Start the Cluster.....                                 | 451/453 |
| Quartz Scheduler Where DSO Installation.....                   | 452/453 |

# Welcome to BigMemory Go



**BigMemory Go lets you put ALL your data in memory,  
limited only by how much RAM you have  
in your machine.**

With BigMemory Go, you get:

- Unlimited in-memory data management on one Java Virtual Machine (JVM)
- 32GB free
- Advanced monitoring, search, and management for in-memory data

To get started, [download BigMemory Go](#). Then check out the handy [Get Started guide](#) for instructions on installing your license key, configuring BigMemory, and connecting your apps to your new BigMemory data stores.

If you would like to purchase a license for more than 32GB of BigMemory, with professional support and maintenance, [contact us](#). If you're looking to run BigMemory across distributed servers, check out [BigMemory Max](#).

More questions? Post them to the [BigMemory Go support forum](#).

# Getting Started With BigMemory Go

## Installing BigMemory Go

Installing BigMemory Go is as easy as downloading the kit and ensuring that the correct files are on your application's classpath. The only platform requirement is using JDK 1.6 or higher.

1. If you do not have a BigMemory Go kit, download it from [here](#).

The kit is packaged as a tar.gz file. Unpack it on the command line or with the appropriate decompression application.

2. The following JARs are found in the kit's lib directory and must be added to your application's classpath:

- ◆ bigmemory-<version>.jar – This is the main JAR to enable BigMemory.
- ◆ ehcache-ee-<version>.jar – This file contains the API to BigMemory Go.
- ◆ slf4j-api-<version>.jar – This file is the bridge, or logging facade, to the BigMemory Go logging framework.
- ◆ slf4j-jdk14-<version>.jar – This is a binding JAR for the provided SLF4J logging framework, java.util.logging. Binding JARs for other frameworks are available from the [SLF4J website](#).

3. Save the BigMemory Go license-key file to the BigMemory Go home directory. This file, called terracotta-license.key, was attached to an email you received after registering for the BigMemory Go download.

Alternatively, you can add the license-key file to your application's classpath, or specify it with the following Java system property:

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

4. BigMemory Go uses Ehcache as its user-facing interface. To configure BigMemory Go, create or update an Ehcache configuration file to specify how much off-heap in-memory storage you want to use. You may also configure BigMemory to write data to a local disk store for fast restart. For example:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
  name="myBigMemoryGoConfig">

  <!-- Tell BigMemory where to write its data to disk. -->
  <diskStore path="/path/to/my/disk/store/directory"/>

  <!-- set 'maxBytesLocalOffHeap' to the amount of off-heap memory you
      want to use. This memory is invisible to the Java garbage collector,
      providing gigabytes to terabytes of in-memory data without garbage
      collection pauses. -->
  <cache name="myBigMemoryGoStore"
    maxBytesLocalHeap="512M"
    maxBytesLocalOffHeap="32G">
    <!-- Tell BigMemory to use the 'localRestartable' persistence
        strategy for fast restart (optional). -->
    <persistence strategy="localRestartable"/>
  </cache>
</ehcache>
```

## Installing BigMemory Go

Set `maxBytesLocalOffHeap` to the amount of off-heap storage you want to use. Depending on your data and how much physical RAM you have available, you can use just a few gigabytes to multiple terabytes of off-heap memory in a single JVM for ultra-fast access with no garbage collection pauses—and you get up to 32GB for free.

Name this configuration file `ehcache.xml` and place it in the top-level of your classpath.

For more information on configuration options, refer to [the configuration documentation](#) and to the reference `ehcache.xml` configuration file in the `config-samples` directory of the BigMemory Go kit.

5. Use the `-XX:MaxDirectMemorySize` Java option to allocate enough direct memory in the JVM to accomodate the off-heap storage specified in your configuration, plus at least 250MB to allow for other direct memory usage that might occur in your application. For example:

```
-XX:MaxDirectMemorySize=33G
```

Set `MaxDirectMemorySize` to the amount of BigMemory you have. For more information about this step, refer to [Allocating Direct Memory in the JVM](#).

Also, allocate at least enough heap using the `-Xmx` Java option to accomodate the on-heap storage specified in your configuration, plus enough extra heap to run the rest of your application. For example:

```
-Xmx1g
```

6. Look through the [code samples in the documentation](#) and in the download kit for examples of how to employ the various features and capabilities of BigMemory Go.

## Configuring BigMemory Go

For a general overview to configuring BigMemory Go, see this [introductory page](#). Specific configuration topics are introduced below.

### Automatic Resource Control

Automatic Resource Control (ARC) gives you fine-grained controls for tuning performance and enabling trade-offs between throughput, latency and data access. Independently adjustable configuration parameters include differentiated tier-based sizing and pinning hot or eternal data in the most effective tier.

### Dynamically Sizing Stores

Tuning often involves sizing stores appropriately. There are a number of ways to size the different BigMemory Go data tiers using simple configuration sizing attributes. The [sizing page](#) explains how to tune tier sizing by configuring dynamic allocation of memory and automatic balancing.

### Pinning Data

One of the most important aspects of running an in-memory data store involves managing the life of the data in each BigMemory Go tier. See the [data-life page](#) for more information on the pinning, expiration, and eviction of data.

## Fast Restartability

BigMemory Go has full fault tolerance, allowing for continuous access to in-memory data after a planned or unplanned shutdown, with the option to store a fully consistent record of the in-memory data on the local disk at all times. [The fast-restart page](#) covers data persistence, fast restartability, and using the local disk as a storage tier for in-memory data (both heap and off-heap stores).

## Using the BigMemory Go API

BigMemory Go provides a full-featured API. See the [code-samples page](#) for a beginner's view of using the API. Selected advanced API features are introduced below.

### Search

Search billions of entries—gigabytes, even terabytes of data—with results returned in less than a second. Data is indexed without significant overhead, and features like "GroupBy" are included.

[The Search API](#) allows you to execute arbitrarily complex queries against data with pre-built indexes. The development of alternative indexes on values provides the ability for data to be looked up based on multiple criteria instead of just keys.

### Transactional Caching

Transactional modes are a powerful extension for performing atomic operations on data stores, keeping your data in sync with your database.

[The transactions page](#) covers the background and configuration information for BigMemory Go transactional modes. [Explicit Locking](#) is another API that can be used as a custom alternative to XA Transactions or Local transactions.

## Administration and Monitoring

The [Terracotta Management Console](#) (TMC) is a web-based monitoring and administration application for tuning cache usage, detecting errors, and providing an easy-to-use access point to integrate with production management systems.

As an alternative to the TMC, standard [JMX-based administration and monitoring](#) is available.

For logging, BigMemory Go uses the flexible [SLF4J logging framework](#).

## Learn More About How BigMemory Go Works

- [Code Samples](#) to learn what you can do with BigMemory Go.
- [Configuration Overview](#) to learn more about how BigMemory Go is configured.
- [BigMemory Go Architecture](#) to learn more about how BigMemory Go works.

# Code Samples

## Introduction

The following code samples illustrate various features of BigMemory Go. They are also available in the BigMemory Go kit in the /code-samples directory.

| Title   | Description   |
|---|---|
| <a href="#">example01-config-file</a>         | BigMemory may be configured declaratively, using an XML configuration file, or programmatically via the fluent configuration API. This sample shows how to configure a basic instance of BigMemory Go declaratively with the XML configuration file.  |
| <a href="#">example02-config-programmatic</a> | Configure a basic instance of BigMemory Go programmatically with the fluent configuration API.  |
| <a href="#">example03-crud</a>                | Basic create, retrieve, update and delete (CRUD) operations available in BigMemory Go.  |
| <a href="#">example04-search</a>              | Basic in-memory search features of BigMemory Go.  |
| <a href="#">example05-arc</a>                 | Automatic Resource Control (ARC) is a powerful capability of BigMemory Go that gives users the ability to control how much data is stored in heap memory and off-heap memory. This sample shows the basic configuration options for data tier sizing using ARC.   |
| <a href="#">example06-cache</a>               | BigMemory Go is a powerful in-memory data management solution. Among its many applications, BigMemory Go may be used as a cache to speed up access to data from slow or expensive databases and other remote data sources. This example shows how to enable and configure the caching features available in BigMemory Go. |

To run the code samples with Maven, you will need to add the Terracotta Maven repositories to your Maven settings.xml file. Add the following repository information to your settings.xml file:

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

For further information, refer to [Working with Apache Maven](#).

## Example 1: Declarative Configuration via XML

To configure BigMemory declaratively with an XML file, create a CacheManager instance, passing the a file name or an URL object to the constructor.

The following example shows how to create a CacheManager with an URL of the XML file in the classpath at /xml/ehcache.xml.

```
CacheManager manager = CacheManager.newInstance(
    getClass().getResource("/xml/ehcache.xml"));
```

## Example 1: Declarative Configuration via XML

```
try {
    Cache bigMemory = manager.getCache("BigMemory");
    // now do stuff with it...

} finally {
    if (manager != null) manager.shutdown();
}
```

Here are the contents of the XML configuration file used by this sample:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
          name="config">
<cache name="BigMemory"
      maxBytesLocalHeap="512M"
      maxBytesLocalOffHeap="32G"
      copyOnRead="true"
      statistics="true"
      eternal="true">
</cache>
</ehcache>
```

The configuration element `maxBytesLocalOffHeap` lets you set how much off-heap memory to use. BigMemory's unique off-heap memory storage lets you use all of the memory available on a server in a single JVM—from gigabytes to multiple terabytes—without causing garbage collection pauses.

## Example 2: Programmatic Configuration

To configure BigMemory Go programmatically, use the [Ehcache fluent configuration API](#).

```
Configuration managerConfiguration = new Configuration()
    .name("bigmemory-config")
    .cache(new CacheConfiguration()
        .name("BigMemory")
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(1, MemoryUnit.GIGABYTES)
        .copyOnRead(true)
        .statistics(true)
        .eternal(true)
    );
CacheManager manager = CacheManager.create(managerConfiguration);
try {
    Cache bigMemory = manager.getCache("BigMemory");
    // now do stuff with it...

} finally {
    if (manager != null) manager.shutdown();
}
```

## Example 3: Create, Read, Update and Delete (CRUD)

The CRUD sample demonstrates basic create, read, update and delete operations.

First, we create a BigMemory data store configured to use 512 MB of heap memory and 32 GB of off-heap memory:

### Example 3: Create, Read, Update and Delete (CRUD)

```
Configuration managerConfiguration = new Configuration();
managerConfiguration.updateCheck(true)
    .monitoring(Configuration.Monitoring.AUTO_DETECT)
    .name("config")
    .cache(new CacheConfiguration()
        .name("BigMemory-Crud")
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(32, MemoryUnit.GIGABYTES)
    );
CacheManager manager = CacheManager.create(managerConfiguration);
Cache bigMemory = manager.getCache("BigMemory-Crud");
```

This instructs the Automatic Resource Control (ARC) capability of BigMemory to keep a maximum of 512 MB of its data in heap for nanosecond to microsecond access. In this example, as the 512 MB of heap memory fills up, ARC will automatically move data to the 32 GB off-heap store where it is available at microsecond speed. This configuration keeps heap sizes small to avoid garbage collection pauses and tuning, but still uses large amounts of in-process memory for ultra-fast access to data.

**Important:** *BigMemory Go is capable of addressing gigabytes to terabytes of in-memory data in a single JVM and it's free to use up to 32 GB. However, to avoid swapping take care not to configure BigMemory Go to use more memory than is physically available on your hardware.*

Now that we have a BigMemory instance configured and available, we can start creating data in it:

```
final Person timDoe = new Person("Tim Doe", 35, Person.Gender.MALE,
    "eck street", "San Mateo", "CA");
bigMemory.put(new Element("key-1", timDoe));
```

Then, we can read data from it:

```
final Element element = bigMemory.get("key-1");
System.out.println("The value for key-1 is " + element.getObjectValue());
```

And update it:

```
final Person pamelaJones = new Person("Pamela Jones", 23, Person.Gender.FEMALE,
    "berry st", "Parsippany", "LA");
bigMemory.put(new Element("key-1", pamelaJones));
final Element updated = bigMemory.get("key-1");
System.out.println("The value for key-1 is now " + updated.getObjectValue() +
    ". key-1 has been updated.");
```

And delete it:

```
bigMemory.remove("key-1");
System.out.println("Try to retrieve key-1.");
final Element removed = bigMemory.get("key-1");
System.out.println("Value for key-1 is " + removed +
    ". Key-1 has been deleted.");
```

You can also create or update multiple entries at once:

```
Collection<Element> elements = new ArrayList<Element>();
elements.add(new Element("1", new Person("Jane Doe", 35,
    Person.Gender.FEMALE, "eck street", "San Mateo", "CA")));
elements.add(new Element("2", new Person("Marie Antoinette", 23,
```

## Example 4: Search

```
Person.Gender.FEMALE, "berry st", "Parsippany", "LA")));
elements.add(new Element("3", new Person("John Smith", 25,
    Person.Gender.MALE, "big wig", "Beverly Hills", "NJ")));
elements.add(new Element("4", new Person("Paul Dupont", 25,
    Person.Gender.MALE, "big wig", "Beverly Hills", "NJ")));
elements.add(new Element("5", new Person("Juliet Capulet", 25,
    Person.Gender.FEMALE, "big wig", "Beverly Hills", "NJ")));

bigMemory.putAll(elements);
```

And read multiple entries at once:

```
final Map<Object, Element> elementsMap = bigMemory.getAll(
    Arrays.asList("1", "2", "3"));
```

And delete multiple entries at once:

```
bigMemory.removeAll(Arrays.asList("1", "2", "3"));
```

And delete everything at once:

```
bigMemory.removeAll();
```

## Example 4: Search

BigMemory Go comes with powerful in-memory search capabilities. This sample shows how to perform basic search operations on your in-memory data.

First, create an instance of BigMemory Go with searchable attributes:

```
Configuration managerConfig = new Configuration()
    .cache(new CacheConfiguration().name("MySearchableDataStore")
        .eternal(true)
        .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
        .maxBytesLocalOffHeap(32, MemoryUnit.GIGABYTES)
        .searchable(new Searchable()
            .searchAttribute(new SearchAttribute().name("age"))
            .searchAttribute(new SearchAttribute().name("gender")
                .expression("value.getGender()"))
            .searchAttribute(new SearchAttribute().name("state")
                .expression("value.getAddress().getState()"))
            .searchAttribute(new SearchAttribute().name("name")
                .className(NameAttributeExtractor.class.getName())))
        )
    );
CacheManager manager = CacheManager.create(managerConfig);
Ehcache bigMemory = manager.getEhcace("MySearchableDataStore");
```

Now, let's put a bunch of stuff into it:

```
bigMemory.put(new Element(1, new Person("Jane Doe", 35, Gender.FEMALE,
    "eck street", "San Mateo", "CA")));
bigMemory.put(new Element(2, new Person("Marie Antoinette", 23, Gender.FEMALE,
    "berry st", "Parsippany", "LA")));
bigMemory.put(new Element(3, new Person("John Smith", 25, Gender.MALE,
    "big wig", "Beverly Hills", "NJ")));
```

## Example 4: Search

```
bigMemory.put(new Element(4, new Person("Paul Dupont", 45, Gender.MALE,
    "cool agent", "Madison", "WI")));
bigMemory.put(new Element(5, new Person("Juliet Capulet", 30, Gender.FEMALE,
    "dah man", "Bangladesh", "MN")));
for (int i = 6; i < 1000; i++) {
    bigMemory.put(new Element(i, new Person("Juliet Capulet" + i, 30,
        Person.Gender.MALE, "dah man", "Bangladesh", "NJ")));
}
```

Next, create some search attributes and construct a query:

```
Attribute<Integer> age = bigMemory.getSearchAttribute("age");
Attribute<Gender> gender = bigMemory.getSearchAttribute("gender");
Attribute<String> name = bigMemory.getSearchAttribute("name");
Attribute<String> state = bigMemory.getSearchAttribute("state");

Query query = bm.createQuery();
query.includeKeys();
query.includeValues();
query.addCriteria(name.ilike("Jul*").and(gender.eq(Gender.FEMALE)))
    .addOrderBy(age, Direction.ASCENDING).maxResults(10);
```

Then, execute the query and look at the results:

```
Results results = query.execute();
System.out.println(" Size: " + results.size());
System.out.println("----Results----\n");
for (Result result : results.all()) {
    System.out.println("Got: Key[" + result.getKey()
        + "] Value class [" + result.getValue().getClass()
        + "] Value [" + result.getValue() + "]");
}
```

We can also use Aggregators to perform computations across query results. Here's an example that computes the average age of all people in the data set:

```
Query averageAgeQuery = bigMemory.createQuery();
averageAgeQuery.includeAggregator(Aggregators.average(age));
System.out.println("Average age: "
    + averageAgeQuery.execute().all().iterator().next()
    .getAggregatorResults());
```

We can also restrict the calculation to a subset based on search attributes. Here's an example that computes the average age of all people in the data set between the ages of 30 and 40:

```
Query agesBetween = bigMemory.createQuery();
agesBetween.addCriteria(age.between(30, 40));
agesBetween.includeAggregator(Aggregators.average(age));
System.out.println("Average age between 30 and 40: "
    + agesBetween.execute().all().iterator().next()
    .getAggregatorResults());
```

Using Aggregators, we can also find number of entries that match our search criteria. Here's an example that finds the number of people in the data set who live in New Jersey:

```
Query newJerseyCountQuery = bigMemory.createQuery().addCriteria(
    state.eq("NJ"));
newJerseyCountQuery.includeAggregator(Aggregators.count());
```

## Example 5: Automatic Resource Control (ARC)

```
System.out.println("Count of people from NJ: "
    + newJerseyCountQuery.execute().all().iterator().next()
    .getAggregatorResults());
```

## Example 5: Automatic Resource Control (ARC)

Automatic Resource Control (ARC) is a powerful capability of BigMemory Go that gives users the ability to control how much data is stored in heap memory and off-heap memory.

The following XML configuration instructs ARC to allocate a maximum of 512 M of heap memory and 32 G of off-heap memory. In this example, when 512 M of heap memory is used, ARC will automatically move data into off-heap memory up to a maximum of 32 G. The amount of off-heap memory you can use is limited only by the amount of physical RAM you have available.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd"
  updateCheck="false" monitoring="autodetect"
  dynamicConfig="true"
  name="MyManager" maxBytesLocalHeap="512M"
  maxBytesLocalOffHeap="32G">

  <defaultCache>
  </defaultCache>

  <cache name="BigMemory1">
  </cache>

  <cache name="BigMemory2">
  </cache>

</ehcache>
```

It's also possible to allocate resources on a per-data set basis. Here's an example of allocating 8 G of off-heap memory to the BigMemory1 data set and 24 G of off-heap memory to the BigMemory2 data set:

```
<ehcache xmlns
  ...
  name="MyManager"
  maxBytesLocalHeap="512M"
  maxBytesLocalOffHeap="32G"
  maxBytesLocalDisk="128G">

  <cache name="BigMemory1"
    maxBytesLocalOffHeap="8G">
  </cache>

  <cache name="BigMemory2"
    maxBytesLocalOffHeap="24G">
  </cache>

</ehcache>
```

## Example 6: Using BigMemory As a Cache

BigMemory Go is a powerful in-memory data management solution. Among its many applications, BigMemory Go may be used as a cache to speed up access to data from slow or expensive databases and other

## Example 6: Using BigMemory As a Cache

remote data sources. This example shows how to enable and configure the caching features available in BigMemory Go.

The following programmatic configuration snippet shows how to set time-to-live (TTL) and time-to-idle (TTI) policies on a data set:

```
Configuration managerConfiguration = new Configuration();
managerConfiguration.updateCheck(true)
    .monitoring(Configuration.Monitoring.AUTO_DETECT)
    .name("cacheManagerCompleteExample")
    .addCache(
        new CacheConfiguration()
            .name("sample-cache")
            .maxBytesLocalHeap(512, MemoryUnit.MEGABYTES)
            .maxBytesLocalOffHeap(1, MemoryUnit.GIGABYTES)
            .timeToLiveSeconds(4)
            .timeToIdleSeconds(2)

    );
CacheManager manager = CacheManager.create(managerConfiguration);
```

The `timeToLiveSeconds` directive sets the maximum age of an element in the data set. Elements older than the maximum TTL will not be returned from the data store. This is useful when BigMemory is used as a cache of external data and you want to ensure the freshness of the cache.

The `timeToIdleSeconds` directive sets the maximum time since last access of an element. Elements that have been idle longer than the maximum TTI will not be returned from the data store. This is useful when BigMemory is being used as a cache of external data and you want to bias the eviction algorithm towards removing idle entries.

If neither TTL nor TTI are set (or set to zero), data will stay in BigMemory until it is explicitly removed.

# Configuration Overview

## Introduction

BigMemory Go supports declarative configuration via an XML configuration file, as well as programmatic configuration via class-constructor APIs. Choosing one approach over the other can be a matter of preference or a requirement, such as when an application requires a certain runtime context to determine appropriate configuration settings.

If your project permits the separation of configuration from runtime use, there are advantages to the declarative approach:

- Cache configuration can be changed more easily at deployment time.
- Configuration can be centrally organized for greater visibility.
- Configuration lifecycle can be separated from application-code lifecycle.
- Configuration errors are checked at startup rather than causing an unexpected runtime error.
- If the configuration file is not provided, a default configuration is always loaded at runtime.

This documentation focuses on XML declarative configuration. Programmatic configuration is explored in certain examples and is documented in [Javadocs](#).

## XML Configuration

BigMemory Go uses Ehcache as its user-facing interface and is configured using the Ehcache configuration system. By default, Ehcache looks for an ASCII or UTF8 encoded XML configuration file called `ehcache.xml` at the top level of the Java classpath. You may specify alternate paths and filenames for the XML configuration file by using [the various CacheManager constructors](#).

To avoid resource conflicts, one XML configuration is required for each CacheManager that is created. For example, directory paths and listener ports require unique values. BigMemory Go will attempt to resolve conflicts, and, if one is found, it will emit a warning reminding the user to use separate configurations for multiple CacheManagers.

The sample `ehcache.xml` is included in the BigMemory Go distribution. It contains full commentary on how to configure each element. This file can also be downloaded from <http://ehcache.org/ehcache.xml>.

## Dynamically Changing Cache Configuration

While most of the BigMemory Go configuration is not changeable after startup, certain cache configuration parameters can be modified dynamically at runtime. These include the following:

- Expiration settings
  - ◆ `timeToLive` – The maximum number of seconds an element can exist in the cache regardless of access. 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).
  - ◆ `timeToIdle` – 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

## Dynamically Changing Cache Configuration

- cache. The default value is 0, which means no TTI eviction takes place (infinite lifetime).
- Local sizing attributes
  - ◆ maxEntriesLocalHeap
  - ◆ maxBytesLocalHeap
  - ◆ maxEntriesLocalDisk
  - ◆ maxBytesLocalDisk.
- memory-store eviction policy
- CacheEventListeners can be added and removed dynamically

Note that the `eternal` attribute, when set to "true", overrides `timeToLive` and `timeToIdle` so that no expiration can take place.

This example shows how to dynamically modify the cache configuration of a running cache:

```
Cache cache = manager.getCache("sampleCache");
CacheConfiguration config = cache.getCacheConfiguration();
config.setTimeToIdleSeconds(60);
config.setTimeToLiveSeconds(120);
config.setMaxEntriesLocalHeap(10000);
config.setMaxEntriesLocalDisk(1000000);
```

Dynamic cache configurations can also be disabled to prevent future changes:

```
Cache cache = manager.getCache("sampleCache");
cache.disableDynamicFeatures();
```

In `ehcache.xml`, you can disable dynamic configuration by setting the `<ehcache>` element's `dynamicConfig` attribute to "false". For a complete list of configuration parameters that can be changed dynamically, see this [introduction to Ehcache configuration](#).

## Passing Copies Instead of References

By default, a `get()` operation on store data returns a reference to that data, and any changes to that data are immediately reflected in the memory store. In cases where an application requires a *copy* of data rather than a reference to it, you can configure the store to return a copy. This allows you to change a copy of the data without affecting the original data in the memory store.

This is configured using the `copyOnRead` and `copyOnWrite` attributes of the `<cache>` and `<defaultCache>` elements in your configuration, or programmatically as follows:

```
CacheConfiguration config = new CacheConfiguration("copyCache", 1000)
    .copyOnRead(true).copyOnWrite(true);
Cache copyCache = new Cache(config);
```

The default configuration is "false" for both options.

To copy elements on `put()`-like and/or `get()`-like operations, a copy strategy is used. The default implementation uses serialization to copy elements. You can provide your own implementation of `net.sf.ehcache.store.compound.CopyStrategy` using the `<copyStrategy>` element:

```
<cache name="copyCache"
    maxEntriesLocalHeap="10"
```

## Passing Copies Instead of References

```
eternal="false"
timeToIdleSeconds="5"
timeToLiveSeconds="10"
copyOnRead="true"
copyOnWrite="true">
<copyStrategy class="com.company.ehcache.MyCopyStrategy"/>
</cache>
```

A single instance of your CopyStrategy is used per cache. Therefore, in your implementation of CopyStrategy.copy(T), T has to be thread-safe.

A copy strategy can be added programmatically in the following way:

```
CacheConfiguration cacheConfiguration = new CacheConfiguration("copyCache", 10);

CopyStrategyConfiguration copyStrategyConfiguration = new CopyStrategyConfiguration();
copyStrategyConfiguration.setClass("com.company.ehcache.MyCopyStrategy");

cacheConfiguration.addCopyStrategy(copyStrategyConfiguration);
```

## Special System Properties

### **net.sf.ehcache.disabled**

Setting this system property to true (using `java -Dnet.sf.ehcache.disabled=true` in the Java command line) disables caching in ehcache. If disabled, no elements can be added to a cache (puts are silently discarded).

### **net.sf.ehcache.use.classic.lru**

When LRU is selected as the eviction policy, set this system property to true (using `java -Dnet.sf.ehcache.use.classic.lru=true` in the Java command line) to use the older LruMemoryStore implementation. This is provided for ease of migration.

### **ehcache.xsd**

Ehcache configuration files must be comply with the Ehcache XML schema, `ehcache.xsd`, which can be downloaded from <http://ehcache.org/ehcache.xsd>.

Each BigMemory Go distribution also contains a copy of `ehcache.xsd`.

### **ehcache-failsafe.xml**

If the CacheManager default constructor or factory method is called, Ehcache looks for a file called `ehcache.xml` in the top level of the classpath. Failing that it looks for `ehcache-failsafe.xml` in the classpath. `ehcache-failsafe.xml` is packaged in the Ehcache JAR and should always be found.

`ehcache-failsafe.xml` provides an extremely simple default configuration to enable users to get started before they create their own `ehcache.xml`.

## ehcache-failsafe.xml

If it used Ehcache will emit a warning, reminding the user to set up a proper configuration. The meaning of the elements and attributes are explained in the section on `ehcache.xml`.

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>
  <defaultCache
    maxEntriesLocalHeap="10000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="120"
    maxEntriesLocalDisk="10000000"
    diskExpiryThreadIntervalSeconds="120"
    memoryStoreEvictionPolicy="LRU">
    <persistence strategy="localTempSwap"/>
  </defaultCache>
</ehcache>
```

## About Default Cache

The `defaultCache` configuration is applied to any cache that is *not* explicitly configured. The `defaultCache` appears in `ehcache-failsafe.xml` by default, and can also be added to any BigMemory Go configuration file.

While the `defaultCache` configuration is not required, an error is generated if caches are created by name (programmatically) with no `defaultCache` loaded.

## More Information on Configuration Topics

| Topic                             | Description   |
|-----------------------------------|---|
| Sizing Caches                     | Tuning Ehcache often involves sizing cached data appropriately. Ehcache provides a number of ways to size the different data tiers using simple cache-configuration sizing attributes. This page explains simplified tuning of cache size by configuring dynamic allocation of memory and automatic load balancing.   |
| Expiration, Pinning, and Eviction | The architecture of an Ehcache node can include a number of tiers that store data. One of the most important aspects of managing cached data involves managing the life of the data in each tier. This page covers managing data life in Ehcache and the Terracotta Server Array, including the pinning features of Automatic Resource Control (ARC).               |
| Fast Restartability               | This page covers persistence, fast restartability, and using the local disk as a storage tier. The Fast Restart feature provides enterprise-ready crash resilience, which can serve as a fast recovery system after failures, a hot mirror of the data set on the disk at the application node, and an operational store with in-memory speed for reads and writes. |
| Code Samples                      | Examples of working with key classes and methods such as <code>CacheManager</code> and <code>Cache</code> , loading configuration, and getting statistics.  |

# Storage Tiers Basics

## Introduction

BigMemory Go has three storage tiers, summarized here:

- Memory store – Heap memory that holds a copy of the hottest subset of data from the off-heap store. Subject to Java GC.
- Off-heap store – Limited in size only by available RAM. Not subject to Java GC. Can store serialized data only. Provides overflow capacity to the memory store.
- Disk store – Backs up in-memory data and provides overflow capacity to the other tiers. Can store serialized data only.

This document defines these storage tiers and details the suitable element types for each storage tier.

Before running in production, it is strongly recommended that you test the BigMemory Go tiers with the actual amount of data you expect to use in production. For information about sizing the tiers, refer to [Cache Configuration Sizing Attributes](#).

## Memory Store

The memory store is always enabled and exists in heap memory. For the best performance, allot as much heap memory as possible without triggering GC pauses, and use the [off-heap store](#) to hold the data that cannot fit in heap (without causing GC pauses).

The memory store has the following characteristics:

- Accepts all data, whether serializable or not
- Fastest storage option
- Thread safe for use by multiple concurrent threads

## Off-Heap Store

The off-heap store extends the in-memory store to memory outside the of the object heap. This store, which is not subject to Java GC, is limited only by the amount of RAM available.

Because off-heap data is stored in bytes, only data that is `Serializable` is suitable for the off-heap store. Any non serializable data overflowing to the `OffHeapMemoryStore` is simply removed, and a `WARNING` level log message emitted.

Since serialization and deserialization take place on putting and getting from the off-heap store, it is theoretically slower than the memory store. This difference, however, is mitigated when GC involved with larger heaps is taken into account.

## Allocating Direct Memory in the JVM

The off-heap store uses the direct-memory portion of the JVM. You must allocate sufficient direct memory for the off-heap store by using the JVM property `MaxDirectMemorySize`.

## Allocating Direct Memory in the JVM

For example, to allocate 2GB of direct memory in the JVM:

```
java -XX:MaxDirectMemorySize=2G ...
```

Since direct memory may be shared with other processes, allocate at least 256MB (or preferably 1GB) more to direct memory than will be [allocated to the off-heap store](#).

Note the following about allocating direct memory:

- If you configure off-heap memory but do not allocate direct memory with `-XX:MaxDirectMemorySize`, the default value for direct memory 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.
- `MaxDirectMemorySize` must be added to the local node's startup environment.
- Direct memory, 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 allocated must be within the constraints of available system memory and configured off-heap memory.
- 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.

## Using Off-Heap Store with 32-bit JVMs

The amount of heap-offload you can achieve is limited by addressable memory. 64-bit systems can allow as much memory as the hardware and operating system can handle, while 32-bit systems have strict limitations on the amount of memory that can be effectively managed.

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 operating-system (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 `OutOfMemoryError` on startup, but one is likely to occur at some point during the JVM's life.

If Java GC issues are afflicting a 32-bit JVM, then off-heap store can help. However, note the following:

- Everything has to fit in 4GB of addressable space. If 2GB of heap is allocated (with `-Xmx2G`) then at most are are 2GB left for off-heap data.
- The JVM process requires some of the 4GB of addressable space for its code and shared libraries plus any extra Operating System overhead.
- Allocating a 3GB heap with `-Xmx`, as well as 2047MB of off-heap memory, will not cause an error at startup, but when it's time to grow the heap an `OutOfMemoryError` is likely.
- If both `-Xms3G` and `-Xmx3G` are used with 2047MB of off-heap memory, the virtual machine will start but then complain as soon as the off-heap store tries to allocate the off-heap buffers.
- Some APIs, such as `java.util.zip.ZipFile` on a 1.5 JVM, may `<mmap>` files in memory. This will also use up process space and may trigger an `OutOfMemoryError`.

## Tuning Off-Heap Store Performance

Memory-related or performance issues that arise during operations can be related to improper allocation of memory to the off-heap store. If performance or functional issues arise that can be traced back to the off-heap store, see the suggested tuning tips in this section.

### 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.

In addition, be sure to allocate at least 15 percent more off-heap memory than the size of your data set. To maximize performance, a portion of off-heap memory is reserved for meta-data and other purposes.

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.

### 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:+UseCompressedOops`. For IBM JVMs, use `-Xcompressedrefs`.

### Slow Off-Heap Allocation

Based configuration, usage, and memory requirements, BigMemory Go could allocate off-heap memory multiple times. If off-heap memory comes under pressure due to over-allocation, the host OS may begin paging to disk, thus slowing down allocation operations. As the situation worsens, an off-heap buffer too large to fit in memory can quickly deplete critical system resources such as RAM and swap space and crash the host OS.

To stop this situation from degrading, off-heap allocation time is measured to avoid allocating buffers too large to fit in memory. If it takes more than 1.5 seconds to allocate a buffer, a warning is issued. If it takes more than 15 seconds, the JVM is halted with `System.exit()` (or a different method if the Security Manager prevents this).

To prevent a JVM shutdown after a 15-second delay has occurred, set the

## Swapiness and Huge Pages

net.sf.ehcache.offheap.DoNotHaltOnCriticalAllocationDelay system property to true. In this case, an error is logged instead.

## Swapiness and Huge Pages

An OS could 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 heap 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*, *large pages*, and *superpages*). Settings for these behaviors vary by OS and JVM. For Oracle HotSpot, `-XX:+UseLargePages` and `-XX:LargePageSizeInBytes=<size>` (where `<size>` is a value allowed by the OS for specific CPUs) can be used to control page size. However, note that this setting does not affect how off-heap memory is allocating. Over-allocating huge pages while also configuring substantial off-heap memory *can starve off-heap allocation and lead to memory and performance problems*.

## Maximum Serialized Size of an Element

This section applies when using BigMemory through the Ehcache API.

Unlike the memory and the disk stores, by default the off-heap store has a 4MB limit for classes with high quality hashcodes, and 256KB limit for those with pathologically bad hashcodes. The built-in classes such as `String` and the `java.lang.Number` subclasses `Long` and `Integer` have high quality hashcodes. This can issues when objects are expected to be larger than the default limits.

To override the default size limits, set the system property

`net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize` to the size you require.

For example,

```
net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M
```

## Reducing Faulting

While the memory store holds a hotset (a subset) of the entire data set, the off-heap store should be large enough to hold the entire data set. The frequency of misses (get operations that fail to find the data in memory) begins to rise when the data is too large to fit into off-heap memory, forcing gets to fetch data from the disk store (called *faulting*). More misses in turn raise latency and lower performance.

For example, tests with a 4GB data set and a 5GB off-heap store recorded no misses. With the off-heap store reduced to 4GB, 1.7 percent of cache operations resulted in misses. With the off-heap store at 3GB, misses reached 15 percent.

## Disk Store

The disk store provides a thread-safe disk-spooling facility that can be used for additional storage and for persisting data through system restarts.

For more information about data persistence on disk, refer to the [Fast Restartability](#) page.

## Serialization

Only data that is `Serializable` can be placed in the disk store. Writes to and from the disk use `ObjectInputStream` and the Java serialization mechanism. Any non-serializable data overflowing to the disk store is removed and a `NotSerializableException` is thrown.

Serialization speed is affected by the size of the objects being serialized and their type. It has been found that:

- The serialization time for a Java object consisting of a large Map of String arrays was 126ms, where the serialized size was 349,225 bytes.
- The serialization time for a byte[] was 7ms, where the serialized size was 310,232 bytes.

Byte arrays are 20 times faster to serialize, making them a better choice for increasing disk-store performance.

## Storage Options

Two disk-store options are available:

- Temporary swap – Allows the memory and off-heap stores to overflow to disk when they become full. This makes the disk a temporary store because overflow data does not survive restarts or failures. When the node is restarted, any existing data on disk is cleared because it is not designed to be reloaded.
- Restartable store – Mirrors the data in memory and provides failure recovery of data. When the node is restarted, the data set is reloaded from disk to the in-memory stores.

# Storage Tiers Advanced

## Introduction

When using BigMemory Go through its Ehcache API, many aspects of the data storage tiers can be controlled through the Ehcache configuration.

## Configuring the Memory Store

The memory store is the top tier and is automatically used by BigMemory Go to store the data hotset because it is the fastest store. It requires no special configuration to enable, and its overall size is taken from the Java heap size. Since it exists in the heap, it is limited by Java GC constraints.

## Memory Use, Spooling, and Expiry Strategy in the Memory Store

All caches specify their maximum in-memory size, in terms of the number of elements, at configuration time.

When an element is added to a cache and it goes beyond its maximum memory size, an existing element is either deleted, if overflow is not enabled, or evaluated for spooling to another tier, if overflow is enabled. The overflow options are `overflowToOffHeap` and `<persistence>` (disk store).

If overflow is enabled, a check for expiry is carried out. If it is expired it is deleted; if not it is spooled. The eviction of an item from the memory store is based on the optional `MemoryStoreEvictionPolicy` attribute specified in the configuration file. Legal values are LRU (default), LFU and FIFO:

- **Least Recently Used (LRU)** \*Default\*\*—LRU is the default setting. The last-used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.
- **Least Frequently Used (LFU)**—For each get call on the element the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element with least number of hits, the Less Frequently Used element, is evicted.
- **First In First Out (FIFO)**—Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

For all the eviction policies there are also `putQuiet` and `getQuiet` methods which do not update the last used timestamp.

When there is a `get` or a `getQuiet` on an element, it is checked for expiry. If expired, it is removed and null is returned. Note that at any point in time there will usually be some expired elements in the cache. Memory sizing of an application must always take into account the maximum size of each cache.

TIP: Calculating the Size of a Cache

`calculateInMemorySize()` is a convenient method which can provide an estimate of the size (in bytes) of the memory store. It returns the serialized size of the cache, providing a rough estimate. Do not use this method in production as it has a negative effect on performance.

## Memory Use, Spooling, and Expiry Strategy in the Memory Store

An alternative would be to have an expiry thread. This is a trade-off between lower memory use and short locking periods and CPU utilization. The design is in favour of the latter. For those concerned with memory use, simply [reduce the cache's size in memory](#) for more information).

# Configuring the Off-Heap Store

If an off-heap store is configured, the corresponding memory store overflows to that off-heap store. Configuring an off-heap store can be done either through XML or programmatically. With either method, the off-heap store is configured on a per-cache basis.

## Declarative Configuration

The following XML configuration creates an off-heap cache with an in-heap store (maxEntriesLocalHeap) of 10,000 elements which overflow to a 1-gigabyte off-heap area.

```
<ehcache updateCheck="false" monitoring="off" dynamicConfig="false">
    <defaultCache maxEntriesLocalHeap="10000"
        eternal="true"
        memoryStoreEvictionPolicy="LRU"
        statistics="false" />

    <cache name="sample-offheap-cache"
        maxEntriesLocalHeap="10000"
        eternal="true"
        memoryStoreEvictionPolicy="LRU"
        overflowToOffHeap="true"
        maxBytesLocalOffHeap="1G"/>
</ehcache>
```

The configuration options are:

### **overflowToOffHeap**

Values may be true or false. When set to true, enables the cache to utilize off-heap memory storage to improve performance. Off-heap memory is not subject to Java GC cycles and has a size limit set by the Java property MaxDirectMemorySize. The default value is false.

### **maxBytesLocalOffHeap**

Sets the amount of off-heap memory available to the cache and is in effect only if `overflowToOffHeap` is true. The minimum amount that can be allocated is 1MB. There is no maximum.

For more information on sizing data stores, refer to [this page](#).

### NOTE: Heap Configuration When Using an Off-heap Store

You should set `maxEntriesLocalHeap` to at least 100 elements when using an off-heap store to avoid performance degradation. Lower values for `maxEntriesLocalHeap` trigger a warning to be logged.

## Programmatic Configuration

The equivalent cache can be created using the following programmatic configuration:

```
public Cache createOffHeapCache()
{
    CacheConfiguration config = new CacheConfiguration("sample-offheap-cache", 10000)
        .overflowToOffHeap(true).maxBytesLocalOffHeap("1G");
    Cache cache = new Cache(config);
    manager.addCache(cache);
    return cache;
}
```

## Configuring the Disk Store

Disk stores are configured on a per CacheManager basis. If one or more caches requires a disk store but none is configured, a default directory is used and a warning message is logged to encourage explicit configuration of the disk store path.

Configuring a disk store is optional. If all caches use only memory and off-heap stores, then there is no need to configure a disk store. This simplifies configuration, and uses fewer threads. This also makes it unnecessary to configure multiple disk store paths when multiple CacheManagers are being used.

## Disk-Store Storage Options

Two disk store options are available:

- Temporary overflow store ("localTempSwap")
- Persistent store ("localRestartable")

See the following sections for more information on these options.

### **localTempSwap**

The "localTempSwap" persistence strategy allows the cache to overflow to disk during cache operation, providing an extra tier for cache storage. This disk storage is temporary and is cleared after a restart.

If the disk store path is not specified, a default path is used, and the default will be auto-resolved in the case of a conflict with another CacheManager.

The localTempSwap disk store creates a data file for each cache on startup called "<cache\_name>.data".

### **localRestartable**

This option implements a restartable store that is a mirror of the in-memory cache. After any restart, data that was last in the cache will automatically load into the restartable store, and from there the data will be available to the cache.

The path to the directory where any required disk files will be created is configured with the <diskStore> sub-element of the Ehcache configuration. In order to use the restartable store, a unique and explicitly specified path is required.

## Disk Store Configuration Element

# Disk Store Configuration Element

Files are created in the directory specified by the <diskStore> configuration element. The <diskStore> element has one attribute called path.

```
<diskStore path="/path/to/store/data"/>
```

Legal values for the path attribute are legal file system paths. For example, for Unix:

```
/home/application/cache
```

The following system properties are also legal, in which case they are translated:

- user.home - User's home directory
- user.dir - User's current working directory
- java.io.tmpdir - Default temp file path
- ehcache.disk.store.dir - A system property you would normally specify on the command line—for example, `java -Dehcache.disk.store.dir=/u01/myapp/diskdir`.

Subdirectories can be specified below the system property, for example:

```
user.dir/one
```

To programmatically set a disk store path:

```
DiskStoreConfiguration diskStoreConfiguration = new DiskStoreConfiguration();  
  
diskStoreConfiguration.setPath("/my/path/dir");  
  
// Already created a configuration object ...  
configuration.addDiskStore(diskStoreConfiguration);  
  
CacheManager mgr = new CacheManager(configuration);
```

**Note:** A CacheManager's disk store path cannot be changed once it is set in configuration. If the disk store path is changed, the CacheManager must be recycled for the new path to take effect.

## Disk Store Expiry and Eviction

Expired elements are eventually evicted to free up disk space. The element is also removed from the in-memory index of elements.

One thread per cache is used to remove expired elements. The optional attribute `diskExpiryThreadIntervalSeconds` sets the interval between runs of the expiry thread.

**Warning:** Setting `diskExpiryThreadIntervalSeconds` to a low value can cause excessive disk-store locking and high CPU utilization. The default value is 120 seconds.

If a cache's disk store has a limited size, Elements will be evicted from the disk store when it exceeds this limit. The LFU algorithm is used for these evictions. It is not configurable or changeable.

## Disk Store Expiry and Eviction

**Note:** With the "localTempSwap" strategy, you can use `maxEntriesLocalDisk` or `maxBytesLocalDisk` at either the Cache or CacheManager level to control the size of the swap-to-disk area.

## Turning off Disk Stores

To turn off disk store path creation, comment out the `diskStore` element in `ehcache.xml`.

The default Ehcache configuration, `ehcache-failsafe.xml`, uses a disk store. To avoid use of a disk store, specify a custom `ehcache.xml` with the `diskStore` element commented out.

## Configuration Examples

These examples show how to allocate 8GB of machine memory to different stores. It assumes a data set of 7GB - say for a cache of 7M items (each 1kb in size).

Those who want minimal application response-time variance (or minimizing GC pause times), will likely want all the cache to be off-heap. Assuming that 1GB of heap is needed for the rest of the app, they will set their Java config as follows:

```
java -Xms1G -Xmx1G -XX:maxDirectMemorySize=7G
```

And their Ehcache config as:

```
<cache
  maxEntriesLocalHeap=100
  overflowToOffHeap="true"
  maxBytesLocalOffHeap="6G"
... />
```

### NOTE: Direct Memory and Off-heap Memory Allocations

To accommodate server communications layer requirements, the value of `maxDirectMemorySize` must be greater than the value of `maxBytesLocalOffHeap`. The exact amount greater depends upon the size of `maxBytesLocalOffHeap`. The minimum is 256MB, but if you allocate 1GB more to the `maxDirectMemorySize`, it will certainly be sufficient. The server will only use what it needs and the rest will remain available.

Those who want best possible performance for a hot set of data, while still reducing overall application response time variance, will likely want a combination of on-heap and off-heap. The heap will be used for the hot set, the offheap for the rest. So, for example if the hot set is 1M items (or 1GB) of the 7GB data. They will set their Java config as follows

```
java -Xms2G -Xmx2G -XX:maxDirectMemorySize=6G
```

And their Ehcache config as:

```
<cache
  maxEntriesLocalHeap=1M
  overflowToOffHeap="true"
  maxBytesLocalOffHeap="5G"
...>
```

## Configuration Examples

This configuration will compare VERY favorably against the alternative of keeping the less-hot set in a database (100x slower) or caching on local disk (20x slower).

Where the data set is small and pauses are not a problem, the whole data set can be kept on heap:

```
<cache  
    maxEntriesLocalHeap=1M  
    overflowToOffHeap="false"  
    ...>
```

Where latency isn't an issue overflow to disk can be used:

```
<cache  
    maxEntriesLocalHeap=1M  
    ...>
```

# Sizing Storage Tiers

## Introduction

Tuning BigMemory Go often involves sizing the data storage tiers appropriately. BigMemory Go provides a number of ways to size the different data tiers using simple sizing attributes. These sizing attributes affect memory and disk resources.

## Sizing Attributes

The following table summarizes the sizing attributes in BigMemory Go's Ehcache API.

| Tier                | Attribute                                | Pooling available at CacheManager Level? | Description   |
|---------------------|--|--|---|
| Memory Store (Heap) | maxEntriesLocalHeap<br>maxBytesLocalHeap | maxBytesLocalHeap only                   | The maximum number of entries or bytes a data set can use in local heap memory, or when set at the CacheManager level (maxBytesLocalHeap only), as a pool available to all data sets under that CacheManager. This setting is required for every cache or at the CacheManager level.  |
| Off-heap Store      | maxBytesLocalOffHeap                     | Yes                                      | The maximum number of bytes a data set can use in off-heap memory, or when set at the CacheManager level, as a pool available to all data sets under that CacheManager.   |
| Disk Store          | maxEntriesLocalDisk<br>maxBytesLocalDisk | maxBytesLocalDisk only                   | The maximum number of entries or bytes a data set can use on the local disk, or when set at the CacheManager level (maxBytesLocalDisk only), as a pool available to all data sets under that CacheManager. Note that these settings apply to temporary swapping of data to disk ("localTempSwap"); these settings do not apply to disk persistence. |

Attributes that set a number of entries or elements take an integer. Attributes that set a memory size (bytes) use the Java -Xmx syntax (for example: "500k", "200m", "2g") or percentage (for example: "20%"). Percentages, however, can be used only in the case where a CacheManager-level pool has been configured.

## Pooling Resources Versus Sizing Individual Data Sets

You can constrain the size of any data set on a specific tier in that data set's configuration. You can also constrain the size of all of a CacheManager's data sets in a specific tier by configuring an overall size at the CacheManager level.

If there is no CacheManager-level pool specified for a tier, an individual data set claims the amount of that tier specified in its configuration. If there is a CacheManager-level pool specified for a tier, an individual data set

## Pooling Resources Versus Sizing Individual Data Sets

claims that amount *from the pool*. In this case, data sets with no size configuration for that tier receive an equal share of the remainder of the pool (after data sets with explicit sizing configuration have claimed their portion).

For example, if a CacheManager with eight data sets pools one gigabyte of heap, and two data sets each explicitly specify 200MB of heap while the remaining data sets do not specify a size, the remaining data sets will share 600MB of heap equally. Note that data sets must use bytes-based attributes to claim a portion of a pool; entries-based attributes such as `maxEntriesLocal` cannot be used with a pool.

On startup, the sizes specified by data sets are checked to ensure that any CacheManager-level pools are not over-allocated. If over-allocation occurs for any pool, an `InvalidConfigurationException` is thrown. Note that percentages should not add up to more than 100% of a single pool.

If the sizes specified by data sets for any tier take exactly the entire CacheManager-level pool specified for that tier, a warning is logged. In this case, data sets that do not specify a size for that tier cannot use the tier, as nothing is left over.

## Memory Store (Heap)

A size must be provided for the heap, either in the CacheManager (`maxBytesLocalHeap` only) or in each individual cache (`maxBytesLocalHeap` or `maxEntriesLocalHeap`). Not doing so causes an `InvalidConfigurationException`.

If a pool is configured, it can be combined with a heap setting in an individual cache. This allows the cache to claim a specified portion of the heap setting configured in the pool. However, in this case the cache setting must use `maxBytesLocalHeap` (same as the CacheManager).

In any case, every cache **must** have a heap setting, either configured explicitly or taken from the pool configured in the CacheManager.

## Off-Heap Store

Off-heap sizing can be configured in bytes only, never by entries.

If a CacheManager has a pool configured for off-heap, your application cannot add caches dynamically that have off-heap configuration — doing so generates an error. In addition, if any caches that used the pool are removed programmatically or through the Terracotta Management Console (TMC), other caches in the pool cannot claim the unused portion. To allot the entire off-heap pool to the remaining caches, remove the unwanted cache from the Ehcache configuration and then reload the configuration.

To use off-heap as a data tier, a cache must have `overflowToOffHeap` set to "true". If a CacheManager has a pool configured for using off-heap, the `overflowToOffHeap` attribute is automatically set to "true" for all caches. In this case, you can prevent a specific cache from overflowing to off-heap by explicitly setting its `overflowToOffHeap` attribute to "false".

## Disk Store

The local disk can be used as a data tier, either for temporarily swapping data or for disk persistence, but not both at once.

## Disk Store

To use the disk for swapping, set the `persistenceStrategy` to "localTempSwap" (refer to [Temporary Disk Storage](#)), and use the `maxBytesLocalDisk` setting to configure the size of this tier.

To use the disk for disk persistence, refer to [Disk Persistence Implementation](#).

# Sizing Examples

The following examples illustrate both pooled and individual cache-sizing configurations.

## Pooled Resources

The following configuration sets pools for all of this CacheManager's caches:

```
<ehcache xmlns...
    Name="CM1"
    maxBytesLocalHeap="100M"
    maxBytesLocalOffHeap="10G"
    maxBytesLocalDisk="50G">
    ...
<cache name="Cache1" ... </cache>
<cache name="Cache2" ... </cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

CacheManager CM1 automatically allocates these pools equally among its three caches. Each cache gets one third of the allocated heap, off-heap, and local disk. Note that at the CacheManager level, resources can be allocated in bytes only.

## Explicitly Sizing Data Sets

You can explicitly allocate resources to specific caches:

```
<ehcache xmlns...
    Name="CM1"
    maxBytesLocalHeap="100M"
    maxBytesLocalOffHeap="10G"
    maxBytesLocalDisk="60G">
    ...
<cache name="Cache1" ...
    maxBytesLocalHeap="50M"
    ...
</cache>

<cache name="Cache2" ...
    maxBytesLocalOffHeap="5G"
    ...
</cache>
<cache name="Cache3" ... </cache>

</ehcache>
```

## Explicitly Sizing Data Sets

In the example above, Cache1 reserves 50Mb of the 100Mb local-heap pool; the other caches divide the remaining portion of the pool equally. Cache2 takes half of the local off-heap pool; the other caches divide the remaining portion of the pool equally. Cache3 receives 25Mb of local heap, 2.5Gb of off-heap, and 20Gb of the local disk.

Caches that reserve a portion of a pool are not required to use that portion. Cache1, for example, has a fixed portion of the local heap but may have any amount of data in heap up to the configured value of 50Mb.

Note that caches must use the same sizing attributes used to create the pool. Cache1, for example, cannot use maxEntriesLocalHeap to reserve a portion of the pool.

## Mixed Sizing Configurations

If a CacheManager does not pool a particular resource, that resource can still be allocated in cache configuration, as shown in the following example.

```
<ehcache xmlns...
    Name="CM2"
    maxBytesLocalHeap="100M">
    ...
<cache name="Cache4" ...
    maxBytesLocalHeap="50M"
    maxEntriesLocalDisk="100000"
    ...
</cache>
<cache name="Cache5" ...
    maxBytesLocalOffHeap="10G"
    ...
</cache>
<cache name="Cache6" ... </cache>
</ehcache>
```

CacheManager CM2 creates one pool (local heap). Its caches all use the local heap and are constrained by the pool setting, as expected. However, cache configuration can allocate other resources as desired. In this example, Cache4 allocates disk space for its data, and Cache5 allocates off-heap space for its data. Cache6 gets 25Mb of local heap only.

## Using Percents

The following configuration sets pools for each tier:

```
<ehcache xmlns...
    Name="CM1"
    maxBytesLocalHeap="1G"
    maxBytesLocalOffHeap="10G"
    maxBytesLocalDisk="50G">
    ...
<!-- Cache1 gets 400Mb of heap, 2.5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache1" ...
    maxBytesLocalHeap="40%">
</cache>
```

## Using Percents

```
<!-- Cache2 gets 300Mb of heap, 5Gb of off-heap, and 5Gb of disk. -->
<cache name="Cache2" ...
maxBytesLocalOffHeap="50%">
</cache>

<!-- Cache2 gets 300Mb of heap, 2.5Gb of off-heap, and 40Gb of disk. -->
<cache name="Cache3" ...
maxBytesLocalDisk="80%">
</cache>
</ehcache>
```

NOTE: Configuring Cache Sizes with Percentages

You can use a percentage of the total JVM heap for the CacheManager maxBytesLocalHeap. The CacheManager percentage, then, is a portion of the total JVM heap, and in turn, the Cache percentage is the portion of the CacheManager pool for that tier.

## Sizing Data Sets Without a Pool

The CacheManager in this example does not pool any resources.

```
<ehcache xmlns...
    Name="CM3"
    ...
    ...

<cache name="Cache7" ...
    maxBytesLocalHeap="50M"
    maxEntriesLocalDisk="100000"
    ...
</cache>

<cache name="Cache8" ...
    maxEntriesLocalHeap="1000"
    maxBytesLocalOffHeap="10G"
    ...
</cache>
<cache name="Cache9" ...
    maxBytesLocalHeap="50M"
    ...
</cache>

</ehcache>
```

Caches can be configured to use resources as necessary. Note that every cache in this example must declare a value for local heap. This is because no pool exists for the local heap; implicit (CacheManager configuration) or explicit (cache configuration) local-heap allocation is required.

## Overflows

Caches that do not specify overflow will overflow if a pool is set for off-heap and disk.

```
<ehcache maxBytesLocalHeap="1g" maxBytesLocalOffHeap="4g"
    maxBytesLocalDisk="100g" >

<cache name="explicitlyAllocatedCache1"
    maxBytesLocalHeap="50m"
    maxBytesLocalOffHeap="200m"
```

## Overflows

```
        timeToLiveSeconds="100">
</cache>

<!-- Does not overflow to disk because overflow has been explicitly
disabled. -->
<cache name="explicitlyAllocatedCache2"
      maxLocalHeap="10%"
      maxBytesLocalOffHeap="200m"
      timeToLiveSeconds="100">
    <persistence strategy="none"/>
</cache>

<!-- Overflows automatically to off-heap and disk because no specific override and
resources are set at the CacheManager level -->
<cache name="automaticallyAllocatedCache1"
      timeToLiveSeconds="100">
</cache>

<!-- Does not use off-heap because overflow has been explicitly
disabled. -->
<cache name="automaticallyAllocatedCache2"
      timeToLiveSeconds="100"
      overflowToOffHeap="false">
</cache>
</ehcache>
```

## Overriding Size Limitations

Pinned caches can override the limits set by cache-configuration sizing attributes, potentially causing OutOfMemory errors. This is because pinning prevents flushing of cache entries to lower tiers. For more information on pinning, see [Pinning, Eviction, and Expiration](#).

## Built-In Sizing Computation and Enforcement

Internal Ehcache mechanisms track data-element sizes and enforce the limits set by CacheManager sizing pools.

### Sizing of Data Set Entries

Elements put in a memory-limited cache will have their memory sizes measured. The entire Element instance added to the cache is measured, including key and value, as well as the memory footprint of adding that instance to internal data structures. Key and value are measured as object graphs – each reference is followed and the object reference also measured. This goes on recursively.

Shared references will be measured by each class that references it. This will result in an overstatement. Shared references should therefore be ignored.

### Ignoring for Size Calculations

For the purposes of measurement, references can be ignored using the `@IgnoreSizeOf` annotation. The annotation may be declared at the class level, on a field, or on a package. You can also specify a file containing the fully qualified names of classes, fields, and packages to be ignored.

## Sizing of Data Set Entries

This annotation is not inherited, and must be added to any subclasses that should also be excluded from sizing.

The following example shows how to ignore the `Dog` class.

```
@IgnoreSizeOf
public class Dog {
    private Gender gender;
    private String name;
}
```

The following example shows how to ignore the `sharedInstance` field.

```
public class MyCacheEntry {
    @IgnoreSizeOf
    private final SharedClass sharedInstance;
    ...
}
```

Packages may be also ignored if you add the `@IgnoreSizeOf` annotation to appropriate `package-info.java` of the desired package. Here is a sample `package-info.java` for and in the `com.company.ignore` package:

```
@IgnoreSizeOf
package com.company.ignore;
import net.sf.ehcache.pool.sizeof.filter.IgnoreSizeOf;
```

Alternatively, you may declare ignored classes and fields in a file and specify a `net.sf.ehcache.sizeof.filter` system property to point to that file.

```
# That field references a common graph between all cached entries
com.company.domain.cache.MyCacheEntry.sharedInstance

# This will ignore all instances of that type
com.company.SharedState

# This ignores a package
com.company.example
```

Note that these measurements and configurations apply only to on-heap storage. Once Elements are moved to off-heap memory or disk, they are serialized as byte arrays. The serialized size is then used as the basis for measurement.

## Configuration for Limiting the Traversed Object Graph

As noted above, sizing caches involves traversing object graphs, a process that can be limited with annotations. This process can also be controlled at both the CacheManager and cache levels.

### Size-Of Limitation at the CacheManager Level

Control how deep the size-of engine can go when sizing on-heap elements by adding the following element at the CacheManager level:

```
<sizeOfPolicy maxDepth="100" maxDepthExceededBehavior="abort"/>
```

This element has the following attributes

## Sizing of Data Set Entries

- `maxDepth` – Controls how many linked objects can be visited before the size-of engine takes any action. This attribute is required.
- `maxDepthExceededBehavior` – Specifies what happens when the max depth is exceeded while sizing an object graph:
  - ◆ "continue" – (DEFAULT) Forces the size-of engine to log a warning and continue the sizing operation. If this attribute is not specified, "continue" is the behavior used.
  - ◆ "abort" – Forces the SizeOf engine to abort the sizing, log a warning, and mark the cache as not correctly tracking memory usage. With this setting, `Ehcache.hasAbortedSizeOf()` returns true.

The `SizeOf` policy can be configured at the `CacheManager` level (directly under `<ehcache>`) and at the cache level (under `<cache>` or `<defaultCache>`). The cache policy always overrides the `CacheManager` if both are set.

### Size-Of Limitation at the Cache level

Use the `<sizeOfPolicy>` as a subelement in any `<cache>` block to control how deep the size-of engine can go when sizing on-heap elements belonging to the target cache. This cache-level setting overrides the `CacheManager` size-of setting.

### Debugging of Size-Of Related Errors

If warnings or errors appear that seem related to size-of measurement (usually caused by the size-of engine walking the graph), generate more log information on sizing activities:

- Set the `net.sf.ehcache.sizeof.verboseDebugLogging` system property to true.
- Enable debug logs on `net.sf.ehcache.pool.sizeof` in your chosen implementation of SLF4J.

## Eviction When Using CacheManager-Level Storage

When a `CacheManager`-level storage pool is exhausted, a cache is selected on which to perform eviction to recover pool space. The eviction from the selected cache is performed using the cache's configured eviction algorithm (LRU, LFU, etc...). The cache from which eviction is performed is selected using the "minimal eviction cost" algorithm described below:

$$\text{eviction-cost} = \text{mean-entry-size} * \text{drop-in-hit-rate}$$

Eviction cost is defined as the increase in bytes requested from the underlying SOR (System of Record, e.g., database) per unit time used by evicting the requested number of bytes from the cache.

If we model the hit distribution as a simple power-law then:

$$P(\text{hit } n\text{'th element}) \sim 1/n^{\alpha}$$

In the continuous limit, this means the total hit rate is proportional to the integral of this distribution function over the elements in the cache. The change in hit rate due to an eviction is then the integral of this distribution function between the initial size and the final size. Assuming that the eviction size is small compared to the overall cache size, we can model this as:

$$\text{drop} \sim \text{access} * 1/x^{\alpha} * \Delta(x)$$

## Eviction When Using CacheManager-Level Storage

where 'access' is the overall access rate (hits + misses), and  $x$  is a unit-less measure of the 'fill level' of the cache. Approximating the fill level as the ratio of hit rate to access rate, and substituting in to the eviction-cost expression, we get:

```
eviction-cost = mean-entry-size * access * 1/(hits/access)^{alpha}
               * (eviction / (byteSize / (hits/access)))
```

Simplifying:

```
eviction-cost = (byteSize / countSize) * access * 1/(h/A)^{alpha}
               * (eviction * hits)/(access * byteSize)
eviction-cost = (eviction * hits) / (countSize * (hits/access)^{alpha})
```

Removing the common factor of 'eviction', which is the same in all caches, lead us to evicting from the cache with the minimum value of:

```
eviction-cost = (hits / countSize) / (hits/access)^{alpha}
```

When a cache has a zero hit-rate (it is in a pure loading phase), we deviate from this algorithm and allow the cache to occupy  $1/n$ 'th of the pool space, where ' $n$ ' is the number of caches using the pool. Once the cache starts to be accessed, we re-adjust to match the actual usage pattern of that cache.

# Pinning, Expiration, and Eviction

## Introduction

This page covers managing the life of the data in each of BigMemory's data storage tiers, including the pinning features of Automatic Resource Control (ARC).

The following are options for data life within the BigMemory tiers:

- Flush – To move an entry to a lower tier. Flushing is used to free up resources while still keeping data in BigMemory.
- Fault – To copy an entry from a lower tier to a higher tier. Faulting occurs when data is required at a higher tier but is not resident there. The entry is not deleted from the lower tiers after being faulted.
- Eviction – To remove an entry from BigMemory. The entry is deleted; it can only be reloaded from an outside source. Entries are evicted to free up resources.
- Expiration – A status based on Time To Live and Time To Idle settings. To maintain performance, expired entries may not be immediately flushed or evicted.
- Pinning – To force data to remain in certain tiers. Pinning can be set on individual entries or an entire cache, and must be used with caution to avoid exhausting a resource such as heap.

## Setting Expiration

BigMemory data entries expire based on parameters with configurable values. When eviction occurs, expired elements are the first to be removed. Having an effective expiration configuration is critical to optimizing the use of resources such as heap and maintaining overall performance.

To add expiration, edit the values of the following <cache> attributes, and tune these values based on results of performance tests:

- `timeToIdleSeconds` – The maximum number of seconds an element can exist in the BigMemory data store without being accessed. The element expires at this limit and will no longer be returned from BigMemory. 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 the BigMemory data store regardless of use. The element expires at this limit and will no longer be returned from BigMemory. The default value is 0, which means no TTL eviction takes place (infinite lifetime).
- `eternal` – If the cache's `eternal` flag is set, it overrides any finite TTI/TTL values that have been set. Individual cache elements may also be set to eternal. Eternal elements and caches do not expire, however they may be evicted.

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

## Pinning Data

Data that should remain in the data store regardless of resource constraints can be pinned. You can pin individual entries, or an entire cache.

## Pinning Individual Cache Entries

### Pinning Individual Cache Entries

Some APIs like OpenJPA and Hibernate require pinning of specific Elements. Specific entries can be programmatically pinned to the containing cache:

```
cache.setPinned(key, true);
```

The entry can be unpinned by the same method:

```
cache.setPinned(key, false);
```

To unpin all of a cache's pinned entries:

```
cache.unpinAll();
```

To check if an entry is pinned:

```
cache.isPinned(key); // Returns a boolean: true if the key is pinned.
```

Pinning a cache entry guarantees its storage in memory (heap or off-heap).

Note that pinning is a status applied to a cache entry's key. The entry's value may be null, and any operations on the value have no effect on the pinning status.

### Pinning a Cache

Entire caches can be pinned using the `pinning` element in cache configuration. This element has a required attribute (`store`) to specify which data tiers the cache should be pinned to:

```
<pinning store="localMemory" />
```

The `store` attribute can have one of the following values:

- `localHeap` – Cache data is pinned to the memory store (on heap). Unexpired entries can never be flushed to a lower tier or be evicted.
- `localMemory` – Cache data is pinned to the memory store or the off-heap store. Unexpired entries can never be flushed to a lower tier or be evicted.
- `inCache` – Cache data is pinned in the cache, which can be in any tier cache data is stored. The tier is chosen based on performance-enhancing efficiency algorithms.

For example, the following cache is configured to pin its entries:

```
<cache name="Cache1" ... >
    <pinning store="inCache" />
</cache>
```

The following cache is configured to pin its entries to heap or off-heap only:

```
<cache name="Cache2" ... >
    <pinning store="localMemory" />
</cache>
```

## Pinning a Cache

If only a limited set of a cache's entries should be pinned, it may be more efficient to pin those individual elements rather than the entire cache.

## Scope of Pinning

Pinning achieved programmatically will not be persisted — after a restart the pinned entries are no longer pinned. Cache pinning in configuration is reinstated with the configuration file.

# How Configuration Affects Element Flushing and Eviction

The following example shows a cache with certain expiration settings:

```
<cache name="myCache"
      eternal="false" timeToIdleSeconds="3600"
      timeToLiveSeconds="0" memoryStoreEvictionPolicy="LFU">
</cache>
```

Note the following about the myCache configuration:

- If a client accesses an entry in myCache that has been idle for more than an hour (timeToIdleSeconds), that element is evicted.
- If an entry expires but is not accessed, and no resource constraints force eviction, then the expired entry remains in place.
- Entries in myCache can live forever if accessed at least once per 60 minutes (timeToLiveSeconds). However, unexpired entries may still be flushed based on other limitations (see [Sizing BigMemory Tiers](#)).

## Pinning Overrides Cache Sizing

Pinning takes priority over configured cache sizes. For example, in the following cache the pinning configuration overrides the maxEntriesOnHeap setting:

```
<cache name="myCache"
      maxEntriesOnHeap="10000"
      maxBytesLocalOffHeap="8g"
      ... >
  <pinning store="localHeap" />
</cache>
```

While expired cache entries (even ones that have been pinned) can always be flushed and eventually evicted, most non-expired elements can be flushed and evicted as well, if resource limitations are reached. However, pinned elements, whether pinned individually or resident in a pinned cache, cannot be evicted if they haven't expired.

### CAUTION: Pinning Could Cause Failures

Potentially, pinned caches could grow to an unlimited size. Caches should never be pinned unless they are designed to hold a limited amount of data (such as reference data) or their usage and expiration characteristics are understood well enough to conclude that they cannot cause errors.

## Data Freshness and Expiration

Databases and other Systems of Record (SOR) that were not built to accommodate caching outside of the database do not normally come with any default mechanism for notifying external processes when data has been updated or modified.

When using BigMemory as a caching system, the following strategies can help to keep the data in the cache in sync:

- **Data Expiration:** Use the eviction algorithms included with Ehcache, along with the `timeToIdleSeconds` and `timeToLiveSeconds` settings, to enforce a maximum time for elements to live in the cache (forcing a re-load from the database or SOR).
- **Message Bus:** Use an application to make all updates to the database. When updates are made, post a message onto a message queue with a key to the item that was updated. All application instances can subscribe to the message bus and receive messages about data that is updated, and can synchronize their local copy of the data accordingly (for example by invalidating the cache entry for updated data)
- **Triggers:** Using a database trigger can accomplish a similar task as the message bus approach. Use the database trigger to execute code that can publish a message to a message bus. The advantage to this approach is that updates to the database do not have to be made only through a special application. The downside is that not all database triggers support full execution environments and it is often unadvisable to execute heavy-weight processing such as publishing messages on a queue during a database trigger.

The Data Expiration method is the simplest and most straightforward. It gives you the most control over the data synchronization, and doesn't require cooperation from any external systems. You simply set a data expiration policy and let Ehcache expire data from the cache, thus allowing fresh reads to re-populate and re-synchronize the cache.

If you choose the Data Expiration method, you can read more about the cache configuration settings at [cache eviction algorithms](#), and review the [timeToIdle](#) and [timeToLive configuration settings](#). The most important consideration when using the expiration method is balancing data freshness with database load. The shorter you make the expiration settings - meaning the more "fresh" you try to make the data - the more load you will incur on the database.

Try out some numbers and see what kind of load your application generates. Even modestly short values such as 5 or 10 minutes will afford significant load reductions.

# Fast Restartability

## Introduction

BigMemory's Fast Restart feature provides enterprise-ready crash resilience by keeping a fully consistent, real-time record of your in-memory data. After any kind of shutdown — planned or unplanned — the next time your application starts up, all of the data that was in BigMemory is still available and very quickly accessible.

The advantages of the Fast Restart feature include:

- In-memory data survives crashes and enables fast restarts. Because your in-memory data does not need to be reloaded from a remote data source, applications can resume at full speed after a restart.
- A real-time record of your in-memory data provides true fault tolerance. Even with BigMemory, where terabytes of data can be held in memory, the Fast Restart feature provides the equivalent of a local "hot mirror," which guarantees full data consistency.
- A consistent record of your in-memory data opens many possibilities for business innovation, such as arranging data sets according to time-based needs or moving data sets around to different locations. The uses of the Fast Restart store can range from a simple key-value persistence mechanism with fast read performance, to an operational store with in-memory speeds during operation for both reads and writes.

## Data Persistence Implementation

The BigMemory Fast Restart feature works by creating a real-time record of the in-memory data, which it persists in a Fast Restart store on the local disk. After any restart, the data that was last in memory (both heap and off-heap stores) automatically loads from the Fast Restart store back into memory.

Data persistence is configured by adding the `<persistence>` sub-element to a cache configuration. The `<persistence>` sub-element includes two attributes: `strategy` and `synchronousWrites`.

```
<cache>
  <persistence strategy="localRestartable|localTempSwap|none|distributed"
               synchronousWrites="false|true"/>
</cache>
```

## Strategy Options

The options for the `strategy` attribute are:

- "**localRestartable**" — Enables the Fast Restart feature which automatically logs all BigMemory data. This option provides fast restartability with fault tolerant data persistence.
- "**localTempSwap**" — Temporarily swaps data to disk when the space allotted for BigMemory is full. This option provides an extra tier for data storage during operation, but this store is not persisted. After a restart, the disk is cleared of any data.
- "**none**" — Does not offload data to disk. With this option, all of the working data is kept in memory only. This is the default mode.
- "**distributed**" — Defers to the `<terracotta>` configuration for persistence settings. This option is for BigMemory Max only.

## Synchronous Writes Options

# Synchronous Writes Options

If the `strategy` attribute is set to "localRestartable", then the `synchronousWrites` attribute may be configured. The options for `synchronousWrites` are:

- **`synchronousWrites="false"`** — This option specifies that an eventually consistent record of the data is kept on disk at all times. Writes to disk happen when efficient, and cache operations proceed without waiting for acknowledgement of writing to disk. After a restart, the data is recovered as it was when last synced. This option is faster than `synchronousWrites="true"`, but after a crash, the last 2-3 seconds of written data may be lost.

If not specified, the default for `synchronousWrites` is "false".

- **`synchronousWrites="true"`** — This option specifies that a fully consistent record of the data is kept on disk at all times. As changes are made to the data set, they are synchronously recorded on disk. The write to disk happens before a return to the caller. After a restart, the data is recovered exactly as it was before shutdown. This option is slower than `synchronousWrites="false"`, but after a crash, it provides full data consistency.

For transaction caching with `synchronousWrites`, soft locks are used to protect access. If there is a crash in the middle of a transaction, then upon recovery the soft locks are cleared on next access.

**Note:** The `synchronousWrites` attribute is also available in the `<terracotta>` sub-element. If configured in both places, it must have the same value.

## Disk Store Path

The path to the directory where any required disk files will be created is configured with the `<diskStore>` sub-element of the Ehcache configuration.

- For "localRestartable", a unique and explicitly specified path is required.
- For "localTempSwap", if the disk store path is not specified, a default path is used for the swap-to-disk area, and the default path will be auto-resolved in the case of a conflict with another CacheManager.

**Note:** The Fast Restart feature does not use the disk tier in the same way that conventional disk persistence does. Therefore, when configured for "localRestartable", `diskStore` size measures such as `Cache.getDiskStoreSize()` or `Cache.calculateOnDiskSize()` are not applicable and will return zero. On the other hand, when configured for "localTempSwap", these measures will return size values.

## Configuration Examples

This section presents possible disk usage configurations for BigMemory Go.

## Options for Crash Resilience

The following configuration provides fast restartability with fully consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
```

## Options for Crash Resilience

```
<persistence strategy="localRestartable" synchronousWrites="true"/>
</cache>
</ehcache>
```

The following configuration provides fast restartability with eventually consistent data persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="false"/>
  </cache>
</ehcache>
```

## Temporary Disk Storage

The "localTempSwap" persistence strategy allows the in-memory data to overflow to disk during BigMemory operation. The disk storage is temporary and is cleared after a restart.

```
<ehcache>
  <diskStore path="java.io.tmpdir"/>
  <cache>
    <persistence strategy="localTempSwap"/>
  </cache>
</ehcache>
```

**Note:** With the "localTempSwap" strategy, you can use `maxEntriesLocalDisk` or `maxBytesLocalDisk` at either the Cache or CacheManager level to control the size of the swap-to-disk area.

## In-memory Only

When the persistence strategy is "none", all data stays in memory (with no overflow to disk nor persistence on disk).

```
<cache>
  <persistence strategy="none"/>
</cache>
```

## Programmatic Configuration Example

The following is an example of how to programmatically configure disk persistence:

```
Configuration cacheManagerConfig = new Configuration()
  .diskStore(new DiskStoreConfiguration()
    .path("/path/to/store/data"));
CacheConfiguration cacheConfig = new CacheConfiguration()
  .name("my-cache")
  .maxBytesLocalHeap(16, MemoryUnit.MEGABYTES)
  .maxBytesLocalOffHeap(256, MemoryUnit.MEGABYTES)
  .persistence(new PersistenceConfiguration().strategy(Strategy.LOCALRESTARTABLE));

cacheManagerConfig.addCache(cacheConfig);

CacheManager cacheManager = new CacheManager(cacheManagerConfig);
Ehcache myCache = cacheManager.getEhcache("my-cache");
```

## Fast Restart Performance

When configured for fast restartability ("localRestartable" persistence strategy), BigMemory becomes active on restart after all of the in-memory data is loaded. The amount of time until BigMemory is restarted is proportionate to the amount of in-memory data and the speed of the underlying infrastructure. Generally, recovery can occur as fast as the disk speed. With an SSD, for example, if you have a read throughput of 1 GB per second, you will see a similar loading speed during recovery.

## Fast Restart Limitations

The following recommendations should be observed when configuring BigMemory for fast restartability:

- The size of on-heap or off-heap stores should not be changed during a shutdown. If the amount of memory allocated is reduced, elements will be evicted upon restart.
- Restartable caches should not be removed from the CacheManager during a shutdown.
- If a restartable cache is disposed, the reference to the cache is deleted, but the cache contents remain in memory and on disk. After a restart, the cache contents are once again recovered into memory and on disk. The way to safely dispose of an unused restartable cache is to call clear on the cache and then dispose, so it doesn't take any space in disk or memory.

# Key Classes and Methods of the BigMemory Go API

## Introduction

BigMemory currently uses Ehcache as its user-facing data access API. See the [Ehcache API documentation](#) for details.

The key Ehcache classes used are:

- CacheManager
- Cache
- Element

These classes form the core of the BigMemory Go API. This document introduces these classes, along with other important components of the BigMemory Go API.

When applications use BigMemory Go through the Ehcache API, a CacheManager is instantiated to manage logical data sets (represented as Cache objects in the Ehcache API—though, they may be used to store any kind of data, not just cache data). These data sets contain name-value pairs called Elements.

The logical representations of these key components are actualized mostly through the classes discussed below. These classes' methods provide the main programmatic access to working with Ehcache.

## CacheManager

The class CacheManager is used to manage caches. Creation of, access to, and removal of caches is controlled by a named CacheManager.

### CacheManager Creation Modes

CacheManager supports two creation modes: singleton and instance. The two types can exist in the same JVM. However, multiple CacheManagers with the same name are not allowed to exist in the same JVM. CacheManager() constructors creating non-Singleton CacheManagers can violate this rule, causing a NullPointerException. If your code may create multiple CacheManagers of the same name in the same JVM, avoid this error by using the [static CacheManager.create\(\) methods](#), which always return the named (or default unnamed) CacheManager if it already exists in that JVM. If the named (or default unnamed) CacheManager does not exist, the CacheManager.create() methods create it.

For singletons, calling CacheManager.create(...) returns the existing singleton CacheManager with the configured name (if it exists) or creates the singleton based on the passed-in configuration.

To work from configuration, use the CacheManager.newInstance(...) method, which parses the passed-in configuration to either get the existing named CacheManager or create that CacheManager if it doesn't exist.

To review, the behavior of the CacheManager creation methods is as follows:

## CacheManager Creation Modes

- `CacheManager.newInstance(Configuration configuration)` – Create a new CacheManager or return the existing one named in the configuration.
- `CacheManager.create()` – Create a new singleton CacheManager with default configuration, or return the existing singleton. This is the same as `CacheManager.getInstance()`.
- `CacheManager.create(Configuration configuration)` – Create a singleton CacheManager with the passed-in configuration, or return the existing singleton.
- `new CacheManager(Configuration configuration)` – Create a new CacheManager, or throw an exception if the CacheManager named in the configuration already exists or if the parameter (`configuration`) is null.

Note that in instance-mode (non-singleton), where multiple CacheManagers can be created and used concurrently in the same JVM, each CacheManager requires its own configuration.

If the Caches under management use the disk store, the disk-store path specified in each CacheManager configuration should be unique. This is because when a new CacheManager is created, a check is made to ensure that no other CacheManagers are using the same disk-store path. Depending upon your persistence strategy, BigMemory Go will automatically resolve a disk-store path conflict, or it will let you know that you must explicitly configure the disk-store path.

If managed caches use only the memory store, there are no special considerations.

If a CacheManager is part of a cluster, there will also be listener ports which must be unique.

See the [API documentation](#) for more information on these methods, including options for passing in configuration. For examples, see [Code Samples](#).

## Cache

This is a thread-safe logical representation of a set of data elements, analogous to a cache region in many caching systems. Once a reference to a cache is obtained (through a CacheManager), logical actions can be performed. The physical implementation of these actions is relegated to the [stores](#).

Caches are instantiated from configuration or programmatically using one of the `Cache()` constructors. Certain cache characteristics, such as ARC-related sizing, and pinning, must be set using configuration.

Cache methods can be used to get information about the cache (for example, `getCacheManager()`, `isNodeBulkLoadEnabled()`, `isSearchable()`, etc.), or perform certain cache-wide operations (for example, `flush`, `load`, `initialize`, `dispose`, etc.).

The methods provided in the `Cache` class also allow you to work with cache elements (for example, `get`, `set`, `remove`, `replace`, etc.) as well as get information about them (for example, `isExpired`, `isPinned`, etc.).

## Element

An element is an atomic entry in a cache. It has a key, a value, and a record of accesses. Elements are put into and removed from caches. They can also expire and be removed by the cache, depending on the cache settings.

There is an API for Objects in addition to the one for Serializable. Non-serializable Objects can be stored only in heap. If an attempt is made to persist them, they are discarded with a DEBUG-level log message but no

## Element

error.

The APIs are identical except for the return methods from Element: `getKeyValue()` and `getObjectValue()` are used by the Object API in place of `getKey()` and `getValue()`.

# BigMemory Go Search API

## Introduction

The BigMemory Go Search API allows you to execute arbitrarily complex queries against caches with pre-built indexes. The development of alternative indexes on values provides the ability for data to be looked up based on multiple criteria instead of just keys.

Searchable attributes may be extracted from both keys and values. Keys, values, or summary values (Aggregators) can all be returned. Here is a simple example: Search for 32-year-old males and return the cache values.

```
Results results = cache.createQuery().includeValues()
    .addCriteria(age.eq(32).and(gender.eq("male"))).execute();
```

### What is Searchable?

Searches can be performed against Element keys and values, but they must be treated as attributes. Some Element keys and values are directly searchable and can simply be added to the search index as attributes. Some Element keys and values must be made searchable by extracting attributes with supported search types out of the keys and values. It is the attributes themselves which are searchable.

## Making a Cache Searchable

Caches can be made searchable, on a per cache basis, either by configuration or programmatically.

### By Configuration

Caches are made searchable by adding a <searchable/> tag to the ehcache.xml.

```
<cache name="cache2" maxBytesLocalHeap="16M" eternal="true" maxBytesLocalOffHeap="256M">
    <persistence strategy="localRestartable"/>
    <searchable/>
</cache>
```

This configuration will scan keys and values and, if they are of supported search types, add them as attributes called "key" and "value" respectively. If you do not want automatic indexing of keys and values, you can disable it with:

```
<cache name="cacheName" ...>
    <searchable keys="false" values="false"/>
    ...
</searchable>
</cache>
```

You might want to do this if you have a mix of types for your keys or values. The automatic indexing will throw an exception if types are mixed.

If you think that you will want to add search attributes after the cache is initialized, you can explicitly indicate the dynamic search configuration. Set the `allowDynamicIndexing` attribute to "true" to enable use of the dynamic attributes extractor (described in the [Defining Attributes](#) section below):

## By Configuration

```
<cache name="cacheName" ...>
    <searchable allowDynamicIndexing="true">
        ...
    </searchable>
</cache>
```

**Note:** Dynamic search is available for offheap-backed caches only.

Often keys or values will not be directly searchable and instead you will need to extract searchable attributes out of them. The following example shows this more typical case. Attribute Extractors are explained in more detail in the following section.

```
<cache name="cache3" maxEntriesLocalHeap="10000" eternal="true" maxBytesLocalOffHeap="10G">
    <persistence strategy="localRestartable"/>
    <searchable>
        <searchAttribute name="age" class="net.sf.ehcache.search.TestAttributeExtractor"/>
        <searchAttribute name="gender" expression="value.getGender()"/>
    </searchable>
</cache>
```

## Programmatically

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").className("net.sf.ehcache.search.TestAttributeExtractor"));
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 Search API, see the `net.sf.ehcache.search*` packages in this [Javadoc](#).

## Defining Attributes

In addition to configuring a cache to be searchable, you must define the attributes that will be used in searches.

Attributes are extracted from keys or values during search. This is done using `AttributeExtractors`. Extracted attributes must be one of the following supported types:

- Boolean
- Byte

## Defining Attributes

- Character
- Double
- Float
- Integer
- Long
- Short
- String
- java.util.Date
- java.sql.Date
- Enum

If an attribute cannot be extracted, due to not being found or being the wrong type, an AttributeExtractorException is thrown on search execution.

**Note:** On the first use of an attribute, the attribute type is detected, validated against supported types, and saved automatically. Once the type is established, it cannot be changed later. For example, if an integer value was initially returned for attribute named "Age" by the attribute extractor, then it is an error for the extractor to return a float for this attribute later on.

## Well-known Attributes

The parts of an Element that are well-known attributes can be referenced by some predefined, well-known names. If a key and/or value is of a supported search type, it is added automatically as an attribute with the name "key" or "value". These well-known attributes have the convenience of being constant attributes made available on the Query class. So, for example, the attribute for "key" may be referenced in a query by Query.KEY. For even greater readability, it is recommended to statically import so that, in this example, you would just use KEY.

### Well-known Attribute Name Attribute Constant

|       |             |
|-------|-------------|
| key   | Query.KEY   |
| value | Query.VALUE |

## Reflection Attribute Extractor

The ReflectionAttributeExtractor is a built-in search attribute extractor which uses JavaBean conventions and also understands a simple form of expression. Where a JavaBean property is available and it is of a searchable type, it can be simply declared:

```
<cache>
  <searchable>
    <searchAttribute name="age"/>
  </searchable>
</cache>
```

The expression language of the ReflectionAttributeExtractor also uses method/value dotted expression chains. The expression chain must start with one of either "key", "value", or "element". From the starting object a chain of either method calls or field names follows. Method calls and field names can be freely mixed in the chain. Some more examples:

```
<cache>
  <searchable>
    <searchAttribute name="age" expression="value.person.getAge()"/>
  </searchable>
```

## Well-known Attributes

```
</cache>
<cache>
  <searchable>
    <searchAttribute name="name" expression="element.toString()"/>
  </searchable>
</cache>
```

**Note:** The method and field name portions of the expression are case sensitive.

## Custom Attribute Extractor

In more complex situations, you can create your own attribute extractor by implementing the `AttributeExtractor` interface. The interface's `attributeFor` method allows you to specify the element to inspect and the attribute name, and it returns the attribute value.

**Note:** These examples assume there are previously created Person objects containing attributes such as name, age, and gender.

Provide your extractor class, as shown in the following example:

```
<cache name="cache2" maxEntriesLocalHeap="0" eternal="true">
  <persistence strategy="none"/>
  <searchable>
    <searchAttribute name="age" class="net.sf.ehcache.search.TestAttributeExtractor"/>
  </searchable>
</cache>
```

For example, a custom attribute extractor could be passed an Employee object and then extract a specific attribute:

```
returnValue = employee.getdept();
```

If you need to pass state to your custom extractor, you may do so with properties, as shown in the following example:

```
<cache>
  <searchable>
    <searchAttribute name="age"
      class="net.sf.ehcache.search.TestAttributeExtractor"
      properties="foo=this,bar=that,etc=12" />
  </searchable>
</cache>
```

If properties are provided, then the attribute extractor implementation must have a public constructor that accepts a single `java.util.Properties` instance.

## Dynamic Attributes Extractor

The `DynamicAttributesExtractor` provides flexibility by allowing the search configuration to be changed after the cache is initialized. This is done with one method call, at the point of element insertion into the cache. The `DynamicAttributesExtractor` method returns a map of attribute names to index and their respective values. This method is called for every `Ehcache.put()` and `replace()` invocation.

Assuming that we have previously created Person objects containing attributes such as name, age, and gender,

## Well-known Attributes

the following example shows how to create a dynamically searchable cache and register the `DynamicAttributesExtractor`:

```
Configuration config = new Configuration();
config.setName("default");
CacheConfiguration cacheCfg = new CacheConfiguration("- PersonCache-");
cacheCfg.setEternal(true);
Searchable searchable = new Searchable().allowDynamicIndexing(true);

cacheCfg.addSearchable(searchable);
config.addCache(cacheCfg);

CacheManager cm = new CacheManager(config);
Ehcache cache = cm.getCache("- PersonCache-");
final String attrNames[] = {"first_name", "age"};
// Now you can register a dynamic attribute extractor that would index the cache elements, using
cache.registerDynamicAttributesExtractor(new DynamicAttributesExtractor() {
    Map<String, Object> attributesFor(Element element) {
        Map<String, Object> attrs = new HashMap<String, Object>();
        Person value = (Person)element.getObjectValue();
        // For example, extract first name only
        String fName = value.getName() == null ? null : value.getName().split("- \\s+")[0];
        attrs.put(attrNames[0], fName);
        attrs.put(attrNames[1], value.getAge());
        return attrs;
    }
});
// Now add some data to the cache
cache.put(new Element(10, new Person("- John Doe-", 34, Person.Gender.MALE)));


Given the code above, the newly put element would be indexed on values of name and age fields, but not gender. If, at a later time, you would like to start indexing the element data on gender, you would need to create a new DynamicAttributesExtractor instance that extracts that field for indexing.
```

## Dynamic Search Rules

- In order to use the `DynamicAttributesExtractor`, the cache must be configured to be searchable and dynamically indexable (refer to [Making a Cache Searchable](#) above).
- A dynamically searchable cache must have a dynamic extractor registered before data is added to it. (This is to prevent potential races between extractor registration and cache loading which might result in an incomplete set of indexed data, leading to erroneous search results.)
- Each call on the `DynamicAttributesExtractor` method replaces the previously registered extractor, as there can be at most one extractor instance configured for each such cache.
- If a dynamically searchable cache is initially configured with a predefined set of search attributes, then this set of attributes will always be queried for extracted values, regardless of whether or not there is a dynamic search attribute extractor configured.
- The initial search configuration takes precedence over dynamic attributes, so if the dynamic attribute extractor returns an attribute name already used in the initial searchable configuration, an exception will be thrown.

## Creating a Query

BigMemory Go Search uses a fluent, object-oriented Query API, following DSL principles, which should be familiar to Java programmers. Here is a simple example:

## Creating a Query

```
Query query = cache.createQuery().addCriteria(age.eq(35)).includeKeys().end();
Results results = query.execute();
```

## Using Attributes in Queries

If declared and available, the well-known attributes are referenced by their names or the convenience attributes are used directly, as shown in this example:

```
Results results = cache.createQuery().addCriteria(Query.KEY.eq(35)).execute();
Results results = cache.createQuery().addCriteria(Query.VALUE.lt(10)).execute();
```

Other attributes are referenced by the names given them in the configuration. For example:

```
Attribute<Integer> age = cache.getSearchAttribute("age");
Attribute<String> gender = cache.getSearchAttribute("gender");
Attribute<String> name = cache.getSearchAttribute("name");
```

## Expressions

A Query is built up using Expressions. Expressions may include logical operators such as <and> and <or>, and comparison operators such as <ge> ( $\geq$ ), <between>, and <like>. The configuration addCriteria(...) is used to add a clause to a query. Adding a further clause automatically "<and>s" the clauses.

```
query = cache.createQuery().includeKeys().addCriteria(age.le(65)).add(gender.eq("male")).end();
```

Both logical and comparison operators implement the Criteria interface. To add a criteria with a different logical operator, explicitly nest it within a new logical operator Criteria Object. For example, to check for age = 35 or gender = female, do the following:

```
query.addCriteria(new Or(age.eq(35),
    gender.eq(Gender.FEMALE))
);
```

More complex compound expressions can be further created with extra nesting. See the [Expression JavaDoc](#) for a complete list of expressions.

## List of Operators

Operators are available as methods on attributes, so they are used by adding a ". ". For example, "lt" means "less than" and is used as age.lt(10), which is a shorthand way of saying age LessThan(10). The full listing of operator shorthand is shown below.

| Shorthand | Criteria Class     | Description  |
|-----------|--------------------|--|
| and       | And                | The Boolean AND logical operator                                     |
| between   | Between            | A comparison operator meaning between two values                     |
| eq        | EqualTo            | A comparison operator meaning Java "equals to" condition             |
| gt        | GreaterThan        | A comparison operator meaning greater than.                          |
| ge        | GreaterThanOrEqual | A comparison operator meaning greater than or equal to.              |
| in        | InCollection       | A comparison operator meaning in the collection given as an argument |

## List of Operators

|       |                 |   |
|-------|-----------------|---|
| lt    | LessThan        | A comparison operator meaning less than.  |
| le    | LessThanOrEqual | A comparison operator meaning less than or equal to   |
| ilike | ILike           | A regular expression matcher. '?' and '*' may be used. Note that placing a wildcard in front of the expression will cause a table scan. ILike is always case insensitive. |
| not   | Not             | The Boolean NOT logical operator  |
| ne    | NotEqualTo      | A comparison operator meaning not the Java "equals to" condition  |
| or    | Or              | The Boolean OR logical operator   |

## Making Queries Immutable

By default, a query can be executed and then modified and re-executed. If `end` is called, the query is made immutable.

## Obtaining and Organizing Query Results

Queries return a `Results` object which contains a list of objects of class `Result`. Each `Element` in the cache found with a query will be represented as a `Result` object. So if a query finds 350 elements, there will be 350 `Result` objects. An exception to this would be if no keys or attributes are included but aggregators are -- in this case, there will be exactly one `Result` present.

A `Result` object can contain:

- the `Element` key - when `includeKeys()` is added to the query,
- the `Element` value - when `includeValues()` is added to the query,
- predefined attribute(s) extracted from an `Element` value - when `includeAttribute(...)` is added to the query. To access an attribute from a `Result`, use `getAttribute(Attribute<T> attribute)`.
- aggregator results - Aggregator results are summaries computed for the search. They are available through `Result.getAggregatorResults` which returns a list of `Aggregators` in the same order in which they were used in the `Query`.

## Aggregators

Aggregators are added with `query.includeAggregator(\<attribute\>.\<aggregator\>)`. For example, to find the sum of the age attribute:

```
query.includeAggregator(age.sum());
```

For a complete list of aggregators, refer to the [Aggregators JavaDoc](#).

## Ordering Results

Query results may be ordered in ascending or descending order by adding an `addOrderBy` clause to the query, which takes as parameters the attribute to order by and the ordering direction. For example, to order the results by ages in ascending order:

```
query.addOrderBy(age, Direction.ASCENDING);
```

## Grouping Results

BigMemory Go query results may be grouped similarly to using an SQL GROUP BY statement. The BigMemory GroupBy feature provides the option to group results according to specified attributes by adding an addGroupBy clause to the query, which takes as parameters the attributes to group by. For example, you can group results by department and location like this:

```
Query q = cache.createQuery();
Attribute<String> dept = cache.getSearchAttribute("dept");
Attribute<String> loc = cache.getSearchAttribute("location");
q.includeAttribute(dept);
q.includeAttribute(loc);
q.addCriteria(cache.getSearchAttribute("salary").gt(100000));
q.includeAggregator(Aggregators.count());
q.addGroupBy(dept, loc);
```

The GroupBy clause groups the results from `includeAttribute()` and allows aggregate functions to be performed on the grouped attributes. To retrieve the attributes that are associated with the aggregator results, you can use:

```
String dept = singleResult.getAttribute(dept);
String loc = singleResult.getAttribute(loc);
```

### GroupBy Rules

Grouping query results adds another step to the query--first results are returned, and second the results are grouped. This necessitates the following rules and considerations when using GroupBy:

- In a query with a GroupBy clause, any attribute specified using `includeAttribute()` should also be included in the GroupBy clause.
- Special KEY or VALUE attributes may not be used in a GroupBy clause. This means that `includeKeys()` and `includeValues()` may not be used in a query that has a GroupBy clause.
- Adding a GroupBy clause to a query changes the semantics of any aggregators passed in, so that they apply only within each group.
- As long as there is at least one aggregation function specified in a query, the grouped attributes are not required to be included in the result set, but they are typically requested anyway to make result processing easier.
- An `addCriteria()` clause applies to all results prior to grouping.
- If `OrderBy` is used with GroupBy, the ordering attributes are limited to those listed in the GroupBy clause.

## Limiting the Size of Results

By default a query will return an unlimited number of results. For example the following query will return all keys in the cache.

```
Query query = cache.createQuery();
query.includeKeys();
query.execute();
```

If too many results are returned, it could cause an OutOfMemoryError. The `maxResults` clause is used to limit the size of the results. For example, to limit the above query to the first 100 elements found:

## Limits the Size of Results

```
Query query = cache.createQuery();
query.includeKeys();
query.maxResults(100);
query.execute();
```

**Note:** When maxResults is used with GroupBy, it limits the number of groups.

When you are done with the results, call `discard()` to free up resources. In the distributed implementation with Terracotta, resources may be used to hold results for paging or return.

## Interrogating Results

To determine what was returned by a query, use one of the interrogation methods on `Results`:

- `hasKeys()`
- `hasValues()`
- `hasAttributes()`
- `hasAggregators()`

## Finding Null (or Not Null) Values

You can find null values (or not-null values) by replacing the null value with the string "NULL" in every element with a null value. A [custom attribute extractor](#) can be used to search for the string "NULL" to find null values (or use a not-equal search for not-null values).

If a string cannot be used for element values, then for each field (that may be a search target) set up a related dummy field and assign it a value of "0" (null) or "1" (not null). Queries can then check the dummy field to find nulls or not-nulls.

## Sample Application

We have created a [simple standalone sample application](#) with few dependencies for you to easily get started with BigMemory Search. You can also check out the source:

```
git clone git://github.com/sharrissf/Ehcache-Search-Sample.git
```

The [Ehcache Test Sources](#) page has further examples on how to use each Search feature.

## Scripting Environments

BigMemory Search is readily amenable to scripting. The following example shows how to use it with BeanShell:

```
Interpreter i = new Interpreter();
//Auto discover the search attributes and add them to the interpreter's context
Map<String, SearchAttribute> attributes = cache.getCacheConfiguration().getSearchAttributes();
for (Map.Entry<String, SearchAttribute> entry : attributes.entrySet()) {
    i.set(entry.getKey(), cache.getSearchAttribute(entry.getKey()));
    LOG.info("Setting attribute " + entry.getKey());
}
//Define the query and results. Add things which would be set in the GUI i.e.
//includeKeys and add to context
```

## Scripting Environments

```
Query query = cache.createQuery().includeKeys();
Results results = null;
i.set("query", query);
i.set("results", results);
//This comes from the freeform text field
String userDefinedQuery = "age.eq(35)";
//Add on the things that we need
String fullQueryString = "results = query.addCriteria(" + userDefinedQuery + ").execute()";
i.eval(fullQueryString);
results = (Results) i.get("results");
assertTrue(2 == results.size());
for (Result result : results.all()) {
LOG.info("") + result.getKey());
}
```

## Implementation and Performance

BigMemory uses a Search index that is maintained at the local node. The index is stored under a directory in the DiskStore and is available whether or not persistence is enabled. Any overflow from the on-heap tier of the cache, whether to the off-heap tier or to the disk tier, is searched using indexes.

Search operations perform in O(log(n)) time. For tips that can aid performance, refer to [Best Practices](#).

For caches that are on-heap only, the implementation does not use indexes. Instead, it performs a fast iteration of the cache, relying on the very fast access to do the equivalent of a table scan for each query. Each element in the cache is only visited once. Attributes are not extracted ahead of time. They are done during query execution.

Search operations perform in O(n) time. Check out this [Maven-based performance test](#) showing performance of an on-heap-only search. The test shows search performance of an average of representative queries at 4.6 ms for a 10,000 entry cache, and 427 ms for a 1,000,000 entry cache. Accordingly, this implementation is suitable for development and testing.

## Best Practices for Optimizing Searches

1. Construct searches wisely by including only the data that is actually required.
  - ◆ Only use `includeKeys()` and/or `includeAttribute()` if those values are actually required for your application logic.
  - ◆ If you don't need values or attributes, be careful not to burden your queries with unnecessary work. For example, if `result.getValue()` is not called in the search results, then don't use `includeValues()` in the original query.
  - ◆ Consider if it would be sufficient to get attributes or keys on demand. For example, instead of running a search query with `includeValues()` and then `result.getValue()`, run the query for keys and include `cache.get()` for each individual key.

**Note:** `includeKeys()` and `includeValues()` have lazy deserialization, which means that keys and values are de-serialized only when `result.getKey()` or `result.getValue()` is called. This means there is a time cost only when the key is needed. However, there is still some time cost with `includeKeys()` and `includeValues()`, so consider carefully when constructing your queries.

2. Searchable keys and values are automatically indexed by default. If you will not be including them in your query, turn off automatic indexing with the following:

## Best Practices for Optimizing Searches

```
<cache name="cacheName" ...>
  <searchable keys="false" values="false"/>
  ...
  </searchable>
</cache>
```

3. Limit the size of the results set with `query.maxResults(int number_of_results)`. Another recommendation for managing the size of the result set is to use a built-in Aggregator function to return a summary statistic (see the `net.sf.ehcache.search.aggregator` package in this [Javadoc](#)).
4. Make your search as specific as possible. Queries with "ILike" criteria and fuzzy (wildcard) searches may take longer than more specific queries. Also, if you are using a wildcard, try making it the trailing part of the string instead of the leading part ("321\*" instead of "\*123"). If you want leading wildcard searches, then you should create a `<searchAttribute>` with the string value reversed in it, so that your query can use the trailing wildcard instead.
5. When possible, use the query criteria "Between" instead of "LessThan" and "GreaterThan", or "LessThanOrEqual" and "GreaterThanOrEqual". For example, instead of using `le(startDate)` and `ge(endDate)`, try not (`between(startDate, endDate)`).
6. Index dates as integers. This can save time and may even be faster if you have to do a conversion later on.

## Concurrency Notes

Unlike cache operations which have selectable concurrency control and/or transactions, queries are asynchronous and Search results are eventually consistent with the caches.

### Index Updating

Although indexes are updated synchronously, their state will lag slightly behind the state of the cache. The only exception is when the updating thread then performs a search.

For caches with concurrency control, an index will not reflect the new state of the cache until a `commit` has been called.

### Query Results

There are several ways unexpected results could present:

- A search returns an Element reference which no longer exists.
- Search criteria select an Element, but the Element has been updated and a new Search would no longer match the Element.
- Aggregators, such as `sum()`, might disagree with the same calculation done by redoing the calculation yourself by re-accessing the cache for each key and repeating the calculation.
- Because the cache is always updated before the search index, it is possible that a value reference may refer to a value that has been removed from the cache. If this happens, the value will be null but the key and attributes which were supplied by the now stale cache index will be non-null. Because values in Ehcache are also allowed to be null, you cannot tell whether your value is null because it has been removed from the cache since the index was last updated or because it is a null value.

## Concurrency Notes

### Recommendations

Because the state of the cache can change between search executions, the following is recommended:

- Add all of the aggregators you want for a query at once, so that the returned aggregators are consistent.
- Use null guards when accessing a cache with a key returned from a search.

# Transactions in Ehcache

## Introduction

BigMemory Go supports Global Transactions, with "xa\_strict" and "xa" modes, and Local Transactions with "local" mode.

### All or nothing

If a cache is enabled for transactions, all operations on it must happen within a transaction context otherwise a `TransactionException` will be thrown.

### Transactional Methods

The following methods require a transactional context to run:

- `put()`
- `get()`
- `getQuiet()`
- `remove()`
- `getKeys()`
- `getSize()`
- `containsKey()`
- `removeAll()`
- `putWithWriter()`
- `removeWithWriter()`
- `putIfAbsent()`
- `removeElement()`
- `replace()`

This list applies to all [transactional modes](#).

All other methods work non-transactionally but can be called on a transactional cache, either within or outside of a transactional context.

### Change Visibility

The isolation level offered in BigMemory's Ehcache is `READ_COMMITTED`. Ehcache can work as an `XAResource`, in which case, full two-phase commit is supported. Specifically:

- All mutating changes to the cache are transactional including `put`, `remove`, `putWithWriter`, `removeWithWriter` and `removeAll`.
- Mutating changes are not visible to other transactions in the local JVM or across the cluster until `COMMIT` has been called.
- Until `COMMIT` has been called, reads such as by `cache.get(...)` by other transactions return the old copy. Reads do not block.

When to use transactional modes

## When to use transactional modes

Transactional modes are a powerful extension of Ehcache allowing you to perform atomic operations on your caches and other data stores.

- "local" — When you want your changes across multiple caches to be performed atomically. Use this mode when you need to update your caches atomically; that is, you can have all your changes be committed or rolled back using a straightforward API. This mode is most useful when a cache contains data calculated from other cached data..
- "xa" — Use this mode when you cache data from other data stores, such as a DBMS or JMS, and want to do it in an atomic way under the control of the JTA API ("Java Transaction API") but without the overhead of full two-phase commit. In this mode, your cached data can get out of sync with the other resources participating in the transactions in case of a crash. Therefore, only use this mode if you can afford to live with stale data for a brief period of time.
- "xa\_strict" — Similar to "xa" but use it only if you need strict XA disaster recovery guarantees. In this mode, the cached data can never get out of sync with the other resources participating in the transactions, even in case of a crash. However, to get that extra safety the performance decreases significantly.

## Requirements

The objects you are going to store in your transactional cache must:

- implement `java.io.Serializable` — This is required to store cached objects when the cache is clustered with Terracotta but it is also required by the copy-on-read / copy-on-write mechanism used to implement isolation.
- override `equals` and `hashCode` — Those must be overridden because the transactional stores rely on element value comparison. See `ElementValueComparator` and the `elementValueComparator` configuration setting.

## Configuration

Transactions are enabled on a cache-by-cache basis with the `transactionalMode` cache attribute. The allowed values are:

- "xa\_strict"
- "xa"
- "local"
- 'off'

The default value is "off". Enabling a cache for "xa\_strict" transactions is shown in the following example:

```
<cache name="xaCache"
    maxEntriesLocalHeap="500"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    diskExpiryThreadIntervalSeconds="1"
    transactionalMode="xa_strict">
</cache>
```

## Transactional Caches with Spring

Note the following when using Spring:

- If you access the cache from an @Transactional Spring-annotated method, then begin/commit/rollback statements are not required in application code because they are emitted by Spring.
- Both Spring and Ehcache need to access the transaction manager internally, and therefore you must inject your chosen transaction manager into Spring's PlatformTransactionManager as well as use an appropriate lookup strategy for Ehcache.
- The Ehcache default lookup strategy might not be able to detect your chosen transaction manager. For example, it cannot detect the WebSphere transaction manager (see [Transactions Managers](#)).
- Configuring a <tx:method> with read-only=true could be problematic with certain transaction managers such as WebSphere.

## Global Transactions

Global Transactions are supported by Ehcache. Ehcache can act as an {XAResource} to participate in JTA transactions under the control of a Transaction Manager. This is typically provided by your application server, however you can also use a third party transaction manager such as Bitronix. To use Global Transactions, select either "xa\_strict" or "xa" mode. The differences are explained in the sections below.

### Implementation

Global transactions support is implemented at the Store level, through XATransactionStore and JtaLocalTransactionStore. The former decorates the underlying MemoryStore implementation, augmenting it with transaction isolation and two-phase commit support through an <XAResource> implementation. The latter decorates a LocalTransactionStore-decorated cache to make it controllable by the standard JTA API instead of the proprietary TransactionController API. During its initialization, the Cache does a lookup the TransactionManager using the provided TransactionManagerLookup implementation. Then, using the TransactionManagerLookup.register(XAResource), the newly created XAResource is registered. The store is automatically configured to copy every Element read from the cache or written to it. Cache is copy-on-read and copy-on-write.

### Failure Recovery

In support of the JTA specification, only *prepared* transaction data is recoverable. Prepared data is persisted onto the cluster and locks on the memory are held. This means that non-clustered caches cannot persist transactions data, therefore recovery errors after a crash may be reported by the transaction manager.

### Recovery

At any time after something went wrong, an XAResource might be asked to recover. Data that has been prepared might either be committed or rolled back during recovery. In accordance with XA, data that has not yet been prepared is discarded. The recovery guarantee differs depending on the XA mode.

## Recovery

### **xa Mode**

With "xa", the cache doesn't get registered as an {XAResource} with the transaction manager but merely can follow the flow of a JTA transaction by registering a JTA {Synchronization}. The cache can end up inconsistent with the other resources if there is a JVM crash in the mutating node. In this mode, some inconsistency might occur between a cache and other XA resources (such as databases) after a crash. However, the cache data remains consistent because the transaction is still fully atomic on the cache itself.

### **xa\_strict Mode**

With "xa\_strict", the cache always responds to the TransactionManager's recover calls with the list of prepared XIDs of failed transactions. Those transaction branches can then be committed or rolled back by the transaction manager. This mode supports the basic XA mechanism of the JTA standard.

## Sample Apps

We have three sample applications showing how to use XA with a variety of technologies.

### **XA Sample App**

This sample application uses JBoss application server. It shows an example using User managed transactions. Although most people will use JTA from within Spring or EJB container rather than managing it themselves, this sample application is useful as a demonstration.

The following snippet from our SimpleTX servlet shows a complete transaction.

```
Ehcace cache = cacheManager.getEhcace("xaCache");
UserTransaction ut = getUserTransaction();
printLine(servletResponse, "Hello...");
try {
    ut.begin();
    int index = serviceWithinTx(servletResponse, cache);
    printLine(servletResponse, "Bye #" + index);
    ut.commit();
} catch(Exception e) {
    printLine(servletResponse,
        "Caught a " + e.getClass() + "! Rolling Tx back");
    if(!printStackTrace) {
        PrintWriter s = servletResponse.getWriter();
        e.printStackTrace(s);
        s.flush();
    }
    rollbackTransaction(ut);
}
```

The source code for the demo can be checked out from the [Terracotta Forge](#). A README.txt explains how to get the sample app going.

### **XA Banking Application**

This application is to show a real world scenario. A Web app reads <account transfer> messages from a queue and tries to execute these account transfers. With JTA turned on, failures are rolled back so that the cached account balance is always the same as the true balance summed from the database. This app is a Spring-based

## XA Banking Application

Java web app running in a Jetty container. It has (embedded) the following components:

- A message broker (ActiveMQ)
- 2 databases (embedded Derby XA instances)
- 2 caches (transactional Ehcache)

All XA Resources are managed by Atomikos TransactionManager. Transaction demarcation is done using Spring AOP's @Transactional annotation. You can run it with: mvn clean jetty:run. Then point your browser at: <http://localhost:9080>. To see what happens without XA transactions: mvn clean jetty:run -Dxa=no

The source code for the demo can be checked out from the [Terracotta Forge](#). A README.txt explains how to get the sample app going.

## Transaction Managers

### Automatically Detected Transaction Managers

Ehcache automatically detects and uses the following transaction managers in the following order:

- GenericJNDI (e.g. Glassfish, JBoss, JTOM and any others that register themselves in JNDI at the standard location of java:/TransactionManager)
- Weblogic (since 2.4.0)
- Bitronix
- Atomikos

No configuration is required; they work out of the box. The first found is used.

### Configuring a Transaction Manager

If your Transaction Manager is not in the above list or you wish to change the priority, provide your own

lookup class based on an implementation of

`net.sf.ehcache.transaction.manager.TransactionManagerLookup` and specify it in place of the `DefaultTransactionManagerLookup` in `ehcache.xml`:

```
<transactionManagerLookup  
    class= "com.mycompany.transaction.manager.MyTransactionManagerLookupClass"  
    properties="" propertySeparator=":"/>
```

Another option is to provide a different location for the JNDI lookup by passing the `jndiName` property to the `DefaultTransactionManagerLookup`. The example below provides the proper location for the `TransactionManager` in GlassFish v3:

```
<transactionManagerLookup  
    class="net.sf.ehcache.transaction.manager.DefaultTransactionManagerLookup"  
    properties="jndiName=java:appserver/TransactionManager" propertySeparator=";" />
```

## Local Transactions

## Local Transactions

Local Transactions allow single phase commit across multiple cache operations, across one or more caches, and in the same CacheManager. This lets you apply multiple changes to a CacheManager all in your own transaction. If you also want to apply changes to other resources, such as a database, then you need to open a transaction to them and manually handle commit and rollback to ensure consistency. Local transactions are not controlled by a Transaction Manager. Instead there is an explicit API where a reference is obtained to a TransactionController for the CacheManager using `cacheManager.getTransactionController()` and the steps in the transaction are called explicitly. The steps in a local transaction are:

- `transactionController.begin()` - This marks the beginning of the local transaction on the current thread. The changes are not visible to other threads or to other transactions.
- `transactionController.commit()` - Commits work done in the current transaction on the calling thread.
- `transactionController.rollback()` - Rolls back work done in the current transaction on the calling thread. The changes done since begin are not applied to the cache. These steps should be placed in a try-catch block which catches `TransactionException`. If any exceptions are thrown, `rollback()` should be called. Local Transactions has its own exceptions that can be thrown, which are all subclasses of `CacheException`. They are:
  - `TransactionException` - a general exception
  - `TransactionInterruptedException` - if `Thread.interrupt()` was called while the cache was processing a transaction.
  - `TransactionTimeoutException` - if a cache operation or commit is called after the transaction timeout has elapsed.

## Introduction Video

Ludovic Orban, the primary author of Local Transactions, presents an [introductory video](#) on Local Transactions.

## Configuration

Local transactions are configured as follows:

```
<cache name="sampleCache"
      ...
      transactionalMode="local"
  </cache>
```

## Isolation Level

As with the other transaction modes, the isolation level is `READ_COMMITTED`.

## Transaction Timeouts

If a transaction cannot complete within the timeout period, then a `TransactionTimeoutException` will be thrown. To return the cache to a consistent state you need to call `transactionController.rollback()`. Because `TransactionController` is at the level of the CacheManager, a default timeout can be set which applies to all transactions across all caches in a CacheManager. If not set, the default is 15 seconds. To change the defaultTimeout:

```
transactionController.setDefaultTransactionTimeout(int defaultTransactionTimeoutSeconds)
```

## Transaction Timeouts

The countdown starts when `begin()` is called. You might have another local transaction on a JDBC connection and you might be making multiple changes. If you think it could take longer than 15 seconds for an individual transaction, you can override the default when you begin the transaction with:

```
transactionController.begin(int transactionTimeoutSeconds) { ... }
```

## Sample Code

This example shows a transaction which performs multiple operations across two caches.

```
CacheManager cacheManager = CacheManager.getInstance();
try {
    cacheManager.getTransactionController().begin();
    cache1.put(new Element(1, "one"));
    cache2.put(new Element(2, "two"));
    cache1.remove(4);
    cacheManager.getTransactionController().commit();
} catch (CacheException e) {
    cacheManager.getTransactionController().rollback()
}
```

## Performance

### Managing Contention

If two transactions, either standalone or across the cluster, attempt to perform a cache operation on the same element then the following rules apply:

- The first transaction gets access
- The following transactions block on the cache operation until either the first transaction completes or the transaction timeout occurs.

Note that when an element is involved in a transaction, it is replaced with a new element with a marker that is locked, along with the transaction ID. The normal cluster semantics are used. Because transactions only work with consistency=strong caches, the first transaction is the thread that manages to atomically place a soft lock on the Element. (This is done with the CAS based `putIfAbsent` and `replace` methods.)

### What granularity of locking is used?

Ehcache uses soft locks stored in the Element itself and is on a key basis.

### Performance Comparisons

Any transactional cache adds an overhead which is significant for writes and nearly negligible for reads. Compared to `transactionalMode="off"`, the time it takes to perform writes will be noticeably slower with either "xa" or "local" specified, and "xa\_strict" will be the slowest.

Accordingly, "xa\_strict" should only be used where full guarantees are required, otherwise one of the other modes should be used.

## FAQ

### Why do some threads regularly time out and throw an exception?

In transactional caches, write locks are in force whenever an element is updated, deleted, or added. With concurrent access, these locks cause some threads to block and appear to deadlock. Eventually the deadlocked threads time out (and throw an exception) to avoid being stuck in a deadlock condition.

### Is IBM Websphere Transaction Manager supported?

Mostly. "xa\_strict" is not supported due to each version of Websphere being a custom implementation; that is, there is no stable interface to implement against. However, "xa", which uses TransactionManager callbacks and "local" are supported.

When using Spring, make sure your configuration is set up correctly with respect to the PlatformTransactionManager and the Websphere TM.

To confirm that Ehcache will succeed, try to manually register a `com.ibm.websphere.jtaextensions.SynchronizationCallback` in the `com.ibm.websphere.jtaextensions.ExtendedJTATransaction`. Simply get `java:comp/websphere/ExtendedJTATransaction` from JNDI, cast that to `com.ibm.websphere.jtaextensions.ExtendedJTATransaction` and call the `registerSynchronizationCallbackForCurrentTran` method. If you succeed, then Ehcache should too.

### How do transactions interact with Write-behind and Write-through caches?

If your transactional enabled cache is being used with a writer, write operations are queued until transaction commit time. Solely a Write-through approach would have its potential XAResource participate in the same transaction. Write-behind is supported, however it should probably not be used with an XA transactional Cache, as the operations would never be part of the same transaction. Your writer would also be responsible for obtaining a new transaction. Using Write-through with a non XA resource would also work, but there is no guarantee the transaction will succeed after the write operations have been executed. On the other hand, any thrown exception during these write operations would cause the transaction to be rolled back by having `UserTransaction.commit()` throw a `RollbackException`.

### Are Hibernate Transactions supported?

Ehcache is a "transactional" cache for Hibernate purposes. The `net.sf.ehcache.hibernate.EhCacheRegionFactory` supports Hibernate entities configured with `<cache usage="transactional"/>`.

### How do I make WebLogic 10 work with transactional Ehcache?

WebLogic uses an optimization that is not supported by our implementation. By default WebLogic 10 spawns threads to start the Transaction on each XAResource in parallel. Because we need transaction work to be performed on the same Thread, you must turn off this optimization by setting `parallel-xa-enabled` option to `false` in your domain configuration:

## How do I make WebLogic 10 work with transactional Ehcache?

```
<jta>
  ...
  <checkpoint-interval-seconds>300</checkpoint-interval-seconds>
  <parallel-xa-enabled>false</parallel-xa-enabled>
  <unregister-resource-grace-period>30</unregister-resource-grace-period>
...
</jta>
```

## How do I make Atomikos work with the Ehcache "xa" mode?

Atomikos has [a bug](#) which makes the "xa" mode's normal transaction termination mechanism unreliable. There is an alternative termination mechanism built in that transaction mode that is automatically enabled when `net.sf.ehcache.transaction.xa.alternativeTerminationMode` is set to true or when Atomikos is detected as the controlling transaction manager. This alternative termination mode has strict requirement on the way threads are used by the transaction manager and Atomikos's default settings won't work. You have to configure the following property to make it work:

```
com.atomikos.icatch.threaded_2pc=false
```

# Explicit Locking

## Introduction

BigMemory Go's Ehcache contains an implementation which provides for explicit locking, using Read and Write locks. With explicit locking, it is possible to get more control over Ehcache's locking behaviour to allow business logic to apply an atomic change with guaranteed ordering across one or more keys in one or more caches. It can therefore be used as a custom alternative to XA Transactions or Local transactions.

With that power comes a caution. It is possible to create deadlocks in your own business logic using this API.

## The API

The following methods are available on Cache and Ehcache.

```
/**  
 * Acquires the proper read lock for a given cache key  
 *  
 * @param key - The key that retrieves a value that you want to protect via locking  
 */  
public void acquireReadLockOnKey(Object key) {  
    this.acquireLockOnKey(key, LockType.READ);  
}  
/**  
 * Acquires the proper write lock for a given cache key  
 *  
 * @param key - The key that retrieves a value that you want to protect via locking  
 */  
public void acquireWriteLockOnKey(Object key) {  
    this.acquireLockOnKey(key, LockType.WRITE);  
}  
/**  
 * Try to get a read lock on a given key. If can't get it in timeout millis then  
 * return a boolean telling that it didn't get the lock  
 *  
 * @param key - The key that retrieves a value that you want to protect via locking  
 * @param timeout - millis until giveup on getting the lock  
 * @return whether the lock was awarded  
 * @throws InterruptedException  
 */  
public boolean tryReadLockOnKey(Object key, long timeout) throws InterruptedException {  
    Sync s = getLockForKey(key);  
    return s.tryLock(LockType.READ, timeout);  
}  
/**  
 * Try to get a write lock on a given key. If can't get it in timeout millis then  
 * return a boolean telling that it didn't get the lock  
 *  
 * @param key - The key that retrieves a value that you want to protect via locking  
 * @param timeout - millis until giveup on getting the lock  
 * @return whether the lock was awarded  
 * @throws InterruptedException  
 */  
public boolean tryWriteLockOnKey(Object key, long timeout) throws InterruptedException {  
    Sync s = getLockForKey(key);  
    return s.tryLock(LockType.WRITE, timeout);  
}
```

## The API

```
/**  
 * Release a held read lock for the passed in key  
 *  
 * @param key - The key that retrieves a value that you want to protect via locking  
 */  
public void releaseReadLockOnKey(Object key) {  
    releaseLockOnKey(key, LockType.READ);  
}  
/**  
 * Release a held write lock for the passed in key  
 *  
 * @param key - The key that retrieves a value that you want to protect via locking  
 */  
public void releaseWriteLockOnKey(Object key) {  
    releaseLockOnKey(key, LockType.WRITE);  
}  
/**  
 * Returns true if a read lock for the key is held by the current thread  
 *  
 * @param key  
 * @return true if a read lock for the key is held by the current thread  
 */  
boolean isReadLockedByCurrentThread(Object key);  
/**  
 * Returns true if a write lock for the key is held by the current thread  
 *  
 * @param key  
 * @return true if a write lock for the key is held by the current thread  
 */  
boolean isWriteLockedByCurrentThread(Object key);
```

## Example

Here is a brief example:

```
String key = "123";  
Foo val = new Foo();  
cache.acquireWriteLockOnKey(key);  
try {  
    cache.put(new Element(key, val));  
} finally {  
    cache.releaseWriteLockOnKey(key);  
}  
...sometime later  
String key = "123";  
cache.acquireWriteLockOnKey(key);  
try {  
    Object cachedVal = cache.get(key).getValue();  
    cachedVal.setSomething("abc");  
    cache.put(new Element(key, cachedVal));  
} finally {  
    cache.releaseWriteLockOnKey(key);  
}
```

## How it works

A READ lock does not prevent other READers from also acquiring a READ lock and reading. A READ lock cannot be obtained if there is an outstanding WRITE lock - it will queue. A WRITE lock cannot be obtained

## How it works

while there are outstanding READ locks - it will queue. In each case the lock should be released after use to avoid locking problems. The lock release should be in a `finally` block. If before each read you acquire a READ lock and then before each write you acquire a WRITE lock, then an isolation level akin to READ\_COMMITTED is achieved.

# Blocking Cache and Self-Populating Cache

## Introduction

The `net.sf.ehcache.constructs` package contains some applied caching classes which use the core classes to solve everyday caching problems. Two of these are `BlockingCache` and `SelfPopulatingCache`.

## Blocking Cache

Imagine you have a very busy web site with thousands of concurrent users. Rather than being evenly distributed in what they do, they tend to gravitate to popular pages. These pages are not static, they have dynamic data which goes stale in a few minutes. Or imagine you have collections of data which go stale in a few minutes. In each case the data is extremely expensive to calculate. Let's say each request thread asks for the same thing. That is a lot of work. Now, add a cache. Get each thread to check the cache; if the data is not there, go and get it and put it in the cache.

Now, imagine that there are so many users contending for the same data that in the time it takes the first user to request the data and put it in the cache, 10 other users have done the same thing. The upstream system, whether a JSP or velocity page, or interactions with a service layer or database are doing ten times more work than they need to. Enter the `BlockingCache`. It is blocking because all threads requesting the same key wait for the first thread to complete. Once the first thread has completed the other threads simply obtain the cache entry and return. The `BlockingCache` can scale up to very busy systems. Each thread can either wait indefinitely, or you can specify a timeout using the `timeoutMillis` constructor argument.

For more information about Blocking Cache, refer to this [Javadoc](#).

## SelfPopulatingCache

You want to use the `BlockingCache`, but the requirement to always release the lock creates gnarly code. You also want to think about what you are doing without thinking about the caching. Enter the `SelfPopulatingCache`. The name `SelfPopulatingCache` is synonymous with Pull-through cache, which is a common caching term. `SelfPopulatingCache` though always is in addition to a `BlockingCache`. `SelfPopulatingCache` uses a `CacheEntryFactory`, that given a key, knows how to populate the entry. Note: JCache inspired `getWithLoader` and `getAllWithLoader` directly in `Ehcache` which work with a `CacheLoader` may be used as an alternative to `SelfPopulatingCache`.

For more information about Self-populating Cache, refer to this [Javadoc](#).

# Cache Decorators

## Introduction

BigMemory Go uses the Ehcache interface, of which Cache is an implementation. It is possible and encouraged to create Ehcache decorators that are backed by a Cache instance, implement Ehcache and provide extra functionality.

The Decorator pattern is one of the well known Gang of Four patterns.

Decorated caches are accessed from the CacheManager using CacheManager.getEhcache(String name). Note that, for backward compatibility, CacheManager.getCache(String name) has been retained. However only CacheManager.getEhcache(String name) returns the decorated cache.

## Creating a Decorator

### Programmatically

Cache decorators are created as follows:

```
BlockingCache newBlockingCache = new BlockingCache(cache);
```

The class must implement Ehcache.

### By Configuration

Cache decorators can be configured directly in ehcache.xml. The decorators will be created and added to the CacheManager.

It accepts the name of a concrete class that extends net.sf.ehcache.constructs.CacheDecoratorFactory

The properties will be parsed according to the delimiter (default is comma ',') and passed to the concrete factory's createDecoratedEhcache(Ehcache cache, Properties properties) method along with the reference to the owning cache.

It is configured as per the following example:

```
<cacheDecoratorFactory  
    class="com.company.SomethingCacheDecoratorFactory"  
    properties="property1=36 ..." />
```

Note that decorators can be configured against the defaultCache. This is very useful for frameworks like Hibernate that add caches based on the defaultCache.

## Adding decorated caches to the CacheManager

Having created a decorator programmatically it is generally useful to put it in a place where multiple threads may access it. Note that decorators created via configuration in ehcache.xml have already been added to the

## Adding decorated caches to the CacheManager

CacheManager.

### Using CacheManager.replaceCacheWithDecoratedCache()

A built-in way is to replace the Cache in CacheManager with the decorated one. This is achieved as in the following example:

```
cacheManager.replaceCacheWithDecoratedCache(cache, newBlockingCache);
```

The CacheManager {replaceCacheWithDecoratedCache} method requires that the decorated cache be built from the underlying cache from the same name.

Note that any overwritten Ehcache methods will take on new behaviours without casting, as per the normal rules of Java. Casting is only required for new methods that the decorator introduces.

Any calls to get the cache out of the CacheManager now return the decorated one.

A word of caution. This method should be called in an appropriately synchronized init style method before multiple threads attempt to use it. All threads must be referencing the same decorated cache. An example of a suitable init method is found in CachingFilter:

```
/**  
 * The cache holding the web pages. Ensure that all threads for a given cache name  
 * are using the same instance of this.  
 */  
private BlockingCache blockingCache;  
/**  
 * Initialises blockingCache to use  
 *  
 * @throws CacheException The most likely cause is that a cache has not been  
 *                         configured in Ehcache's configuration file ehcache.xml  
 *                         for the filter name  
 */  
public void doInit() throws CacheException {  
    synchronized (this.getClass()) {  
        if (blockingCache == null) {  
            final String cacheName = getCacheName();  
            Ehcache cache = getCacheManager().getEhcache(cacheName);  
            if (!(cache instanceof BlockingCache)) {  
                //decorate and substitute  
                BlockingCache newBlockingCache = new BlockingCache(cache);  
                getCacheManager().replaceCacheWithDecoratedCache(cache, newBlockingCache);  
            }  
            blockingCache = (BlockingCache) getCacheManager().getEhcache(getCacheName());  
        }  
    }  
}  
  
Ehcache blockingCache = singletonManager.getEhcache("sampleCache1");
```

The returned cache will exhibit the decorations.

### Using CacheManager.addDecoratedCache()

Sometimes you want to add a decorated cache but retain access to the underlying cache.

## Using CacheManager.addDecoratedCache()

The way to do this is to create a decorated cache and then call `cache.setName(new_name)` and then add it to CacheManager with `CacheManager.addDecoratedCache()`.

```
/**  
 * Adds a decorated {@link Ehcache} to the CacheManager. This method neither creates  
 * the memory/disk store nor initializes the cache. It only adds the cache reference  
 * to the map of caches held by this cacheManager.  
  
 * It is generally required that a decorated cache, once constructed, is made available * to other execution  
 * threads. The simplest way of doing this is to either add it to * the cacheManager with a different name or  
 * substitute the original cache with the * decorated one. *  
  
 * This method adds the decorated cache assuming it has a different name. If another * cache (decorated or  
 * not) with the same name already exists, it will throw * {@link ObjectExistsException}. For replacing existing  
 * cache with another decorated cache having same name, please use * {@link  
 #replaceCacheWithDecoratedCache(Ehcache, Ehcache)} *  
  
 * Note that any overridden Ehcache methods by the decorator will take on new * behaviours without casting.  
 Casting is only required for new methods that the * decorator introduces. For more information see the well  
 known Gang of Four * Decorator pattern. * * @param decoratedCache * @throws ObjectExistsException * if  
 another cache with the same name already exists. */ public void addDecoratedCache(Ehcache  
 decoratedCache) throws ObjectExistsException {
```

## Built-in Decorators

### BlockingCache

A blocking decorator for an Ehcache, backed by a {@link Ehcache}.

It allows concurrent read access to elements already in the cache. If the element is null, other reads will block until an element with the same key is put into the cache. This is useful for constructing read-through or self-populating caches. BlockingCache is used by CachingFilter.

### SelfPopulatingCache

A selfpopulating decorator for Ehcache that creates entries on demand.

Clients of the cache simply call it without needing knowledge of whether the entry exists in the cache. If null the entry is created. The cache is designed to be refreshed. Refreshes operate on the backing cache, and do not degrade performance of get calls.

SelfPopulatingCache extends BlockingCache. Multiple threads attempting to access a null element will block until the first thread completes. If refresh is being called the threads do not block - they return the stale data. This is very useful for engineering highly scalable systems.

## Caches with Exception Handling

These are decorated. See [Cache Exception Handlers](#) for full details.

# Event Listeners

## CacheManager Event Listeners

BigMemory Go's Ehcache implementation includes CacheManager event listeners. These listeners allow implementers to register callback methods that will be executed when a CacheManager event occurs. Cache listeners implement the `CacheManagerEventListener` interface. The events include:

- adding a Cache
- removing a Cache

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

## Configuration

One `CacheManagerEventListenerFactory` and hence one `CacheManagerEventListener` can be specified per CacheManager instance. The factory is configured as below:

```
<cacheManagerEventListenerFactory class="" properties="" />
```

The entry specifies a `CacheManagerEventListenerFactory` which will be used to create a `CacheManagerEventListener`, which is notified when Caches are added or removed from the CacheManager. The attributes of a `CacheManagerEventListenerFactory` are:

- `class` — a fully qualified factory class name.
- `properties` — comma-separated properties having meaning only to the factory.

Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing. If no class is specified, or there is no `cacheManagerEventListenerFactory` element, no listener is created. There is no default.

## Implementing a CacheManager Event Listener Factory and CacheManager Event Listener

`CacheManagerEventListenerFactory` is an abstract factory for creating `cacheManager` listeners. Implementers should provide their own concrete factory extending this abstract factory. It can then be configured in `ehcache.xml`. The factory class needs to be a concrete subclass of the abstract factory `CacheManagerEventListenerFactory`, which is reproduced below:

```
/**  
 * An abstract factory for creating {@link CacheManagerEventListener}s. Implementers should  
 * provide their own concrete factory extending this factory. It can then be configured in  
 * ehcache.xml  
 */  
public abstract class CacheManagerEventListenerFactory {  
    /**  
     * Create a CacheManagerEventListener  
     */
```

## Implementing a CacheManager Event Listener Factory and CacheManager Event Listener

```
* @param properties implementation specific properties. These are configured as comma-
* separated name value pairs in ehcache.xml. Properties may be null.
* @return a constructed CacheManagerEventListener
*/
public abstract CacheManagerEventListener
    createCacheManagerEventListener(Properties properties);
}
```

The factory creates a concrete implementation of CacheManagerEventListener, which is reproduced below:

```
/**
 * Allows implementers to register callback methods that will be executed when a
 * CacheManager event occurs.
 * The events include:
 *
 * adding a Cache
 * removing a Cache
 *
 *
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of
 * the implementer to safely handle the potential performance and thread safety issues
 * depending on what their listener is doing.
 */
public interface CacheManagerEventListener {
 /**
 * Called immediately after a cache has been added and activated.
 *
 * Note that the CacheManager calls this method from a synchronized method. Any attempt to
 * call a synchronized method on CacheManager from this method will cause a deadlock.
 *
 * Note that activation will also cause a CacheEventListener status change notification
 * from {@link net.sf.ehcache.Status#STATUS_UNINITIALISED} to
 * {@link net.sf.ehcache.Status#STATUS_ALIVE}. Care should be taken on processing that
 * notification because:
 * <ul>
 * <li>the cache will not yet be accessible from the CacheManager.
 * <li>the addCaches methods which cause this notification are synchronized on the
 * CacheManager. An attempt to call {@link net.sf.ehcache.CacheManager#getCache(String)}
 * will cause a deadlock.
 * </ul>
 * The calling method will block until this method returns.
 *
 * @param cacheName the name of the Cache the operation relates to
 * @see CacheEventListener
 */
void notifyCacheAdded(String cacheName);
 /**
 * Called immediately after a cache has been disposed and removed. The calling method will
 * block until this method returns.
 *
 * Note that the CacheManager calls this method from a synchronized method. Any attempt to
 * call a synchronized method on CacheManager from this method will cause a deadlock.
 *
 * Note that a {@link CacheEventListener} status changed will also be triggered. Any
 * attempt from that notification to access CacheManager will also result in a deadlock.
 * @param cacheName the name of the Cache the operation relates to
 */
void notifyCacheRemoved(String cacheName);
}
```

## Cache Event Listeners

The implementations need to be placed in the classpath accessible to Ehcache. Ehcache uses the ClassLoader returned by `Thread.currentThread().getContextClassLoader()` to load classes.

# Cache Event Listeners

BigMemory Go's Ehcache implementation also includes Cache event listeners. Cache listeners allow implementers to register callback methods that will be executed when a cache event occurs. Cache listeners implement the `CacheEventListener` interface. The events include:

- an Element has been put
- an Element has been updated. Updated means that an Element exists in the Cache with the same key as the Element being put.
- an Element has been removed
- an Element expires, either because `timeToLive` or `timeToIdle` have been reached.

Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing. Listeners are guaranteed to be notified of events in the order in which they occurred. Elements can be put or removed from a Cache without notifying listeners by using the `putQuiet` and `removeQuiet` methods.

## Configuration

Cache event listeners are configured per cache. Each cache can have multiple listeners. Each listener is configured by adding a `cacheEventListenerFactory` element as follows:

```
<cache ...>
  <cacheEventListenerFactory class="" properties="" listenFor="" />
  ...
</cache>
```

The entry specifies a `CacheEventListenerFactory` which is used to create a `CacheEventListener`, which then receives notifications. The attributes of a `CacheEventListenerFactory` are:

- `class` — a fully qualified factory class name.
- `properties` — optional comma-separated properties having meaning only to the factory.
- `listenFor` — describes which events will be delivered in a clustered environment (defaults to "all").

These are the possible values:

- ◆ "all" — the default is to deliver all local and remote events
- ◆ "local" — deliver only events originating in the current node
- ◆ "remote" — deliver only events originating in other nodes (for BigMemory Max only)

Callbacks to listener methods are synchronous and unsynchronized. It is the responsibility of the implementer to safely handle the potential performance and thread safety issues depending on what their listener is doing.

## Implementing a Cache Event Listener Factory and Cache Event Listener

# Implementing a Cache Event Listener Factory and Cache Event Listener

A CacheEventListenerFactory is an abstract factory for creating cache event listeners. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml. The following example demonstrates how to create an abstract CacheEventListenerFactory:

```
/**  
 * An abstract factory for creating listeners. Implementers should provide their own  
 * concrete factory extending this factory. It can then be configured in ehcache.xml  
 *  
 */  
public abstract class CacheEventListenerFactory {  
/**  
 * Create a CacheEventListener  
 *  
 * @param properties implementation specific properties. These are configured as comma  
 * separated name value pairs in ehcache.xml  
 * @return a constructed CacheEventListener  
 */  
public abstract CacheEventListener createCacheEventListener(Properties properties);  
}
```

The following example demonstrates how to create a concrete implementation of the CacheEventListener interface:

```
/**  
 * Allows implementers to register callback methods that will be executed when a cache event  
 * occurs.  
 * The events include:  
 * <ol>  
 * <li>put Element  
 * <li>update Element  
 * <li>remove Element  
 * <li>an Element expires, either because timeToLive or timeToIdle has been reached.  
 * </ol>  
 *  
 * Callbacks to these methods are synchronous and unsynchronized. It is the responsibility of  
 * the implementer to safely handle the potential performance and thread safety issues  
 * depending on what their listener is doing.  
 *  
 * Events are guaranteed to be notified in the order in which they occurred.  
 *  
 * Cache also has putQuiet and removeQuiet methods which do not notify listeners.  
 *  
 */  
public interface CacheEventListener extends Cloneable {  
/**  
 * Called immediately after an element has been removed. The remove method will block until  
 * this method returns.  
 *  
 * Ehcache does not check for  
 *  
 * As the {@link net.sf.ehcache.Element} has been removed, only what was the key of the  
 * element is known.  
 *  
 * @param cache the cache emitting the notification  
 * @param element just deleted
```

## Implementing a Cache Event Listener Factory and Cache Event Listener

```
/*
void notifyElementRemoved(final Ehcache cache, final Element element) throws CacheException;
/**
 * Called immediately after an element has been put into the cache. The
 * {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
 * will block until this method returns.
 *
 * Implementers may wish to have access to the Element's fields, including value, so the
 * element is provided. Implementers should be careful not to modify the element. The
 * effect of any modifications is undefined.
 *
 * @param cache the cache emitting the notification
 * @param element the element which was just put into the cache.
 */
void notifyElementPut(final Ehcache cache, final Element element) throws CacheException;
/**
 * Called immediately after an element has been put into the cache and the element already
 * existed in the cache. This is thus an update.
 *
 * The {@link net.sf.ehcache.Cache#put(net.sf.ehcache.Element)} method
 * will block until this method returns.
 *
 * Implementers may wish to have access to the Element's fields, including value, so the
 * element is provided. Implementers should be careful not to modify the element. The
 * effect of any modifications is undefined.
 *
 * @param cache the cache emitting the notification
 * @param element the element which was just put into the cache.
 */
void notifyElementUpdated(final Ehcache cache, final Element element) throws CacheException;
/**
 * Called immediately after an element is found to be expired. The
 * {@link net.sf.ehcache.Cache#remove(Object)} method will block until this method returns.
 *
 * As the {@link Element} has been expired, only what was the key of the element is known.
 *
 * Elements are checked for expiry in Ehcache at the following times:
 * <ul>
 * <li>When a get request is made
 * <li>When an element is spooled to the diskStore in accordance with a MemoryStore
 * eviction policy
 * <li>In the DiskStore when the expiry thread runs, which by default is
 * {@link net.sf.ehcache.Cache#DEFAULT_EXPIRY_THREAD_INTERVAL_SECONDS}
 * </ul>
 * If an element is found to be expired, it is deleted and this method is notified.
 *
 * @param cache the cache emitting the notification
 * @param element the element that has just expired
 *
 * Deadlock Warning: expiry will often come from the DiskStore
 * expiry thread. It holds a lock to the DiskStore at the time the
 * notification is sent. If the implementation of this method calls into a
 * synchronized Cache method and that subsequently calls into
 * DiskStore a deadlock will result. Accordingly implementers of this method
 * should not call back into Cache.
 */
void notifyElementExpired(final Ehcache cache, final Element element);
/**
 * Give the replicator a chance to cleanup and free resources when no longer needed
 */
void dispose();
/**
```

## Adding a Listener Programmatically

```
* Creates a clone of this listener. This method will only be called by Ehcache before a
* cache is initialized.
*
* This may not be possible for listeners after they have been initialized. Implementations
* should throw CloneNotSupportedException if they do not support clone.
* @return a clone
* @throws CloneNotSupportedException if the listener could not be cloned.
*/
public Object clone() throws CloneNotSupportedException;
}
```

Two other methods are also available:

- void notifyElementEvicted(Ehcache cache, Element element)

Called immediately after an element is evicted from the cache. Eviction, which happens when a cache entry is deleted from a store, should not be confused with removal, which is a result of calling Cache.removeElement(Element).

- void notifyRemoveAll(Ehcache cache)

Called during Ehcache.removeAll() to indicate that all elements have been removed from the cache in a bulk operation. The usual  
notifyElementRemoved(net.sf.ehcache.Ehcache, net.sf.ehcache.Element)  
is not called. Only one notification is emitted because performance considerations do not allow for serially processing notifications where potentially millions of elements have been bulk deleted.

The implementations need to be placed in the classpath accessible to Ehcache. See the page on [Classloading](#) for details on how the loading of these classes will be done.

## Adding a Listener Programmatically

To add a listener programmatically, follow this example:

```
cache.getCacheEventNotificationService().registerListener(myListener);
```

# Cache Exception Handlers

## Introduction

BigMemory Go's Ehcache implementation includes Cache exception handlers. By default, most cache operations will propagate a runtime CacheException on failure. An interceptor, using a dynamic proxy, may be configured so that a CacheExceptionHandler can be configured to intercept Exceptions. Errors are not intercepted.

Caches with ExceptionHandling configured are of type Ehcache. To get the exception handling behavior they must be referenced using CacheManager.getEhcache(), not CacheManager.getCache(), which returns the underlying undecorated cache.

CacheExceptionHandlers may be set either declaratively in the ehcache.xml configuration file, or programmatically.

## Declarative Configuration

Cache event listeners are configured per cache. Each cache can have at most one exception handler. An exception handler is configured by adding a `cacheExceptionHandlerFactory` element as shown in the following example:

```
<cache ...>
  <cacheExceptionHandlerFactory
    class="net.sf.ehcache.exceptionhandler.CountingExceptionHandlerFactory"
    properties="logLevel=FINE"/>
</cache>
```

## Implementing a Cache Exception Handler Factory and Cache Exception Handler

A `CacheExceptionHandlerFactory` is an abstract factory for creating cache exception handlers. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in ehcache.xml. The factory class needs to be a concrete subclass of the abstract factory class `CacheExceptionHandlerFactory`, which is reproduced below:

```
/**
 * An abstract factory for creating <code>CacheExceptionHandler</code>s at configuration
 * time, in ehcache.xml.
 * <p/>
 * Extend to create a concrete factory
 *
 */
public abstract class CacheExceptionHandlerFactory {
 /**
 * Create an <code>CacheExceptionHandler</code>
 *
 * @param properties implementation specific properties. These are configured as comma
 * separated name value pairs in ehcache.xml
 * @return a constructed CacheExceptionHandler
 */
 public abstract CacheExceptionHandler createExceptionHandler(Properties properties);
```

## Implementing a Cache Exception Handler Factory and Cache Exception Handler

```
}
```

The factory creates a concrete implementation of the `CacheExceptionHandler` interface, which is reproduced below:

```
/*
 * A handler which may be registered with an Ehcache, to handle exception on Cache operations.
 *
 * Handlers may be registered at configuration time in ehcache.xml, using a
 * CacheExceptionHandlerFactory, or * set at runtime (a strategy).
 *
 * If an exception handler is registered, the default behaviour of throwing the exception
 * will not occur. The handler * method on Exception will be called. Of course, if
 * the handler decides to throw the exception, it will * propagate up through the call stack.
 * If the handler does not, it won't.
 *
 * Some common Exceptions thrown, and which therefore should be considered when implementing
 * this class are listed below:
 * <ul>
 * <li>{@link IllegalStateException} if the cache is not
 * {@link net.sf.ehcache.Status#STATUS_ALIVE}
 * <li>{@link IllegalArgumentException} if an attempt is made to put a null
 * element into a cache
 * <li>{@link net.sf.ehcache.distribution.RemoteCacheException} if an issue occurs
 * in remote synchronous replication
 * </li>
 * </li>
 * </ul>
 *
 */
public interface CacheExceptionHandler {
 /**
 * Called if an Exception occurs in a Cache method. This method is not called
 * if an Error occurs.
 *
 * @param Ehcache    the cache in which the Exception occurred
 * @param key        the key used in the operation, or null if the operation does not use a
 * key or the key was null
 * @param exception the exception caught
 */
 void onException(Ehcache ehcache, Object key, Exception exception);
}
```

The implementations need to be placed in the classpath accessible to Ehcache. See the page on [Classloading](#) for details on how classloading of these classes will be done.

## Programmatic Configuration

The following example shows how to add exception handling to a cache, and then adding the cache back into cache manager so that all clients obtain the cache handling decoration.

```
CacheManager cacheManager = ...
Ehcache cache = cacheManager.getCache("exampleCache");
ExceptionHandler handler = new ExampleExceptionHandler(...);
cache.setCacheLoader(handler);
Ehcache proxiedCache = ExceptionHandlingDynamicCacheProxy.createProxy(cache);
cacheManager.replaceCacheWithDecoratedCache(cache, proxiedCache);
```

# Cache Extensions

## Introduction

BigMemory Go's Ehcache implementation includes Cache extensions. Cache extensions are a general-purpose mechanism to allow generic extensions to a Cache. Cache extensions are tied into the Cache lifecycle. For that reason, this interface has the lifecycle methods.

Cache extensions are created using the `CacheExtensionFactory`, which has a `createCacheCacheExtension()` method that takes as a parameter a Cache and properties. It can thus call back into any public method on Cache, including, of course, the load methods. Cache extensions are suitable for timing services, where you want to create a timer to perform cache operations. (Another way of adding Cache behavior is to decorate a cache. See [BlockingCache](#) for an example of how to do this.)

Because a `CacheExtension` holds a reference to a Cache, the `CacheExtension` can do things such as registering a `CacheEventListener` or even a `CacheManagerEventListener`, all from within a `CacheExtension`, creating more opportunities for customization.

## Declarative Configuration

Cache extensions are configured per cache. Each cache can have zero or more. A `CacheExtension` is configured by adding a `cacheExceptionHandlerFactory` element as shown in the following example:

```
<cache ...>
    <cacheExtensionFactory class="com.example.FileWatchingCacheRefresherExtensionFactory"
        properties="refreshIntervalMillis=18000, loaderTimeout=3000,
        flushPeriod=whatever, someOtherProperty=someValue ..."/>
</cache>
```

## Implementing a Cache Extension Factory and Cache Extension

A `CacheExtensionFactory` is an abstract factory for creating cache extension. Implementers should provide their own concrete factory, extending this abstract factory. It can then be configured in `ehcache.xml`. The factory class needs to be a concrete subclass of the abstract factory class `CacheExtensionFactory`, which is reproduced below:

```
/**
 * An abstract factory for creating CacheExtensions. Implementers should
 * provide their own * concrete factory extending this factory. It can then be configured
 * in ehcache.xml.
 *
 */
public abstract class CacheExtensionFactory {
    /**
     * @param cache the cache this extension should hold a reference to, and to whose
     * lifecycle it should be bound.
     * @param properties implementation specific properties configured as delimiter separated
     * name value pairs in ehcache.xml
     */
    public abstract CacheExtension createCacheExtension(Ehcache cache, Properties properties);
```

## Implementing a Cache Extension Factory and CacheExtension

```
}
```

The factory creates a concrete implementation of the CacheExtension interface, which is reproduced below:

```
/**  
 * This is a general purpose mechanism to allow generic extensions to a Cache.  
 *  
 * CacheExtensions are tied into the Cache lifecycle. For that reason this interface has the  
 * lifecycle methods.  
 *  
 * CacheExtensions are created using the CacheExtensionFactory which has a  
 * createCacheCacheExtension() method which takes as a parameter a Cache and  
 * properties. It can thus call back into any public method on Cache, including, of course,  
 * the load methods.  
 *  
 * CacheExtensions are suitable for timing services, where you want to create a timer to  
 * perform cache operations. The other way of adding Cache behaviour is to decorate a cache.  
 * See {@link net.sf.ehcache.constructs.blocking.BlockingCache} for an example of how to do  
 * this.  
 *  
 * Because a CacheExtension holds a reference to a Cache, the CacheExtension can do things  
 * such as registering a CacheEventListener or even a CacheManagerEventListener, all from  
 * within a CacheExtension, creating more opportunities for customisation.  
 */  
public interface CacheExtension {  
    /**  
     * Notifies providers to initialise themselves.  
     *  
     * This method is called during the Cache's initialise method after it has changed its  
     * status to alive. Cache operations are legal in this method.  
     *  
     * @throws CacheException  
     */  
    void init();  
    /**  
     * Providers may be doing all sorts of exotic things and need to be able to clean up on  
     * dispose.  
     *  
     * Cache operations are illegal when this method is called. The cache itself is partly  
     * disposed when this method is called.  
     *  
     * @throws CacheException  
     */  
    void dispose() throws CacheException;  
    /**  
     * Creates a clone of this extension. This method will only be called by Ehcache before a  
     * cache is initialized.  
     *  
     * Implementations should throw CloneNotSupportedException if they do not support clone  
     * but that will stop them from being used with defaultCache.  
     *  
     * @return a clone  
     * @throws CloneNotSupportedException if the extension could not be cloned.  
     */  
    public CacheExtension clone(Ehcache cache) throws CloneNotSupportedException;  
    /**  
     * @return the status of the extension  
     */  
    public Status getStatus();
```

## Programmatic Configuration

```
}
```

The implementations need to be placed in the classpath accessible to ehcache. See the page on [Classloading](#) for details on how class loading of these classes will be done.

# Programmatic Configuration

Cache extensions may also be programmatically added to a Cache as shown.

```
TestCacheExtension testCacheExtension = new TestCacheExtension(cache, ...);  
testCacheExtension.init();  
cache.registerCacheExtension(testCacheExtension);
```

# Cache Eviction Algorithms

## Introduction

A cache eviction algorithm is a way of deciding which element to evict when the cache is full. In BigMemory Go's Ehcache, the memory store and the off-heap store may be limited in size. (For information about sizing, refer to [Sizing BigMemory Tiers](#)). When these stores get full, elements are evicted. The eviction algorithms in Ehcache determine which elements are evicted. The default is LRU.

What happens on eviction depends on the cache configuration. If a disk store is configured, the evicted element is flushed to disk; otherwise it will be removed. The disk store size by default is unbounded. But a maximum size can be set (see [Cache Configuration Sizing Attributes](#) for more information). If the disk store is full, then adding an element will cause one to be evicted unless it is unbounded. The disk store eviction algorithm is not configurable. It uses LFU.

## Provided MemoryStore Eviction Algorithms

The idea here is, given a limit on the number of items to cache, how to choose the thing to evict that gives the *best* result.

In 1966 Laszlo Belady showed that the most efficient caching algorithm would be to always discard the information that will not be needed for the longest time in the future. This is a theoretical result that is unimplementable without domain knowledge. The Least Recently Used ("LRU") algorithm is often used as a proxy. It works pretty well because of the locality of reference phenomenon and is the default in most caches.

A variation of LRU is the default eviction algorithm in Ehcache.

Altogether Ehcache provides three eviction algorithms to choose from for the memory store.

### Least Recently Used (LRU)

This is the default and is a variation on Least Frequently Used.

The oldest element is the Less Recently Used (LRU) element. The last used timestamp is updated when an element is put into the cache or an element is retrieved from the cache with a get call.

This algorithm takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

If probabilistic eviction does not suit your application, a true Least Recently Used deterministic algorithm is available by setting `java -Dnet.sf.ehcache.use.classic.lru=true`.

### Least Frequently Used (LFU)

For each get call on the element, the number of hits is updated. When a put call is made for a new element (and assuming that the max limit is reached), the element with least number of hits (the Least Frequently Used element) is evicted.

## Least Frequently Used (LFU)

If cache element use follows a Pareto distribution, this algorithm may give better results than LRU.

LFU is an algorithm unique to Ehcache. It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

## First In First Out (FIFO)

Elements are evicted in the same order as they come in. When a put call is made for a new element (and assuming that the max limit is reached for the memory store) the element that was placed first (First-In) in the store is the candidate for eviction (First-Out).

This algorithm is used if the use of an element makes it less likely to be used in the future. An example here would be an authentication cache.

It takes a random sample of the Elements and evicts the smallest. Using the sample size of 15 elements, empirical testing shows that an Element in the lowest quartile of use is evicted 99% of the time.

## Plugging in your own Eviction Algorithm

BigMemory Go allows you to plugin in your own eviction algorithm. You can utilize any Element metadata which makes possible some very interesting approaches. For example, evict an Element if it has been hit more than 10 times.

```
/**  
 * Sets the eviction policy strategy. The Cache will use a policy at startup.  
 * There are three policies which can be configured: LRU, LFU and FIFO. However  
 * many other policies are possible. That the policy has access to the whole element  
 * enables policies based on the key, value, metadata, statistics, or a combination  
 * of any of the above.  
 *  
 * It is safe to change the policy of a store at any time. The new policy takes  
 * effect immediately.  
 *  
 * @param policy the new policy  
 */  
public void setMemoryStoreEvictionPolicy(Policy policy) {  
    memoryStore.setEvictionPolicy(policy);  
}
```

A Policy must implement the following interface:

```
public interface Policy {  
    /**  
     * @return the name of the Policy. Inbuilt examples are LRU, LFU and FIFO.  
     */  
    String getName();  
    /**  
     * Finds the best eviction candidate based on the sampled elements. What  
     * distinguishes this approach from the classic data structures approach is  
     * that an Element contains metadata (e.g. usage statistics) which can be used  
     * for making policy decisions, while generic data structures do not. It is  
     * expected that implementations will take advantage of that metadata.  
     *  
     * @param sampledElements this should be a random subset of the population
```

## Plugging in your own Eviction Algorithm

```
* @param justAdded we probably never want to select the element just added.  
* It is provided so that it can be ignored if selected. May be null.  
* @return the selected Element  
*/  
Element selectedBasedOnPolicy(Element[] sampledElements, Element justAdded);  
/**  
 * Compares the desirability for eviction of two elements  
 *  
 * @param element1 the element to compare against  
 * @param element2 the element to compare  
 * @return true if the second element is preferable for eviction to the first  
 * element under the policy  
 */  
boolean compare(Element element1, Element element2);  
}
```

## Disk Store Eviction Algorithm

The disk store uses the Least Frequently Used algorithm to evict an element when it is full.

# Class Loading and Class Loaders

## Introduction

Class loading, within the plethora of environments that Ehcache can be running, could be complex. But with BigMemory Go, all class loading is done in a standard way in one utility class: ClassLoaderUtil.

## Plugin Class Loading

Ehcache allows plugins for events and distribution. These are loaded and created as follows:

```
/***
 * Creates a new class instance. Logs errors along the way. Classes are loaded
 * using the Ehcache standard classloader.
 *
 * @param className a fully qualified class name
 * @return null if the instance cannot be loaded
 */
public static Object createNewInstance(String className) throws CacheException {

    Class clazz;
    Object newInstance;
    try {
        clazz = Class.forName(className, true, getStandardClassLoader());
    } catch (ClassNotFoundException e) {
        //try fallback
        try {
            clazz = Class.forName(className, true, getFallbackClassLoader());
        } catch (ClassNotFoundException ex) {
            throw new CacheException("Unable to load class " + className +
                    ". Initial cause was " + e.getMessage(), e);
        }
    }
    try {
        newInstance = clazz.newInstance();
    } catch (IllegalAccessException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    } catch (InstantiationException e) {
        throw new CacheException("Unable to load class " + className +
                ". Initial cause was " + e.getMessage(), e);
    }
    return newInstance;
}

/**
 * Gets the ClassLoader that all classes in ehcache, and extensions,
 * should use for classloading. All ClassLoading in Ehcache should use this one.
 * This is the only thing that seems to work for all of the class loading
 * situations found in the wild.
 * @return the thread context class loader.
 */
public static ClassLoader getStandardClassLoader() {
    return Thread.currentThread().getContextClassLoader();
}

/***
```

## Plugin Class Loading

```
* Gets a fallback ClassLoader that all classes in ehcache, and
* extensions, should use for classloading. This is used if the context class loader
* does not work.
* @return the ClassLoaderUtil.class.getClassLoader();
*/
public static ClassLoader getFallbackClassLoader() {
    return ClassLoaderUtil.class.getClassLoader();
}
```

If this does not work for some reason, a CacheException is thrown with a detailed error message.

## Loading of ehcache.xml resources

If the configuration is otherwise unspecified, Ehcache looks for a configuration in the following order:

- Thread.currentThread().getContextClassLoader().getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache.xml")
- ConfigurationFactory.class.getResource("/ehcache-failsafe.xml")

Ehcache uses the first configuration found. Note the use of "/ehcache.xml" which requires that ehcache.xml be placed at the root of the classpath (i.e., not in any package).

# The Terracotta Management Console

The Terracotta Management Console (TMC) is a web-based administration and monitoring application providing a wealth of advantages, including:

- Multilevel security architecture, with end-to-end SSL secure connections available
- Feature-rich and easy-to-use in-browser interface
- Remote management capabilities requiring only a web browser and network connection
- Cross-platform deployment
- Role-based authentication
- Aggregate statistics from multiple nodes
- Flexible deployment model plugs into both development environments and secure production architectures

The TMC can monitor standalone Ehcache nodes, version 2.6 and higher. The Terracotta Management Server (TMS) acts as an aggregator and also provides a connection and security context for the TMC. The TMS must be available and accessible for the TMC to provide management services.

For more information on using the TMC, see the following sources:

- The README.txt file in the TMC kit for installation, configuration, basic security setup, and startup instructions.
- The on-board online help available from within the TMC for information on the user interface.
- The [security setup page](#) to learn how to use advanced security features.

**NOTE:** The Terracotta Management Console is now available for standalone enterprise Ehcache and replaces Ehcache Monitor for monitoring, management, and administration. For download instructions, contact pm@terracotta.org.

## Using Multiple Instances of BigMemory Go CacheManagers With the TMC

Note that when loading multiple instances BigMemory Go CacheManagers with the TMC rest agent enabled in the same JVM, they must be loaded by distinct classloaders. Two different web applications (two different WARs), for example, are loaded by different classloaders.

## The TMC Update Checker

The Update Checker automatically checks to see if you have the latest updates, and collects diagnostic information on TMC usage. The Update Checker is on by default. To disable the update checker, use the following system property:

```
-Dcom.terracotta.management.skipUpdateCheck=true
```

# Terracotta Management Console Security Setup

## Introduction

The Terracotta Management Console (TMC) includes a flexible, multi-level security architecture to integrate with a wide variety of contexts.

The following levels of security are available:

- No authentication, and no or limited secured connections. The non-enterprise version of the TMC is restricted to this mode, while enterprise versions can run in this mode by choice.
- Default role-based user authentication only. This is built in and set up when you first connect to an enterprise version of the TMC.
- Basic security offering authentication and authorization of clients (managed agents), as well as message hashing and other protective measures.
- Secured connections based on Secure Sockets Layer (SSL) technology, which includes certificate-based server authentication, can be used in conjunction with basic security.
- Certificate-based client authentication to enhance SSL-based security. In this case, no message hashing is used.

With the exception of certificate-based client authentication and message hashing, these different security layers can be used together to provide the overall level of security required by your environment. Except as noted below, security features are available only with an enterprise version of the TMC.

This document discusses security from the perspective of the TMC. However, the TMC and the Terracotta Management Server (TMS) function in the same security context, and by extension the security measures discussed here also apply to the TMS.

## No Security

Upon initial connection to an enterprise version of the TMC, the authentication setup page appears, where you can choose to run the TMC with authentication or without. Authentication can also be enabled/disabled in the TMC Settings panel once the TMC is running. If you enable authentication, all of the security features described in this document are available.

If you do not enable authentication, you will be directly connected to the TMC without being prompted for a login/password. Note that with authentication disabled, the TMC cannot connect to secured nodes.

If you are running the non-enterprise version of the TMC, no security features are available. You can, however, [force SSL connections](#) between browsers and the TMC.

## Default Security

Default security consists of the built-in role-based accounts that are used to log into the TMC. Note that connections between the TMC and managed agents such as Ehcache clients, remain unsecured. This level of security controls access to the TMC only, and is appropriate for environments where all components, including the TMC, managed agents, and any custom RIAs, are on protected networks. An internal network behind a firewall, where all access is trusted, is one example of such an environment.

## Default Security

Set up the role-based accounts using the account setup page, which appears when you first connect to the TMC with authentication enabled, or after you enable authentication in the TMC **Settings** dialog. You can set up two role-based accounts:

- Administrator – This account has read-write control over all aspects of the TMC, including accounts and connections.
- Operator – This read-only account can view statistics and other information about configured connections. This account cannot add or edit accounts or connections.

After you enter a username and password for each account, click **Setup**. You will be logged out, and can now log in using the credentials you have set up.

## Basic Connection Security

You can secure the connections between the TMS and managed agents using a built-in hash-based message authentication scheme. This level of security authenticates managed agents (providing client-server authentication without need of certificates) and ensures that communication between the TMS and managed agents is secure. Use this level of security in environments where the TMS may be exposed to unwanted connection attempts from rogue agents, or managed agents may come under attack from a rogue TMS.

To use basic security, follow these steps:

1. Have a Terracotta trial or full license file in the Terracotta root installation directory.
2. Use the EE version of the agent (`ehcache-ee-rest-agent-<version>.jar`).
3. Enable security (authentication via identity assertion) on the REST service by adding the "securityServiceLocation" attribute to the managementRESTService element in the managed agent's configuration. The following example is for Ehcache:

```
<ehcache ...>
...
<managementRESTService enabled="true"
    securityServiceLocation="http://localhost:9889/tmc/api/assertIdentity" />
...
</ehcache>
```

If `securityServiceLocation` is not set, the authentication feature is disabled. To enable, set its value to the location of the TMC installation, with `/tmc/api/assertIdentity` appended as in the example above.

4. Create a shared secret for the assertion of trust between the TMC and any security-enabled instances of the Ehcache REST service you wish it to connect to.

Using a command line, run the following script located in the TMC `bin` directory:

```
./add-tc-agent.sh <url>
```

where `<url>` is the URL of the agent. This value should correspond exactly to the URL you use within the TMC to connect to the given agent. For example:

```
./add-tc-agent.sh http://localhost:9888
```

Use `add-tc-agent.bat` with Microsoft Windows.

## Basic Connection Security

The script will prompt you for a shared secret of your choice. Be sure to note the secret that you enter, as you may need to enter it again in a later step.

5. If you will use the TMC to manage more than one agent, run the add-tc-agent script once for each agent, using that agent's URL. The script saves these entries to the file <user\_home>/.tc/mgmt/keychain. Do not move or delete this file—it must remain accessible to the TMC at that location.
6. Each agent with a keychain entry must also have a local copy of the keychain file saved to the same directory:

```
<user_home>/.tc/mgmt/keychain
```

To create separate keychain files for each agent, see [Creating Individual Keychain Files](#).

## Creating Individual Keychain Files

You may need to create additional keychain files for managed agents, if they run on different machines from the TMS. If you run the application with the agent on the same machine as the agent (for example, the local host), then you don't need to make an additional keychain file. The one you created for the TMS is sufficient.

To create an individual keychain file for a managed agent, use the keychain script provided in the kit's tmc/bin directory:

```
bin/keychain.sh -O -S -c /path/to/myKeychainFile http://myHost:9889/
```

where myKeychainFile is the name of the keychain file to create, and http://myHost:9889/ is the URI associated with the managed agent. This is the same URI configured in the TMC to connect to the managed agent. Use keychain.bat with Microsoft Windows.

The "/path/to/myKeychainFile" should be a temporary location on the machine running this script. (If it is on the machine with the TMS, be careful not to overwrite the keychain file that you created for the TMS.) On the machine running the agent, copy the keychain file to the directory:

```
<user_home>/.tc/mgmt/keychain
```

When you run the keychain script, the following prompt should appear:

```
Terracotta Management Console - Keychain Client
KeyChain file successfully created in /path/to/myKeychainFile
Open the keychain by entering its master key:
```

Enter the master key, then answer the prompts for the secret to be associated with the managed agent:

```
Enter the password you wish to associate with this URL:
Password for http://myHost:9889/ successfully stored
```

Note the acknowledgment upon successful generation of the keychain file. However, the script does not enforce the matching of the secret that you associate with the URI—this string-URI combination must match the one stored in the keychain file with the TMC.

## Adding SSL

In an environment where connections may be intercepted, or a higher level of authentication is required, adding SSL provides encryption and certificate-based server authentication. SSL can be used to enhance basic security.

To add SSL, follow these steps:

1. Enable SSL on the REST service by setting the managementRESTService element's "sslEnabled" attribute to "true" in the managed agent's configuration. The following example is for Ehcache:

```
<ehcache ...>
...
<managementRESTService enabled="true"
securityServiceLocation="http://localhost:9889/tmc/api/assertIdentity"
sslEnabled="true" />
...
</ehcache>
```

2. Provide an identity store for the managed agent either at the default location, \${user.home}/.tc/mgmt/keystore, or by setting the store's location with the system property javax.net.ssl.keyStore.

The identity store is where the server-authentication certificate is stored. If the identity store cannot be found, the managed agent fails at startup.

3. Add a password for the managed agent's identity store to its keychain.

The password must be keyed with the identity-store file's URI. Or set the password with the system property javax.net.ssl.keyStorePassword. If no password is found, the managed agent fails at startup.

4. The JVM running the TMC must have the same server-authentication certificate in one of the following locations:

- ◆ the default trust store for the JVM (typically the cacerts file)
- ◆ \${user.home}/.tc/mgmt/tms-truststore
- ◆ a location configured with the system property javax.net.ssl.trustStore

5. If a custom truststore (not cacerts) is designated for the TMC, the truststore password must be included in the TMC keychain.

The password must be keyed with the truststore file's URI. Or set the password with the system property javax.net.ssl.trustStorePassword.

## Certificate-Based Client Authentication

You can configure certificate-based client authentication environments where you are required to use that standard or simply to use the type of authentication and encryption offered by certificates. Configuring this option for managed agents disables hash-based message authentication, a [basic security](#) feature that is unneeded with certificate-based client authentication. Note that you must [configure SSL](#) to use this security option.

To enable certificate-based client authentication, follow these steps:

## Certificate-Based Client Authentication

1. Enable client authentication on the REST service by setting the managementRESTService element's "needClientAuth" attribute to "true" in the managed agent's configuration. The following example is for Ehcache:

```
<ehcache ...>
...
<managementRESTService enabled="true"
securityServiceLocation="http://localhost:9889/tmc/api/assertIdentity"
sslEnabled="true" needClientAuth="true" />
...
</ehcache>
```

2. Provide a truststore for the managed agent at the default location, \${user.home}/.tc/mgmt/truststore, or by setting the truststore location with the system property javax.net.ssl.trustStore.
3. The password for the truststore must be included in the managed agent's keychain.

The password must be keyed with the truststore file's URI. Or set the password with the system property javax.net.ssl.trustStorePassword.

4. Provide an identity store for the TMS at the default location, \${user.home}/.tc/mgmt/tms-keystore, or by setting the identity-store location with the system property javax.net.ssl.keyStore.

The managed agent will be rejected by the TMS unless a valid certificate is found.

5. The password for the TMS identity store must be included in the TMS keychain.

The password must be keyed with the identity-store file's URI. Or set the password with the system property javax.net.ssl.keyStorePassword.

6. To allow an SSL connection from the managed agent, an SSL connector must be configured. If the TMC is deployed with the provided Jetty web server, add the following to tmc/etc/jetty.xml as shown:

```
<Call name="addConnector">
<Arg>
<New class="org.eclipse.jetty.server.ssl.SslSelectChannelConnector">
<Arg>
<New class="org.eclipse.jetty.http.ssl.SslContextFactory">
<Set name="keyStore"/>/home/.tc/mgmt/tms-keystore</Set>
<Set name="keyStorePassword">OBF:1v9ulw1c1ym51xmq1rwd1rwh1xmkl1ym91w261v8s</Set>
<Set name="keyManagerPassword">OBF:1v9ulw1c1ym51xmq1rwd1rwh1xmkl1ym91w261v8s</Set>
<Set name="TrustStore"/>/home/.tc/mgmt/tms-truststore</Set>
<Set name="keyStorePassword">OBF:1v9ulw1c1ym51xmq1rwd1rwh1xmkl1ym91w261v8s</Set>
<Set name="needClientAuth">true</Set>
</New>
</Arg>
<Set name="port">9999</Set>
<Set name="maxIdleTime">30000</Set>
</New>
</Arg>
</Call>
```

Note the following about the configuration shown:

- ◆ If the TMS WAR is deployed with a different container, make the equivalent changes appropriate to that container.
- ◆ The SSL port must be free (unused by any another process) to avoid collisions.

## Forcing SSL connections For TMC Clients

- ◆ maxIdleTime can be set to a value that suits your environment.
- ◆ If the default keystore or truststore are not being used, enter the correct paths to the keystore and truststore being used.
- ◆ Passwords have been obfuscated using a built-in Jetty tool:

```
java -cp lib/jetty-runner.jar  
org.eclipse.jetty.util.security.Password myPassword
```

This command, which must be run from the TMC root directory, returns an obfuscated version of myPassword.

## Forcing SSL connections For TMC Clients

If the TMC is deployed with the provided Jetty web server, web browsers connecting to the TMC can use an unsecured connection (via HTTP port 9889). A secure SSL-based connection is also available (via HTTPS port 9443).

To force all web browsers to connect using SSL, disable the non-secure connector by commenting out that connector in tmc/etc/jetty.xml:

```
<!-- DISABLED non-secure connector  
<Call name="addConnector">  
  <Arg>  
    <New class="org.eclipse.jetty.server.nio.SelectChannelConnector">  
      <Set name="host"><Property name="jetty.host" /></Set>  
      <Set name="port"><Property name="jetty.port" default="9889"/></Set>  
      <Set name="forwarded">true</Set>  
      <Set name="maxIdleTime">300000</Set>  
      <Set name="Acceptors">2</Set>  
      <Set name="statsOn">false</Set>  
      <Set name="confidentialPort">8443</Set>  
      <Set name="lowResourcesConnections">20000</Set>  
      <Set name="lowResourcesMaxIdleTime">5000</Set>  
    </New>  
  </Arg>  
</Call>  
-->
```

If the TMC WAR is deployed with a different container, make the equivalent changes appropriate to that container.

## About the Default Keystore

By default, the built-in Jetty container's configuration file (tmc/etc/jetty.xml) uses a JKS identity store saved at the root directory (tmc). This keystore contains a self-signed certificate (not signed by a certificate authority). If you intend to use this "untrusted" certificate, all SSL browser connections must recognize this certificate and register it as an exception for future connections. This is usually done at the time the browser first connects to the TMS.

# Management and Monitoring using JMX

## Introduction

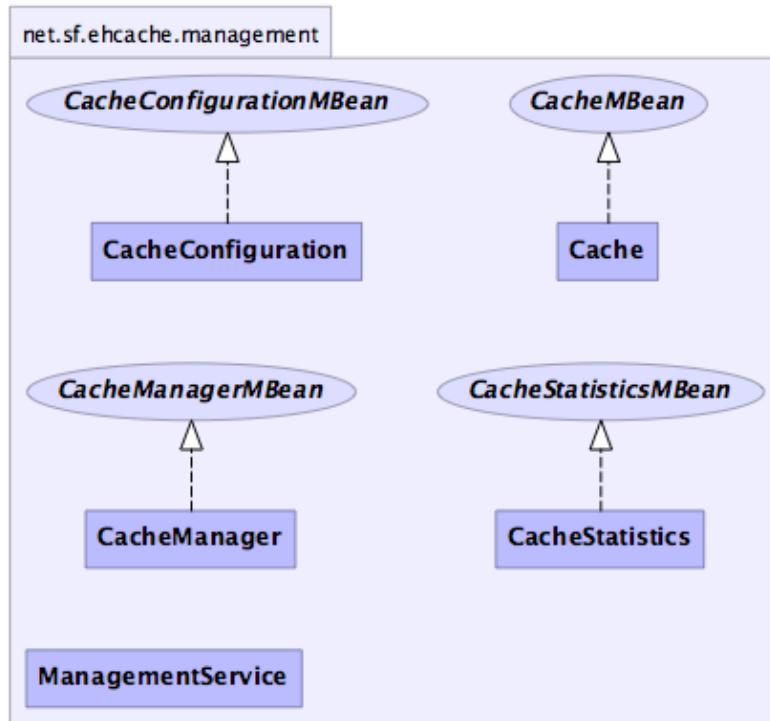
JMX creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure. This provides an alternative to the [Terracotta Management Console](#) when use of a custom or third-party tools.

## JMX Overview

The `net.sf.ehcache.management` package contains MBeans and a `ManagementService` for JMX management of BigMemory Go. It is in a separate package so that JMX libraries are only required if you wish to use it - there is no leakage of JMX dependencies into the core Ehcache package.

Use `net.sf.ehcache.management.ManagementService.registerMBeans(...)` static method to register a selection of MBeans to the MBeanServer provided to the method. If you wish to monitor Ehcache but not use JMX, just use the existing public methods on `Cache` and `CacheStatistics`.

The Management package is illustrated in the following image.



generated by yDoc

## MBeans

BigMemory Go uses Standard MBeans. MBeans are available for the following:

## MBeans

- CacheManager
- Cache
- CacheConfiguration
- CacheStatistics

All MBean attributes are available to a local MBeanServer. The CacheManager MBean allows traversal to its collection of Cache MBeans. Each Cache MBean likewise allows traversal to its CacheConfiguration MBean and its CacheStatistics MBean.

## JMX Remoting

The Remote API allows connection from a remote JMX Agent to an MBeanServer via an MBeanServerConnection. Only Serializable attributes are available remotely. The following Ehcache MBean attributes are available remotely:

- limited CacheManager attributes
- limited Cache attributes
- all CacheConfiguration attributes
- all CacheStatistics attributes

Most attributes use built-in types. To access all attributes, you need to add ehcache.jar to the remote JMX client's classpath. For example, jconsole -J-Djava.class.path=ehcache.jar.

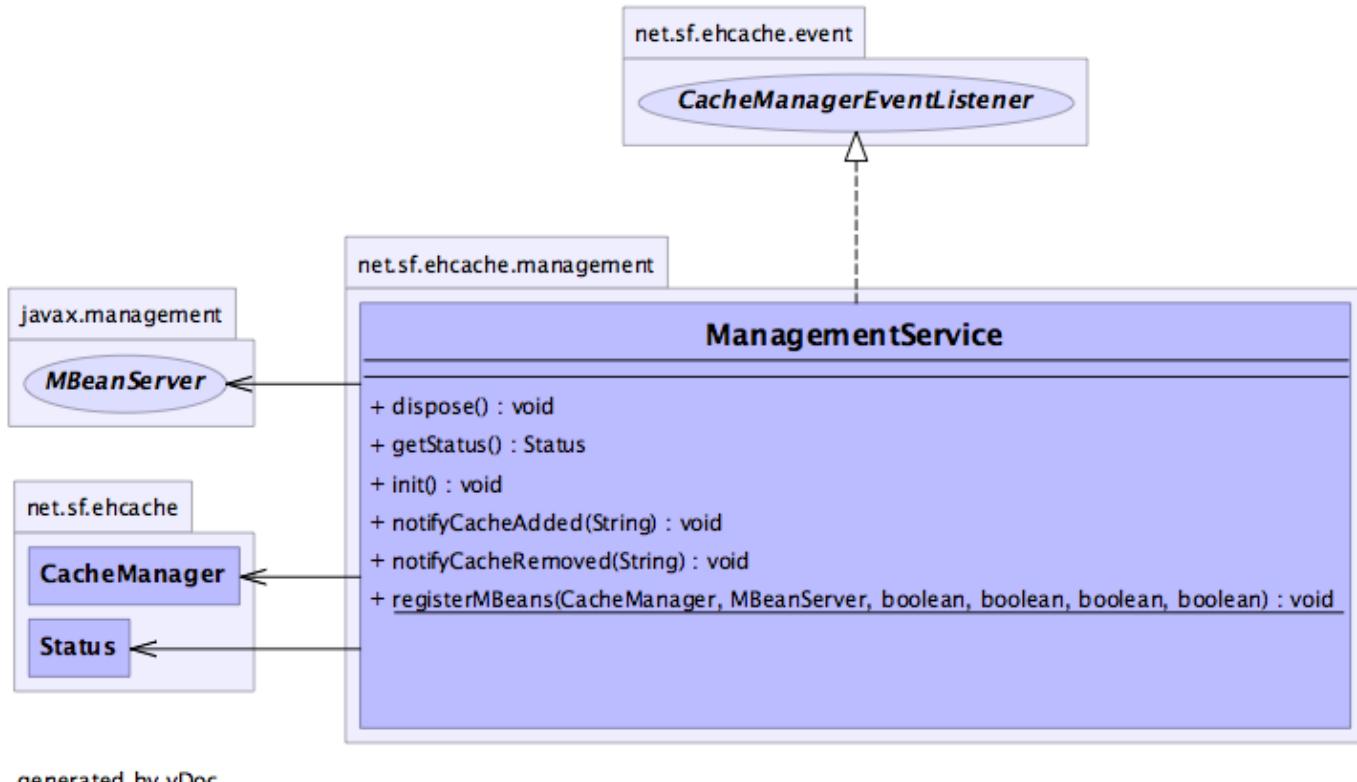
## ObjectName naming scheme

- CacheManager - "net.sf.ehcache:type=CacheManager,name=<CacheManager>"
- Cache - "net.sf.ehcache:type=Cache,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheConfiguration
- "net.sf.ehcache:type=CacheConfiguration,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheStatistics -  
"net.sf.ehcache:type=CacheStatistics,CacheManager=<cacheManagerName>,name=<cacheName>"

## The Management Service

The ManagementService class is the API entry point.

## The Management Service



generated by yDoc

There is only one method, `ManagementService.registerMBeans` which is used to initiate JMX registration of a `CacheManager`'s instrumented MBeans. The `ManagementService` is a `CacheManagerEventListener` and is therefore notified of any new Caches added or disposed and updates the `MBeanServer` appropriately. Once initiated the MBeans remain registered in the `MBeanServer` until the `CacheManager` shuts down, at which time the MBeans are deregistered. This behaviour ensures correct behaviour in application servers where applications are deployed and undeployed.

```

/**
 * This method causes the selected monitoring options to be registered
 * with the provided MBeanServer for caches in the given CacheManager.
 *
 * While registering the CacheManager enables traversal to all of the other
 * items, this requires programmatic traversal. The other options allow entry points closer
 * to an item of interest and are more accessible from JMX management tools like JConsole.
 * Moreover CacheManager and Cache are not serializable, so remote monitoring is not
 * possible for CacheManager or Cache, while CacheStatistics and CacheConfiguration are.
 * Finally CacheManager and Cache enable management operations to be performed.
 *
 * Once monitoring is enabled caches will automatically added and removed from the
 * MBeanServer as they are added and disposed of from the CacheManager. When the
 * CacheManager itself shutdowns all registered MBeans will be unregistered.
 *
 * @param cacheManager the CacheManager to listen to
 * @param mBeanServer the MBeanServer to register MBeans to
 * @param registerCacheManager Whether to register the CacheManager MBean
 * @param registerCaches Whether to register the Cache MBeans
 * @param registerCacheConfigurations Whether to register the CacheConfiguration MBeans
 * @param registerCacheStatistics Whether to register the CacheStatistics MBeans
 */
public static void registerMBeans(

```

## JConsole Example

```
net.sf.ehcache.CacheManager cacheManager,  
MBeanServer mBeanServer,  
boolean registerCacheManager,  
boolean registerCaches,  
boolean registerCacheConfigurations,  
boolean registerCacheStatistics) throws CacheException {
```

## JConsole Example

This example shows how to register CacheStatistics in the JDK platform MBeanServer, which works with the JConsole management agent.

```
CacheManager manager = new CacheManager();  
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();  
ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

CacheStatistics MBeans are then registered.

The screenshot shows the JConsole interface with the title "J2SE 5.0 Monitoring & Management Console: 3075@localhost". The "Connection" tab is selected. Below it, the "MBeans" tab is active. On the left, a tree view shows the following hierarchy under "net.sf.ehcache":

- JMImplementation
- java.lang
- java.util.logging
- net.sf.ehcache
  - CacheStatistics
    - net.sf.ehcache.CacheManager@881cb3
      - CachedLogin
      - FooterPageCache
      - SimplePageCachingFilter
      - SimplePageCachingFilterWithBlankPageProblem
      - SimplePageFragmentCachingFilter
      - net.sf.ehcache.constructs.asynchronous.MessageCache
      - persistentLongExpiryIntervalCache
      - sampleCache1
      - sampleCache2
      - sampleCacheNidle
      - sampleCacheNotEternalButNidleOrExpiry
      - sampleIdlingExpiringCache

CacheStatistics MBeans in JConsole

## Performance

Collection of cache statistics is not entirely free of overhead. In production systems where monitoring is not required, statistics can be disabled. This can be done either programmatically by calling `setStatisticsEnabled(false)` on the cache instance, or in configuration by setting the `statistics="false"` attribute of the relevant cache configuration element. Statistics are off by default.

# Logging

## Introduction

BigMemory Go uses the SLF4J logging facade, so you can plug in your own logging framework. This page covers Ehcache logging. For more information about slf4j in general, refer to the [SLF4J site](#).

## SLF4J Logging

With SLF4J, users must choose a concrete logging implementation at deploy time. The options include Maven and the download kit.

### Concrete Logging Implementation Use in Maven

The maven dependency declarations are reproduced here for convenience. Add *one* of these to your Maven POM.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-jdk14</artifactId>
    <version>1.5.8</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
    <version>1.5.8</version>
</dependency>
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-simple</artifactId>
    <version>1.5.8</version>
</dependency>
```

### Concrete Logging Implementation Use in the Download Kit

The slf4j-api and slf4j-jdk14 jars are in the kit along with the BigMemory Go jars so that, if the app does not already use SLF4J, you have everything you need. Additional concrete logging implementations can be downloaded from [SLF4J website](#).

## Recommended Logging Levels

BigMemory Go seeks to trade off informing production support developers or important messages and cluttering the log. ERROR messages should not occur in normal production and indicate that action should be taken.

WARN messages generally indicate a configuration change should be made or an unusual event has occurred. DEBUG and TRACE messages are for development use. All DEBUG level statements are surrounded with a guard so that no performance cost is incurred unless the logging level is set. Setting the logging level to DEBUG should provide more information on the source of any problems. Many logging systems enable a logging level change to be made without restarting the application.

# Shutting Down BigMemory

## Introduction

BigMemory is shut down through the Ehcache API. Note that Hibernate automatically shuts down its Ehcache CacheManager. The recommended way to shutdown the Ehcache is:

- to call CacheManager.shutdown()
- in a web app, register the Ehcache ShutdownListener

Though not recommended, Ehcache also lets you register a JVM shutdown hook.

## ServletContextListener

Ehcache provides a ServletContextListener that shuts down the CacheManager. Use this to shut down Ehcache automatically, when the web application is shut down. To receive notification events, this class must be configured in the deployment descriptor for the web application. To do so, add the following to web.xml in your web application:

```
<listener>
  <listener-class>net.sf.ehcache.constructs.web.ShutdownListener</listener-class>
</listener>
```

## The Shutdown Hook

The Ehcache CacheManager can optionally register a shutdown hook. To do so, set the system property net.sf.ehcache.enableShutdownHook=true. This will shut down the CacheManager when it detects the Virtual Machine shutting down and it is not already shut down.

Use the shutdown hook when the CacheManager is not already being shutdown by a framework you are using, or by your application.

**Note:** Shutdown hooks are inherently problematic. The JVM is shutting down, so sometimes things that can never be null are. Ehcache guards against as many of these as it can, but the shutdown hook should be the last option to use.

The shutdown hook is on CacheManager. It simply calls the shutdown method. The sequence of events is:

- call dispose for each registered CacheManager event listener.
- call dispose for each Cache. Each Cache will:
  - shutdown the MemoryStore. The MemoryStore will flush to the DiskStore.
  - shutdown the DiskStore. If the DiskStore is persistent ("localRestartable"), it will write the entries and index to disk.
- shutdown each registered CacheEventListener.
- set the Cache status to shutdown, preventing any further operations on it.
- set the CacheManager status to shutdown, preventing any further operations on it.

The shutdown hook runs when:

## The Shutdown Hook

- A program exists normally. For example, `System.exit()` is called, or the last non-daemon thread exits.
- the Virtual Machine is terminated, e.g. CTRL-C. This corresponds to `kill -SIGTERM pid` or `kill -15 pid` on Unix systems.

The shutdown hook will not run when:

- the Virtual Machine aborts.
- A `SIGKILL` signal is sent to the Virtual Machine process on Unix systems, e.g. `kill -SIGKILL pid` or `kill -9 pid`.
- A `TerminateProcess` call is sent to the process on Windows systems.

## Dirty Shutdown

If Ehcache is shutdown dirty, all in-memory data will be retained if BigMemory is configured for restartability. For more information, refer to [Fast Restartability](#).

# Using Hibernate and BigMemory Go

## Introduction

Big Memory Go easily integrates with the [Hibernate](#) Object/Relational persistence and query service. Gavin King, the maintainer of Hibernate, is also a committer to the BigMemory Go's Ehcache project. This ensures BigMemory Go will remain a first class data store for Hibernate. Configuring BigMemory Go for Hibernate is simple. The basic steps are:

- Download and install BigMemory Go into your project
- Configure BigMemory Go as a cache provider in your project's Hibernate configuration.
- Configure second-level caching in your project's Hibernate configuration.
- Configure Hibernate caching for each entity, collection, or query you wish to cache.
- Configure the ehcache.xml file as necessary for each entity, collection, or query configured for caching.

For more information regarding cache configuration in Hibernate see the [Hibernate](#) documentation.

## Download and Install

The Hibernate provider is in the ehcache-core module. Download [the latest version](#) of the Ehcache core module.

## Build with Maven

Dependency versions vary with the specific kit you intend to use. Since kits are guaranteed to contain compatible artifacts, find the artifact versions you need by downloading a kit. Configure or add the following repository to your build (pom.xml):

```
<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>
```

Configure or add the Ehcache core module defined by the following dependency to your build (pom.xml):

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache-core</artifactId>
  <version>${ehcacheVersion}</version>
</dependency>
```

For the Hibernate-Ehcache integration, add the following dependency:

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-ehcache</artifactId>
  <version>${hibernateVersion}</version>
</dependency>
```

## Build with Maven

For example, the Hibernate-Ehcache integration dependency for Hibernate 4.0.0 is:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-ehcache</artifactId>
    <version>4.0.0</version>
</dependency>
```

**NOTE:** Some versions of hibernate-ehcache may have a dependency on a specific version of Ehcache. Check the hibernate-ehcache POM for more information.

## Configure BigMemory Go as the Second-Level Cache Provider

To configure BigMemory Go as a Hibernate second-level cache, set the region factory property to one of the following in the Hibernate configuration. Hibernate configuration is configured either via hibernate.cfg.xml, hibernate.properties or Spring. The format given is for hibernate.cfg.xml.

### Hibernate 3.3

For instance creation:

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

To force Hibernate to use a singleton of Ehcache CacheManager:

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

### Hibernate 4.x

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory` and `org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory`.

## Enable Second-Level Cache and Query Cache Settings

In addition to configuring the second-level cache provider setting, you will need to turn on the second-level cache (by default it is configured to off - 'false' - by Hibernate). This is done by setting the following property in your hibernate config:

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

You may also want to turn on the Hibernate query cache. This is done by setting the following property in your hibernate config:

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

Optional

## Optional

The following settings or actions are optional.

### Configuration Resource Name

The configurationResourceName property is used to specify the location of the Ehcache configuration file to be used with the given Hibernate instance and cache provider/region-factory. The resource is searched for in the root of the classpath. It is used to support multiple CacheManagers in the same VM. It tells Hibernate which configuration to use. An example might be "ehcache-2.xml". When using multiple Hibernate instances it is therefore recommended to use multiple non-singleton providers or region factories, each with a dedicated Ehcache configuration resource.

```
net.sf.ehcache.configurationResourceName=/name_of_ehcache.xml
```

### Set the Hibernate cache provider programmatically

The provider can also be set programmatically in Hibernate by adding necessary Hibernate property settings to the configuration before creating the SessionFactory:

```
Configuration.setProperty("hibernate.cache.region.factory_class",
                           "net.sf.ehcache.hibernate.EhCacheRegionFactory")
```

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory`.

## Putting it all together

If you are enabling both second-level caching and query caching, then your hibernate config file should contain the following:

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

An equivalent Spring configuration file would contain:

```
<prop key="hibernate.cache.use_second_level_cache">true</prop>
<prop key="hibernate.cache.use_query_cache">true</prop>
<prop key="hibernate.cache.region.factory_class">
  net.sf.ehcache.hibernate.EhCacheRegionFactory</prop>
```

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory`.

## Configure Hibernate Entities to use Second-Level Caching

In addition to configuring the Hibernate second-level cache provider, Hibernate must also be told to enable caching for entities, collections, and queries. For example, to enable cache entries for the domain object `com.somecompany.someproject.domain.Country` there would be a mapping file something like the following:

## Configure Hibernate Entities to use Second-Level Caching

```
<hibernate-mapping>
<class
name="com.somecompany.someproject.domain.Country"
table="ut_Countries"
dynamic-update="false"
dynamic-insert="false"
>
...
</class>
</hibernate-mapping>
```

To enable caching, add the following element.

```
<cache usage="read-write|nonstrict-read-write|read-only" />
```

For example:

```
<hibernate-mapping>
<class
name="com.somecompany.someproject.domain.Country"
table="ut_Countries"
dynamic-update="false"
dynamic-insert="false"
>
  <cache usage="read-write" />
...
</class>
</hibernate-mapping>
```

This can also be achieved using the @Cache annotation, e.g.

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

## Definition of the different cache strategies

### **read-only**

Caches data that is never updated.

### **nonstrict-read-write**

Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database. Configure your cache timeout accordingly!

### **read-write**

Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read", this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.

# Configure

Because the `ehcache.xml` file has a defaultCache, caches will always be created when required by Hibernate. However more control can be exerted by specifying a configuration per cache, based on its name. In particular, because Hibernate caches are populated from databases, there is potential for them to get very large. This can be controlled by capping their `maxEntriesLocalHeap` and specifying whether to swap to disk beyond that. Hibernate uses a specific convention for the naming of caches of Domain Objects, Collections, and Queries.

## Domain Objects

Hibernate creates caches named after the fully qualified name of Domain Objects. So, for example to create a cache for `com.somecompany.someproject.domain.Country` create a cache configuration entry similar to the following in `ehcache.xml`.

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <cache
    name="com.somecompany.someproject.domain.Country"
    maxEntriesLocalHeap="10000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    <persistence strategy="localTempSwap"/>
  />
</ehcache>
```

### Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Objects.

## Collections

Hibernate creates collection caches named after the fully qualified name of the Domain Object followed by `".` followed by the collection field name. For example, a `Country` domain object has a set of `advancedSearchFacilities`. The Hibernate doclet for the accessor looks like:

```
/**
 * Returns the advanced search facilities that should appear for this country.
 * @hibernate.set cascade="all" inverse="true"
 * @hibernate.collection-key column="COUNTRY_ID"
 * @hibernate.collection-one-to-many class="com.wotif.jaguar.domain.AdvancedSearchFacility"
 * @hibernate.cache usage="read-write"
 */
public Set getAdvancedSearchFacilities() {
  return advancedSearchFacilities;
}
```

You need an additional cache configured for the set. The `ehcache.xml` configuration looks like:

```
<?xml version="1.0" encoding="UTF-8"?>
<ehcache>
  <cache name="com.somecompany.someproject.domain.Country"
    maxEntriesLocalHeap="50"
```

## Collections

```
    eternal="false"
    timeToLiveSeconds="600"
    <persistence strategy="localTempSwap"/>
/>
<cache
name="com.somecompany.someproject.domain.Country.advancedSearchFacilities"
maxEntriesLocalHeap="450"
eternal="false"
timeToLiveSeconds="600"
<persistence strategy="localTempSwap"/>
/>
</ehcache>
```

## Hibernate CacheConcurrencyStrategy

read-write, nonstrict-read-write and read-only policies apply to Domain Object collections.

## Queries

Hibernate allows the caching of query results using two caches.

### StandardQueryCache

This cache is used if you use a query cache without setting a name. A typical ehcache.xml configuration is:

```
<cache
name="org.hibernate.cache.StandardQueryCache"
maxEntriesLocalHeap="5"
eternal="false"
timeToLiveSeconds="120"
<persistence strategy="localTempSwap"/>
/>
```

### UpdateTimestampsCache

Tracks the timestamps of the most recent updates to particular tables. It is important that the cache timeout of the underlying cache implementation be set to a higher value than the timeouts of any of the query caches. In fact, it is recommend that the the underlying cache not be configured for expiry at all. A typical ehcache.xml configuration is:

```
<cache
name="org.hibernate.cache.UpdateTimestampsCache"
maxEntriesLocalHeap="5000"
eternal="true"
<persistence strategy="localTempSwap"/>
/>
```

### Named Query Caches

In addition, a QueryCache can be given a specific name in Hibernate using `Query.setCacheRegion(String name)`. The name of the cache in ehcache.xml is then the name given in that method. The name can be whatever you want, but by convention you should use "query." followed by a descriptive name. E.g.

```
<cache name="query.AdministrativeAreasPerCountry"
maxEntriesLocalHeap="5"
eternal="false"
```

## Queries

```
timeToLiveSeconds="86400"  
<persistence strategy="localTempSwap"/>  
>
```

## Using Query Caches

For example, let's say we have a common query running against the Country Domain. Code to use a query cache follows:

```
public List getStreetTypes(final Country country) throws HibernateException {  
    final Session session = createSession();  
    try {  
        final Query query = session.createQuery(  
            "select st.id, st.name"  
            + " from StreetType st "  
            + " where st.country.id = :countryId "  
            + " order by st.sortOrder desc, st.name");  
        query.setLong("countryId", country.getId().longValue());  
        query.setCacheable(true);  
        query.setCacheRegion("query.StreetTypes");  
        return query.list();  
    } finally {  
        session.close();  
    }  
}
```

The `query.setCacheable(true)` line caches the query. The `query.setCacheRegion("query.StreetTypes")` line sets the name of the Query Cache. Alex Miller has a good article on the query cache [here](#).

## Hibernate CacheConcurrencyStrategy

None of read-write, nonstrict-read-write and read-only policies apply to Domain Objects. Cache policies are not configurable for query cache. They act like a non-locking read only cache.

## Demo App

We have demo application showing how to use the Hibernate CacheRegionFactory.

## Hibernate Tutorial

Check out from the [Terracotta Forge](#).

## Performance Tips

### Session.load

`Session.load` will always try to use the cache.

### Session.find and Query.find

## `Session.find` and `Query.find`

`Session.find` does not use the cache for the primary object. Hibernate will try to use the cache for any associated objects. `Session.find` does however cause the cache to be populated. `Query.find` works in exactly the same way. Use these where the chance of getting a cache hit is low.

## `Session.iterate` and `Query.iterate`

`Session.iterate` always uses the cache for the primary object and any associated objects. `Query.iterate` works in exactly the same way. Use these where the chance of getting a cache hit is high.

## FAQ

### If I'm using BigMemory Go with my app and with Hibernate for second-level caching, should I try to use the CacheManager created by Hibernate for my app's caches?

While you could share the resource file between the two CacheManagers, a clear separation between the two is recommended. Your app may have a different lifecycle than Hibernate, and in each case your CacheManager [Automatic Resource Control \(ARC\)](#) settings may need to be different.

### Should I use the provider in the Hibernate distribution or in BigMemory Go's Ehcache?

Since Hibernate 2.1, Hibernate has included an Ehcache CacheProvider. That provider is periodically synced up with the provider in the Ehcache Core distribution. New features are generally added in to the Ehcache Core provider and then the Hibernate one.

### What is the relationship between the Hibernate and Ehcache projects?

Gavin King and Greg Luck cooperated to create Ehcache and include it in Hibernate. Since 2009, Greg Luck has been a committer on the Hibernate project so as to ensure Ehcache remains a first-class 2nd level cache for Hibernate.

### Does BigMemory Go support the transactional strategy?

Yes. It was introduced in Ehcache 2.1.

### Why do certain caches sometimes get automatically cleared by Hibernate?

Whenever a `Query.executeUpdate()` is run, for example, Hibernate invalidates affected cache regions (those corresponding to affected database tables) to ensure that no data stale data is cached. This should also happen whenever stored procedures are executed.

For more information, see this [Hibernate bug report](#).

How are Hibernate Entities keyed?

## **How are Hibernate Entities keyed?**

Hibernate identifies cached Entities via an object id. This is normally the primary key of a database row.

## **Are compound keys supported?**

Yes.

## **I am getting this error message: An item was expired by the cache while it was locked. What is it?**

Soft locks are implemented by replacing a value with a special type that marks the element as locked, thus indicating to other threads to treat it differently to a normal element. This is used in the Hibernate Read/Write strategy to force fall-through to the database during the two-phase commit - since we don't know exactly what should be returned by the cache while the commit is in process (but the database does). If a soft-locked Element is evicted by the cache during the two-phase commit, then once the two-phase commit completes, the cache will fail to update (since the soft-locked Element was evicted) and the cache entry will be reloaded from the database on the next read of that object. This is obviously non-fatal, but could cause a small rise in database load.

So, in summary the Hibernate messages are not problematic. The underlying cause is the probabilistic evictor can theoretically evict recently loaded items. You can also use the deterministic evictor to avoid this problem. Specify the `java -Dnet.sf.ehcache.use.classic.lru=true` system property to turn on classic LRU which contains a deterministic evictor.

# Using Coldfusion and BigMemory Go

## Introduction

ColdFusion ships with BigMemory Go's Ehcache. The ColdFusion community has actively engaged with Ehcache and have put out lots of great blogs. Here are three to get you started. For a short introduction, see [Raymond Camden's blog](#). For more in-depth analysis, see [Rob Brooks-Bilson's nine-part Blog Series](#) or [14 days of ColdFusion caching](#), by Aaron West, covering a different topic each day.

## Example Integration

To integrate BigMemory Go with ColdFusion, first add the BigMemory Go jars to your web application lib directory.

The following code demonstrates how to call Ehcache from ColdFusion. It will cache a CF object in Ehcache and the set expiration time to 30 seconds. If you refresh the page many times within 30 seconds, you will see the data from cache. After 30 seconds, you will see a cache miss, then the code will generate a new object and put in cache again.

```
<CFOBJECT type="JAVA" class="net.sf.ehcache.CacheManager" name="cacheManager">
<cfset cache=cacheManager.getInstance().getCache ("MyBookCache")>
<cfset myBookElement=#cache.get ("myBook")#>
<cfif IsDefined("myBookElement")>
    <cfoutput>
        myBookElement: #myBookElement#<br />
    </cfoutput>
    <cfif IsStruct (myBookElement.getObjectValue ())>
        <strong>Cache Hit</strong><p/>
        <!-- Found the object from cache -->
        <cfset myBook = #myBookElement.getObjectValue ()#>
    </cfif>
</cfif>
<cfif IsDefined("myBook")>
<cfelse>
    <strong>Cache Miss</strong>
    <!-- object not found in cache, go ahead create it -->
    <cfset myBook = StructNew ()>
    <cfset a = StructInsert (myBook, "cacheTime", LSTimeFormat (Now (), 'hh:mm:ss tt'), 1)>
    <cfset a = StructInsert (myBook, "title", "EhCache Book", 1)>
    <cfset a = StructInsert (myBook, "author", "Greg Luck", 1)>
    <cfset a = StructInsert (myBook, "ISBN", "ABCD123456", 1)>
    <CFOBJECT type="JAVA" class="net.sf.ehcache.Element" name="myBookElement">
    <cfset myBookElement.init ("myBook", myBook)>
    <cfset cache.put (myBookElement)>
</cfif>
<cfoutput>
    Cache time: #myBook ["cacheTime"]#<br />
    Title: #myBook ["title"]#<br />
    Author: #myBook ["author"]#<br />
    ISBN: #myBook ["ISBN"]#
</cfoutput>
```

# Using Spring and BigMemory Go

## Introduction

BigMemory Go's Ehcache has had excellent Spring integration for years. Spring 3.1 includes an Ehcache implementation. See the [Spring 3.1 JavaDoc](#).

## Spring 3.1

Spring Framework 3.1 has a generic cache abstraction for transparently applying caching to Spring applications. It has caching support for classes and methods using two annotations:

### @Cacheable

Cache a method call. In the following example, the value is the return type, a Manual. The key is extracted from the ISBN argument using the id.

```
@Cacheable(value="manual", key="#isbn.id")
public Manual findManual(ISBN isbn, boolean checkWarehouse)
```

### @CacheEvict

Clears the cache when called.

```
@CacheEvict(value = "manuals", allEntries=true)
public void loadManuals(InputStream batch)
```

For an excellent blog post covering SpEL expressions, see  
<http://blog.springsource.com/2011/02/23/spring-3-1-m1-caching/>.

## Spring 2.5 - 3.1: Annotations For Spring

This open source, led by Eric Dalquist, predates the Spring 3.1 project. You can use it with earlier versions of Spring, or you can use it with 3.1.

### @Cacheable

As with Spring 3.1 it uses an @Cacheable annotation to cache a method. In this example calls to findMessage are stored in a cache named "messageCache". The values are of type Message. The id for each entry is the id argument given.

```
@Cacheable(cacheName = "messageCache")
public Message findMessage(long id)
```

### @TriggersRemove

And for cache invalidation, there is the @TriggersRemove annotation. In this example, cache.removeAll() is called after the method is invoked.

## @TriggersRemove

```
@TriggersRemove(cacheName = "messagesCache",
when = When.AFTER_METHOD_INVOCATION, removeAll = true)
public void addMessage(Message message)
```

See <http://blog.goyello.com/2010/07/29/quick-start-with-ehcache-annotations-for-spring/> for a blog post explaining its use and providing further links.

# The Annotations for Spring Project

To dynamically configure caching of method return values, use the [Ehcache Annotations for Spring project at code.google.com](#). This project will allow you to configure caching of method calls dynamically using just configuration. The way it works is that the parameter values of the method will be used as a composite key into the cache, caching the return value of the method.

For example, suppose you have a method: Dog getDog (String name).

Once caching is added to this method, all calls to the method will be cached using the "name" parameter as a key.

So, assume at time t0 the application calls this method with the name equal to "fido". Since "fido" doesn't exist, the method is allowed to run, generating the "fido" Dog object, and returning it. This object is then put into the cache using the key "fido".

Then assume at time t1 the application calls this method with the name equal to "spot". The same process is repeated, and the cache is now populated with the Dog object named "spot".

Finally, at time t2 the application again calls the method with the name "fido". Since "fido" exists in the cache, the "fido" Dog object is returned from the cache instead of calling the method.

To implement this in your application, follow these steps:

### Step 1:

Add the jars to your application as listed on the [Ehcache Annotations for Spring project site](#).

### Step 2:

Add the Annotation to methods you would like to cache. Lets assume you are using the Dog getDog(String name) method from above:

```
@Cacheable(name="getDog")
Dog getDog(String name)
{
    ....
}
```

### Step 3:

Configure Spring. You must add the following to your Spring configuration file in the beans declaration section:

```
<ehcache:annotation-driven cache-manager="ehCacheManager" />
```

## The Annotations for Spring Project

More details can be found at:

- [Ehcache Annotations for Spring project](#)
- [the project getting started page](#)
- [this blog](#)

# JSR107 Support

Information on BigMemory Go's Ehcache support of JSR107 is available on [github](#).

# BigMemory Go FAQ

The BigMemory Go Technical FAQ answers frequently asked questions on how to use BigMemory Go, integration with other products, and solving issues. Other resources for resolving issues include:

- [Release Notes](#) – Lists features and issues for specific versions of BigMemory Go and other Terracotta products.
- [Compatibility Information](#) – Includes tables on compatible versions of BigMemory Go and other Terracotta products, JVMs, and application servers.
- [Terracotta Forums](#) – If your question doesn't appear below, consider posting it on the Ehcache Forum.

## Getting Started

### Is BigMemory Go really free for 32GB?

Yes.

### What if I want to use BigMemory Go for more than 32GB?

It's easy. To expand your license with a software subscription for more capacity, [contact Terracotta](#).

### What if I need enterprise support for BigMemory Go?

Terracotta provides enterprise support for BigMemory Go as part of software subscription. To get enterprise support, [contact Terracotta](#).

### Can I use BigMemory Go in production with my free 32GB license?

Yes.

### Is there any limit on the number of servers or applications I can use with my free 32GB license for BigMemory Go?

No. Deploy BigMemory Go with as many applications and on as many servers as you like.

### What's the difference between BigMemory Go and BigMemory Max?

BigMemory Go is for in-memory data management on a single JVM (in-process) and comes with 32GB free. BigMemory Max is for distributed in-memory management across an array of servers. For more on Go vs. Max, see [BigMemory Overview](#).

## Configuration

### Where is the source code?

BigMemory Go is free to use, but it is not an open-source product. See the [Ehcache website](#) for an open-source caching project.

Where is the source code?

## Can you use more than one instance of BigMemory Go in a single JVM?

Yes. Create a CacheManager using `new CacheManager(...)` and keep hold of the reference. The singleton approach, accessible with the `getInstance(...)` method, is still available too. However, hundreds of caches can be supported with one CacheManager, so use separate CacheManagers where different configurations are needed. The Hibernate Provider has also been updated to support this behavior.

## What elements are mandatory in ehcache.xml?

See the file `ehcache.xsd` in the BigMemory Go kit for the latest information on required configuration elements.

## How is auto-versioning of elements handled?

Automatic element versioning works only with memory-store caches only. BigMemory Go does not use auto-versioning.

To enable auto-versioning, set the system property `net.sf.ehcache.element.version.auto` to true (it is false by default). Manual (user provided) versioning of cache elements is ignored when auto-versioning is in effect. Note that if this property is turned on for one of the ineligible caches, auto-versioning will silently fail.

## How do I get a memory-only store to persist to disk between JVM restarts?

BigMemory Go offers [fast, robust disk persistence](#) set through configuration.

## What is the recommended way to write to a database?

There are two patterns available: [write-through](#) and [write-behind](#) caching. In write-through caching, writes to the cache cause writes to an underlying resource. The cache acts as a facade to the underlying resource. With this pattern, it often makes sense to read through the cache too. Write-behind caching uses the same client API; however, the write happens asynchronously.

While file systems or a web-service clients can underlie the facade of a write-through cache, the most common underlying resource is a database.

## Can I use BigMemory Go as a memory store only?

Yes. Just set the persistence strategy (in the `<cache>` configuration element) to "none":

```
<cache>
  ...
  <persistence strategy="none"/>
  ...
</cache>
```

## Can I use BigMemory Go as a disk store only?

No. However, you can minimize the usage of memory using [sizing configuration](#).

Where is the source code?

## Is it thread-safe to modify element values after retrieval from a store?

Remember that a value in an element is globally accessible from multiple threads. It is inherently not thread-safe to modify the value. It is safer to retrieve a value, delete the element and then reinsert the value.

The [UpdatingCacheEntryFactory](#) does work by modifying the contents of values in place in the cache. This is outside of the core of BigMemory Go and is targeted at high performance CacheEntryFactories for SelfPopulatingCaches.

## Can non-serializable objects be stored?

Non-serializable object can be stored only in the BigMemory Go memory store (heap). If an attempt is made to overflow a non-serializable element to the BigMemory Go off-heap or disk stores, the element is removed and a warning is logged.

## What is the difference between TTL, TTI, and eternal?

These three configuration attributes can be used to design [effective data lifetimes](#). Their assigned values should be tested and tuned to help optimize performance. `timeToIdleSeconds` (TTI) is the maximum number of seconds that an element can exist in the store without being accessed, while `timeToLiveSeconds` (TTL) is the maximum number of seconds that an element can exist in the store whether or not it has been accessed. If the `eternal` flag is set, elements are allowed to exist in the store eternally and none are evicted. The eternal setting overrides any TTI or TTL settings.

These attributes are set in the configuration file per cache. To set them per element, you must do so [programmatically](#).

## If null values are stored in the cache, how can my code tell the difference between "intentional" nulls and non-existent entries?

Your application is querying the database excessively only to find that there is no result. Since there is no result, there is nothing to cache. To prevent the query from being executed unnecessarily, cache a null value, signalling that a particular key doesn't exist.

In code, checking for intentional nulls versus non-existent cache entries may look like:

```
// cache an explicit null value:  
  
cache.put(new Element("key", null));  
  
Element element = cache.get("key");  
  
if (element == null) {  
  
    // nothing in the cache for "key" (or expired) ...  
  
} else {  
  
    // there is a valid element in the cache, however getObjectValue() may be null:  
  
    Object value = element.getObjectValue();  
  
    if (value == null) {
```

## Where is the source code?

```
// a null value is in the cache ...  
}  
else {  
    // a non-null value is in the cache ...  
}  
}  
}
```

The cache configuration in `ehcache.xml` may look similar to the following:

```
<cache  
    name="some.cache.name"  
    maxEntriesLocalHeap="10000"  
    eternal="false"  
    timeToIdleSeconds="300"  
    timeToLiveSeconds="600"  
/>
```

Use a finite `timeToLiveSeconds` setting to force an occasional update.

## How many threads does BigMemory Go use, and how much memory does that consume?

The amount of memory consumed per thread is determined by the Stack Size. This is set using `-Xss`.

## What happens when `maxEntriesLocalHeap` is reached? Are the oldest items expired when new ones come in?

When the maximum number of elements in memory is reached, the Least Recently Used (LRU) element is removed. "Used" in this case means inserted with a `put` or accessed with a `get`. The LRU element is flushed asynchronously to the off-heap store.

## Why is there an expiry thread for the disk store but not for the other stores?

Because the in-memory data is allowed a fixed maximum number of elements or bytes, it will have a maximum memory use equal to the number of elements multiplied by the average size. When an element is added beyond the maximum size, the LRU element gets flushed to the disk store. Running an expiry thread in memory turns out to be a very expensive operation and potentially contentious. It is far more efficient to only check expiry when need rather than explicitly search for it. The tradeoff is higher average memory use.

The disk-store expiry thread keeps the disk clean. There is hopefully less contention for the disk store's locks because commonly used values are in memory. If you are concerned about CPU utilization and locking in the disk store, you can set the `diskExpiryThreadIntervalSeconds` to a high number, such as 1 day. Or, you can effectively turn it off by setting the `diskExpiryThreadIntervalSeconds` to a very large value.

## What eviction strategies are supported?

LRU, LFU and FIFO eviction strategies are supported.

## How does element equality work in serialization mode?

An element (key and value) in BigMemory is guaranteed to `.equals()` another as it moves between stores.

Where is the source code?

## Can you use BigMemory Go as a second-level cache in Hibernate and BigMemory Go outside of Hibernate at the same time?

Yes. You use one instance of BigMemory Go with one ehcache.xml. You configure your caches with Hibernate names for use by Hibernate. You can have other caches which you interact with directly, outside of Hibernate.

# Operations

## How do you get an element without affecting statistics?

Use the [Cache.getQuiet\(\)](#) method. It returns an element without updating statistics.

## Is there a simple way to disable BigMemory Go when testing?

Set the system property `net.sf.ehcache.disabled=true` to disable BigMemory Go. This can easily be done using `-Dnet.sf.ehcache.disabled=true` on the command line. If BigMemory Go is disabled, no elements will be added to the stores.

## How do I dynamically change cache attributes at runtime?

This is not possible. However, you can achieve the same result as follows:

1. Create a new cache:

```
Cache cache = new Cache("test2", 1, true, true, 0, 0, true, 120, ...);  
cacheManager.addCache(cache);
```

See the [BigMemory API documentation](#) for the full parameters.

2. Get a list of keys using `cache.getKeys`, then get each element and put it in the new cache.

None of this will use much memory because the new cache elements have values that reference the same data as the original cache.

3. Use `cacheManager.removeCache("oldcachename")` to remove the original cache.

## Do you need to explicitly shut down the CacheManager when you finish with BigMemory Go?

There is a [shutdown hook](#) which calls the shutdown on JVM exit. If the JVM keeps running after you stop using BigMemory Go, you should call `CacheManager.getInstance().shutdown()` so that the threads are stopped and cache memory is released back to the JVM.

## Can you use BigMemory Go after a CacheManager.shutdown()?

When you call `CacheManager.shutdown()` it sets the singleton in CacheManager to null. Using a cache after this generates a CacheException.

However, if you call `CacheManager.create()` to instantiate a new CacheManager, then you can still use BigMemory Go. Internally the CacheManager singleton gets set to the new one, allowing you to create and shut down any number of times.

How do you get an element without affecting statistics?

### Why are statistics counters showing 0 for active caches?

Statistics gathering is disabled by default in order to optimize performance. You can enable statistics gathering in caches in one of the following ways:

- In cache configuration by adding `statistics="true"` to the `<cache>` element.
- Programmatically when setting a cache's configuration.
- In the
- In the [Terracotta Management Console](#).

To function, certain features in the Terracotta Management Console require statistics to be enabled.

### How do I detect deadlocks in BigMemory Go?

BigMemory Go does not experience deadlocks. However, deadlocks in your application code can be detected with certain tools, such as the JDK tool JConsole.

## Troubleshooting

### I have created a new cache and its status is STATUS\_UNINITIALISED. How do I initialise it?

You need to add a newly created cache to a CacheManager before it gets initialised. Use code like the following:

```
CacheManager manager = CacheManager.create();
Cache myCache = new Cache("testDiskOnly", 0, true, false, 5, 2);
manager.addCache(myCache);
```

### Why did a crashed standalone BigMemory node not come up with all data intact?

Persistence was not configured or not configured correctly on the node.

### I added data Client 1, but I can't see it on Client 2. Why not?

BigMemory Go does not distribute data. See [BigMemory Max](#).

### I have a small data set, and yet latency seems to be high.

There are a few ways to try to solve this, in order of preference:

1. Try pinning the cache. If the data set fits comfortably in heap and is not expected to grow, this will speed up gets by a noticeable factor. Pinning certain elements and/or tuning ARC settings might also be effective for certain use cases.
2. Increase the size of the off-heap store to allow data sets that cannot fit in heap—but can fit in memory—to remain very close to your application.

### I am using Java 6 and getting a java.lang.VerifyError on the Backport Concurrent classes. Why?

The backport-concurrent library is used in BigMemory Go to provide `java.util.concurrency` facilities for Java 4 - Java 6. Use either the Java 4 version which is compatible with Java 4-6, or use the version for your JDK.

I have created a new cache and its status is STATUS\_UNINITIALISED. How do I initialise it?

### I get a javax.servlet.ServletException: Could not initialise servlet filter when using SimplePageCachingFilter. Why?

If you use this default implementation, the cache name is called "SimplePageCachingFilter". You need to define a cache with that name in ehcache.xml. If you override CachingFilter, you are required to set your own cache name.

### Why is there a warning in my application's log that a new CacheManager is using a resource already in use by another CacheManager?

```
WARN CacheManager ... Creating a new instance of CacheManager using the diskStorePath "C:\temp\tempcache" which is already used by an existing CacheManager.
```

This means that, for some reason, your application is trying to create one or more additional instances of CacheManager with the same configuration. Depending upon your persistence strategy, BigMemory Go will automatically resolve the disk-path conflict, or it will let you know that you must explicitly configure the diskStorePath.

To eliminate the warning:

- Use a separate configuration per instance.
- If you only want one instance, use the singleton creation methods, i.e.,  
`CacheManager.getInstance()`. In Hibernate, there is a special provider for this called  
`net.sf.ehcache.hibernate.SingletonEhCacheProvider`. See [Hibernate](#).

### What does the following error mean? "Caches cannot be added by name when default cache config is not specified in the config. Please add a default cache config in the configuration."

The `defaultCache` is optional. When you try to programmatically add a cache by name, `CacheManager.add(String name)`, a default cache is expected to exist in the CacheManager configuration. To fix this error, add a `defaultCache` to the CacheManager's configuration.

# BigMemory Max Tutorial

Follow these steps to run a sample application with BigMemory Max.

## Prerequisites

BigMemory is designed to use large amounts of memory for caching without causing long garbage collection pauses. While you can get value from BigMemory in applications using 1GB of memory or less, we recommend that you try this sample with a 64-bit JVM on a machine with 8GB or more of free memory.

## 1 Download and Install

To run this tutorial, you will need to download the following:

1. [BigMemory \(distributed in the Terracotta Enterprise kit\)](#) >
2. The trial license for Enterprise Ehcache (available when you sign up for the 30-day trial)
3. [The Ehcache Pounder sample](#) >

### Install the Terracotta Enterprise kit

Install the Terracotta Enterprise kit:

```
%> java -jar terracotta-ee-3.7.4.jar
```

### Untar the Ehcache Pounder

The Ehcache Pounder is a sample application that exercises the characteristics of different data stores for Enterprise Ehcache, including the default on-heap store, the BigMemory off-heap store, and the disk store.

This tutorial will show the performance and latency differences between the default Ehcache on-heap data store and BigMemory used as an off-heap data store. The source for the Ehcache Pounder can be found on [GitHub](#). For best results, run the tutorial on a machine with at least eight gigabytes of free RAM.

Untar the Ehcache Pounder bundle. For example (note that your version may vary):

```
%> tar xfvz ehcache-pounder-0.0.7-SNAPSHOT-distribution.tar.gz
```

### Copy the Enterprise Ehcache JAR from the Terracotta Enterprise kit to the top-level Ehcache Pounder directory

The Pounder expects the Enterprise Ehcache core JAR library to be in its top-level directory. Copy the Enterprise Ehcache core JAR library to the top-level Ehcache Pounder directory:

```
%> cp terracotta-ee-3.7.4/ehcache/lib/ehcache-core-ee-2.6.jar \
ehcache-pounder-0.0.7-SNAPSHOT/ehcache-core-ee.jar
```

Note: when copied into the ehcache-pounder directory, the ehcache-core library must be named 'ehcache-core-ee.jar'.

### Copy the License Key to the Pounder Directory

## Download and Install

The Pounder expects the trial license key to be in its top-level directory. Copy the the trial license to the Pounder directory:

```
%> cp terracotta-ee-3.7.4/terracotta-license.key \
ehcache-pounder-0.0.7-SNAPSHOT/
```

## 2 Choose a Sample Template

The Ehcache Pounder sample has been pre-configured to run with different heap and cache sizes, depending on the memory available in your environment. See the 'templates' directory in the Ehcache Pounder sample for the list of available configuration templates. For example, if you only have enough free memory to run the sample with 500 megabytes of cache data, use the '500MB-BigMemory' and '500MB-on-heap' templates for the following steps in this tutorial. Likewise, if you have enough free memory to run the sample with 20 gigabytes of cache data, use the '20GB-BigMemory' and '20GB-on-heap' templates.

### Important

Make sure you have enough free memory to run the sample you choose.

BigMemory will be effective only if you ensure that there is enough free RAM to service the amount of memory you configure for your sample. You should ensure that the Java process will not be swapped out.

To this end, you should first check your available free memory, and also monitor the free memory while you run these samples.

Tips on checking your available memory:

- On Linux run: top, free, or vmstat
- On MAC OSX run: vm\_stat, or simply use the Activity Monitor
- On Windows: look at the performance tab of Task Manager, or your other favorite tool

The following steps in this tutorial will use the 20 GB configuration templates. Make sure you choose the configuration templates suitable for the amount of free memory available on your machine.

## 3 Run the On-Heap Sample

We will start by running the sample using a standard, on-heap memory store. This will store all cache data on the Java heap. While the Java heap provides the fastest access to cache data, at larger heap sizes garbage collection pauses can have a severe negative impact on application performance. For the 20GB sample, garbage collection pauses can last for minutes, during which time the application halts all normal operation. To see these long pauses in action, let's run the sample:

```
%> cd ehcache-pounder-0.0.7-SNAPSHOT/templates/20GB-on-heap/
%> sh run-pounder.sh
```

**These scripts were tested with Java 1.6 in 64bit mode, on an Oracle JVM. To run on 1.5 JVMs you may need to remove `-XX:+UseCompressedOops` and add `-d64` to run in 64bit mode. Other changes to startup options may be necessary, such as substituting `-Xcompressedrefs` for `-XX:+UseCompressedOops` if using an IBM JVM.**

By default, the run-pounder.sh script tees the terminal output to /tmp/pounder.log. You can edit the run script to tee the output to any file you choose. It is also configured to send verbose garbage collection

## Choose a Sample Template

logging to /tmp/pounder.gc.log. You can inspect that file to see when garbage collection pauses are happening. Later in the tutorial, we will show how to use a free, open source tool called GCV viewer to analyze the verbose garbage collection logging.

### Results

The Ehcache Pounder has a warmup phase and a run phase. During the warmup phase, it writes cache entries into the cache as fast as it can. During the run phase, it cycles through the cache entries with multiple threads, reading and writing cache entries according to the configured read/write ratio. It goes through a configurable number of rounds of this run cycle. At the end of the run phase, it prints out summary results for each round and the run phase as a whole. Here is an example of the results using the 20GB-on-heap template:

```
All Rounds:  
Round 1: elapsed time: 120329, final cache size: 19999980, tps: 166210  
Round 2: elapsed time: 75314, final cache size: 19999980, tps: 265554  
Round 3: elapsed time: 80299, final cache size: 19999980, tps: 249069  
Round 4: elapsed time: 74010, final cache size: 19999980, tps: 270233  
ONHEAP Pounder Final Results  
TOTAL TIME: 229623ms, AVG TPS (excluding round 1): 261618.67
```

You can see that the average transaction rate is around 260,000 transactions per second (TPS). Since the JVM performs runtime optimizations during the first round that affect performance, the summary results exclude the first round from its total time and average transaction rate calculation. Your results will vary depending on the environment you run the test in. (*Note: the transaction rate indicated by the Ehcache Pounder is meant to compare the performance of the various Ehcache data stores. It is not meant as an indication of the transaction rate you should expect from using Enterprise Ehcache in your application. The transaction rate you will see when using Enterprise Ehcache in your application will depend on many factors, including how your application uses the cache, how the cache is configured, and the hardware it runs on.*)

## 4 Run the BigMemory Sample

To see the improvement BigMemory provides for both transaction rate (throughput) and application responsiveness (latency), run the BigMemory sample:

```
%> cd ehcache-pounder-<version>/templates/20GB-BigMemory/  
%> sh run-pounder.sh
```

### Results

The BigMemory sample runs *exactly* the same code as the on-heap sample. The only differences are the size of the JVM heap and the data store that Ehcache is configured to use. Since BigMemory uses an off-heap storage mechanism, it only uses 200MB of heap, even though the cache size is 20GB. As a result, there are no long garbage collection pauses so the application runs faster and with much more predictable latency.

```
All Rounds:  
Round 1: elapsed time: 76427, final cache size: 19999980, tps: 261687  
Round 2: elapsed time: 24284, final cache size: 19999980, tps: 823587  
Round 3: elapsed time: 24517, final cache size: 19999980, tps: 815760  
Round 4: elapsed time: 24371, final cache size: 19999980, tps: 820647  
BigMemory Pounder Final Results  
TOTAL TIME: 73172ms, AVG TPS (excluding round 1): 819998.0
```

## Choose a Sample Template

As you can see, the same application runs several times faster with BigMemory because it spends nearly no time in garbage collection.

## 5 Analyze Garbage Collection

To see how garbage collection affects application performance, it's important to know when garbage collection pauses happen and how long they stop your application from working. We will use GCViewer, a free open source garbage collection log analyzer and visualization tool available from <http://www.tagtraum.com/gcviewer.html> >

If you run the pounder sample in Hotspot, detailed logs of garbage collection activity are written to a log file. By default, these logs are written to /tmp/pounder.gc.log. Edit run-pounder.sh to change the default GC log file location.

### GCViewer Tips

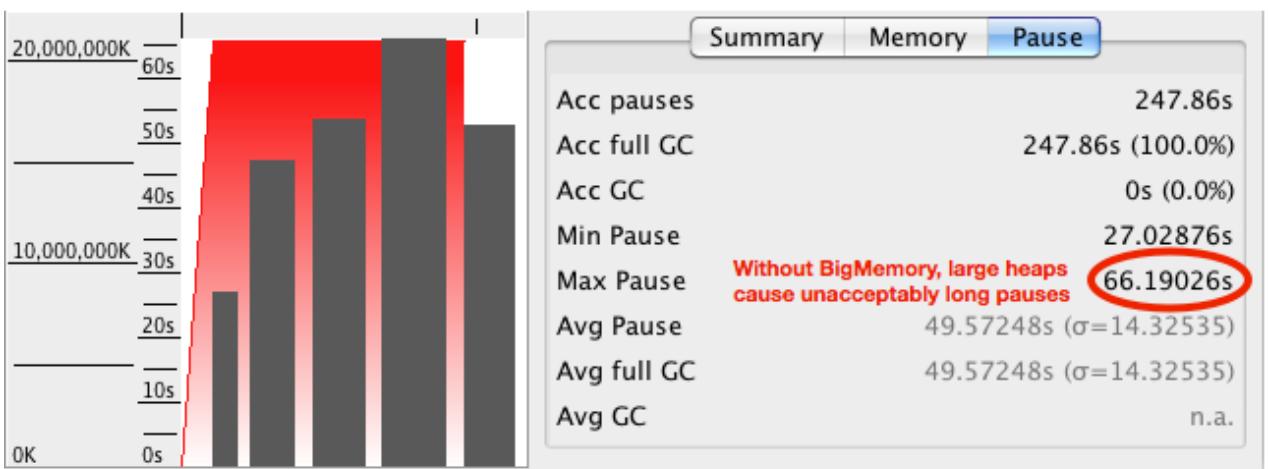
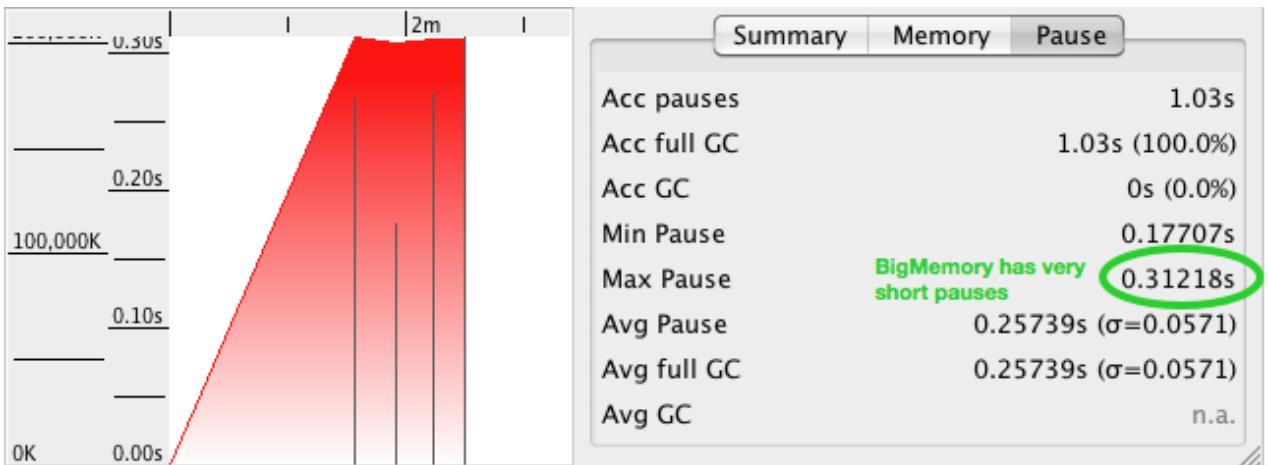
GCViewer is a powerful and free open source tool. However, it may take some getting used to. You can always inspect the log manually to see the garbage collection profile. Here are some basic tips to help you use GCViewer:

- There are known issues parsing some of the garbage collector log output. Often, you may just ignore the parsing errors and still get a useful graph.
- Try using a magnification of 5000% to see more detail.
- Try turning off some of the view options in the "View" menu to filter out extraneous information.
- The gc log settings in run-pounder.sh are specific to Hotspot. See the [GCViewer page](#) for JVM argument settings for use with other JVMs.
- **Even if GCViewer doesn't work for your JVM output, you can learn a lot by scanning through the GC log in your favorite text editor. For example, look for occurrences of Full GC in your on-heap test compared to your BigMemory test.**

Presented below are some samples of GCViewer's visualization of heap usage and garbage collection activity analysis for the BigMemory and on-heap pounder example runs discussed earlier in this tutorial.

### BigMemory's Positive Effect on Application Latency

## Choose a Sample Template

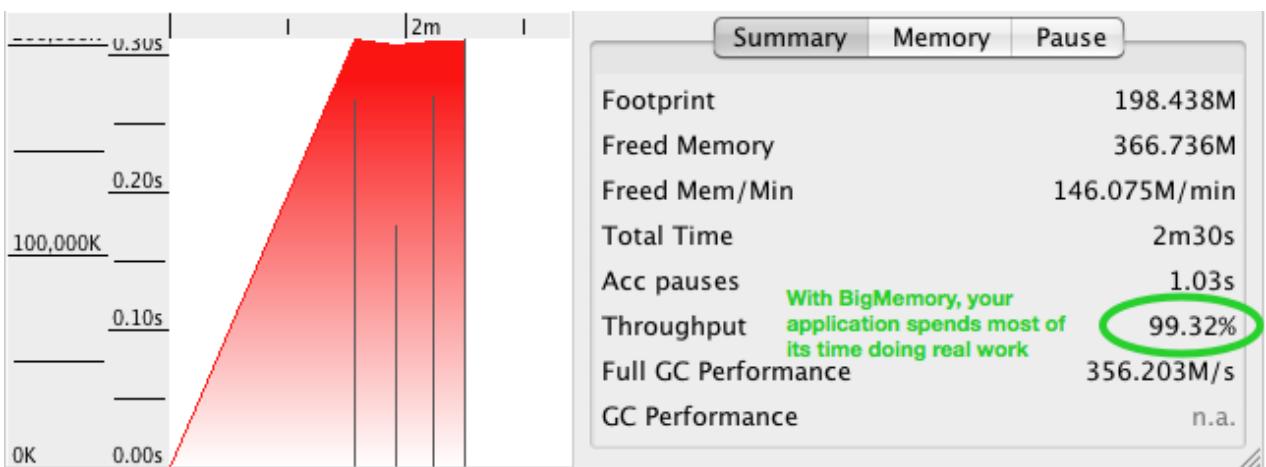


The graph in the top image shows the heap usage and garbage collection events for BigMemory. Because nearly all of the 20GB cache is off-heap in BigMemory, the heap is set to a very small 200MB. As a result, the largest GC pause is a similarly small 0.3 seconds. This means that the application spends the vast majority of its time doing actual work and delivers small and predictable latency.

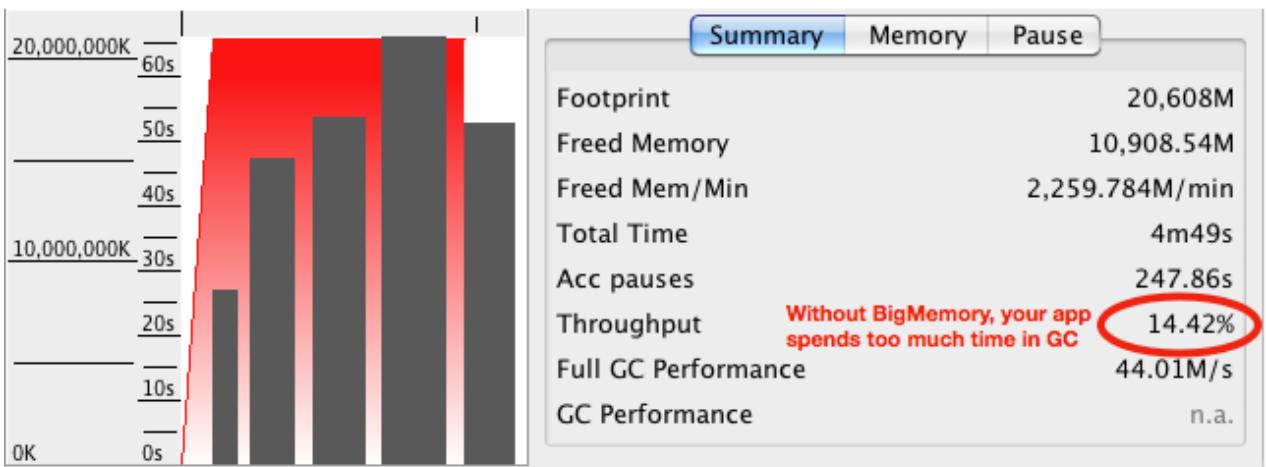
The graph in the lower image above shows the heap usage and garbage collection events for the on-heap run. Because all of the 20GB cache is on the JVM heap, it must be set to 20GB. As a result, there are many GC pauses (indicated by the large black bars), the shortest one lasting for almost 30 seconds, the longest for more than a minute. This means that the application spends much of its time doing nothing but garbage collection and pauses unpredictably, unresponsive for seconds or minutes at a time.

### BigMemory's Positive Effect on Application Throughput

## Choose a Sample Template



*Garbage collection analysis for 20GB BigMemory pounder execution*



*Garbage collection analysis for 20GB on-heap pounder execution*

VisualGC's measure of throughput, shown above, describes the percentage of time the JVM spends running your application code versus performing GC operations. BigMemory exhibits far better performance, since it spends most of its time doing real work. Without it the JVM spends much of its time doing garbage collection.

## Next Steps

See the BigMemory documentation on Ehcache.org for all of the information you need to use BigMemory in your application.

[Have someone contact me with more information >](#)

# Working with Terracotta License Files

A Terracotta license file is required to run enterprise versions of Terracotta products. The name of the file is `terracotta-license.key` and must not be changed. 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 node using an enterprise version of Terracotta software 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 software kit. To avoid having to explicitly specify the file's location, you can leave it in the 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 Terracotta kits. 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.

## 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`.

## 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
```

## Verifying Products and Features

```
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.

# Configuring BigMemory

BigMemory is configured in the Terracotta server's environment and in its configuration file.

## 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 [Configuring Off-Heap](#)).

## 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 block as shown:

```
<server host="myHost" name="server1">
  ...
  <persistence>
    ...
    <offheap>
      <enabled>true</enabled>
    <!-- Allocate 9GB of off-heap memory for clustered data. -->
    <maxDataSize>9g</maxDataSize>
  </offheap>
  ...
</persistence>
...
</server>
```

The amount of configured off-heap memory must be at least 128MB and at least 32MB less than the amount of memory allocated by `MaxDirectMemorySize`. This is because 32MB of memory is utilized by the server's communication layer.

If, at startup, a server determines that the memory allocated by `MaxDirectMemorySize` is insufficient, an error similar to the following is logged:

## Configuring Off-Heap

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.

## 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.

### Notes for 32-Bit Systems

In a 32-bit process model, the amount of heap-offload you can achieve is limited by the addressable memory. The maximum virtual address size of the process is typically 4 GB, 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.

### Default Value of Direct Memory Space

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.

## Optimizing BigMemory

Note the following recommendations:

- Thoroughly test BigMemory with your application before going to production. **It is recommended that you test BigMemory with the actual amount of data you expect to use in production.**
- Be sure to allot at least 15 percent more off-heap memory to BigMemory than the size of your data set. To maximize performance, BigMemory reserves a portion of off-heap memory to store meta-data and other purposes.
- If working with distributed cache, consider using the [sizing parameters](#) available through Ehcache

## Optimizing BigMemory

configuration.

If performance or functional issues arise, see the suggested tuning tips in this section.

## 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.

## 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:+UseCompressedOops`. For IBM JVMs, use `-Xcompressedrefs`.

## Swapiness and Huge Pages

An OS could 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 heap memory use more efficient, Linux, Microsoft Windows, and Oracle Solaris users should review their configuration and usage of swapiness as well as the size of the swapped memory pages. In general, BigMemory benefits from lowered swapiness and the use of *huge pages* (also known as *big pages*, *large pages*, and *superpages*). Settings for these behaviors vary by OS and JVM. For Oracle HotSpot, `-XX:+UseLargePages` and `-XX:LargePageSizeInBytes=<size>` (where `<size>` is a value allowed by the OS for specific CPUs) can be used to control page size. However, note that this setting does not affect how off-heap memory is allocating. Over-allocating huge pages while also configuring substantial off-heap memory *can starve off-heap allocation and lead to memory and performance problems*.

## Maximum Serialized Size of an Element

This section applies when using BigMemory through the Ehcache API.

Unlike the memory and the disk stores, by default the off-heap store has a 4MB limit for classes with high quality hashcodes, and 256KB limit for those with pathologically bad hashcodes. The built-in classes such as `String` and the `java.lang.Number` subclasses `Long` and `Integer` have high quality hashcodes. This can cause issues when objects are expected to be larger than the default limits.

To override the default size limits, set the system property

`net.sf.ehcache.offheap.cache_name.config.idealMaxSegmentSize` to the size you require.

For example,

```
net.sf.ehcache.offheap.com.company.domain.State.config.idealMaxSegmentSize=30M
```

# The Terracotta Server Array

The Terracotta Server Array provides the platform for Terracotta products and the backbone for Terracotta clusters. 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 inter-node 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

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.

To learn about the Terracotta Server Array with specific products such as BigMemory and Enterprise Ehcache, see the Terracotta documentation for those products.

# Terracotta Server Arrays Architecture

## Introduction

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.

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.

## 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.

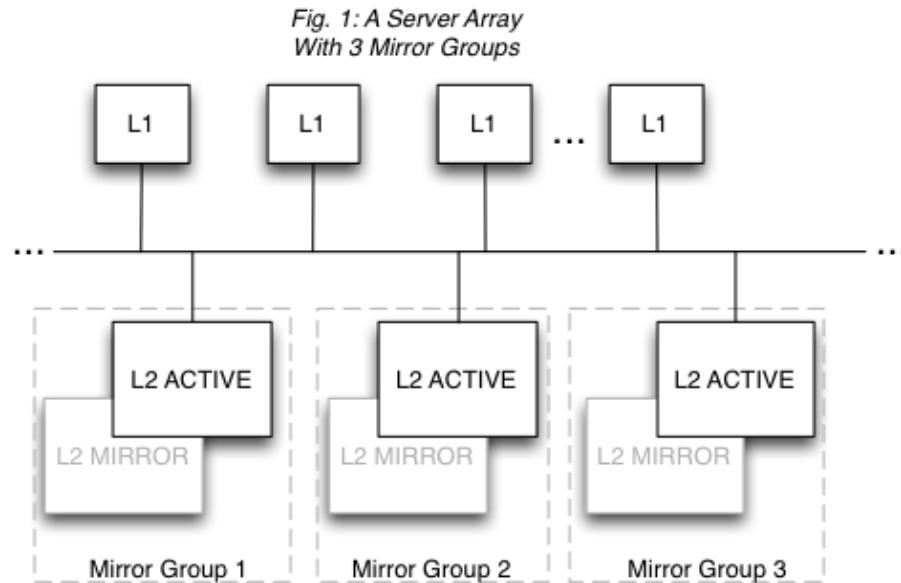
TIP: Switching Server-Array Databases

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. 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: 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: To reset to the default embedded database, remove all Derby-related properties from the Terracotta configuration

## Definitions and Functional Characteristics

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:

- 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](#).

## 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":

networked-active-passive 5

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 \(Servers Configuration Section\)](#)

## 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.

## 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](#).

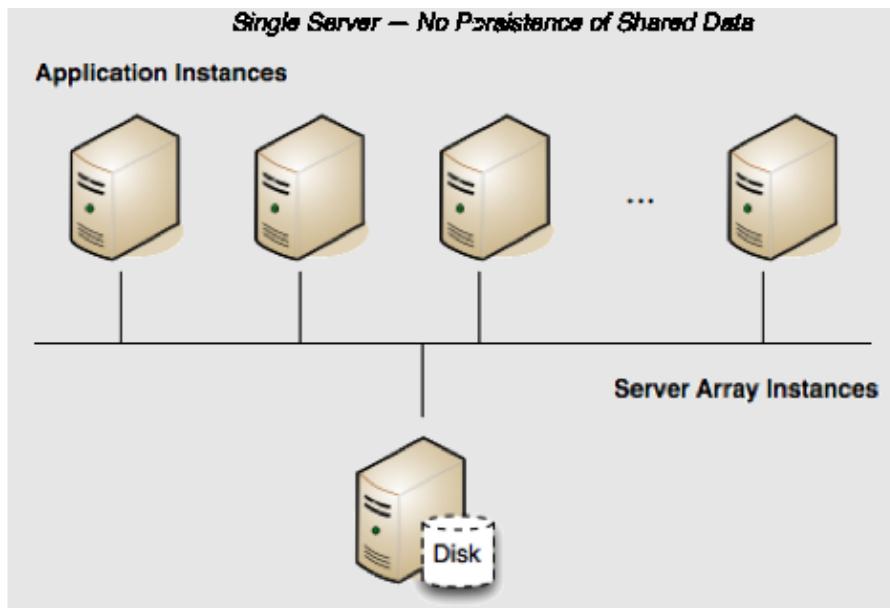
## Cluster Structure and Behavior

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](#)).

### 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">
```

## Terracotta Cluster in Development

```
<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.

## 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:

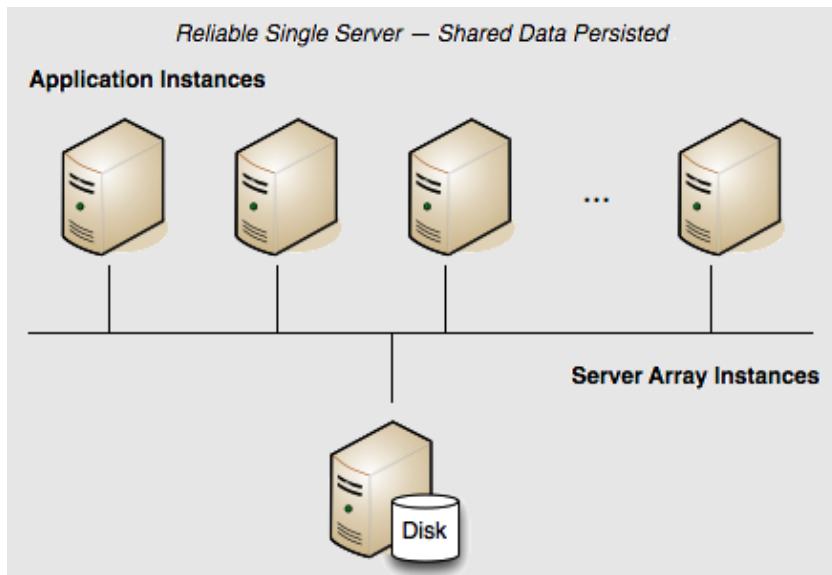
```
<?xml version="1.0" encoding="UTF-8" ?>
<tc:tc-config xmlns:tc="http://www.terracotta.org/config"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
  <servers>
    <server name="Server1">
      <data>/opt/terracotta/server1-data</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.

## Terracotta Cluster With Reliability



### 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>
```

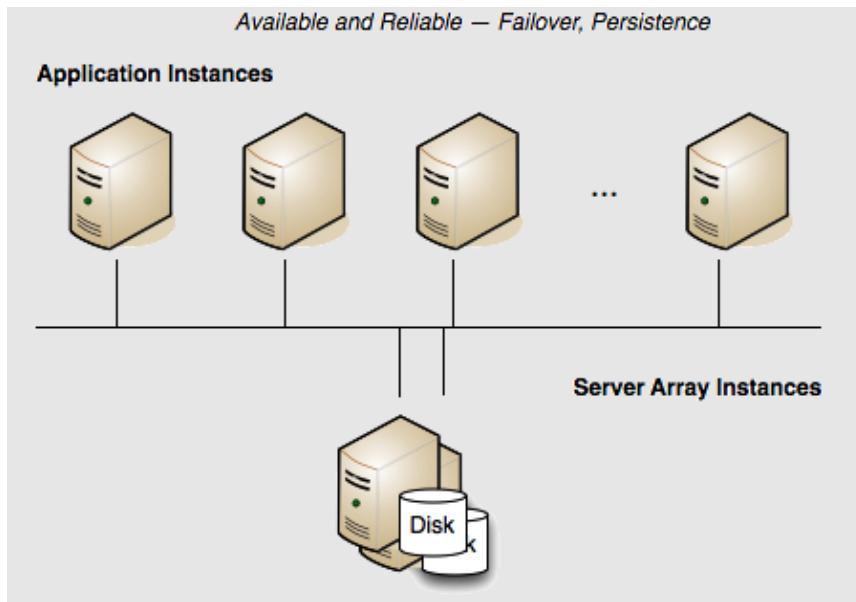
The `<client-reconnect-window>` does not have to be explicitly set if the default value is acceptable. However, in a single-server cluster `<client-reconnect-window>` is in effect only if persistence mode is set to "permanent-store".

## Terracotta Server Array with High Availability

## 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>
```

## Terracotta Server Array with High Availability

```
</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

## Terracotta Server Array with High Availability

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="" />
```

where <myValue> is an integer. To prevent any backups from being saved, add the following element to the Terracotta configuration file's <tc-properties> block:

```
<property "l2.nha.dirtydb.backup.enabled" value="false" />
```

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.

NOTE: If the script detects that the passive server in STANDBY state isn't reachable, it issues a warning and fails to shut down the active server. If failover is not a concern, you can override this behavior with the --force flag.

Clients should connect to the new active server.

## Terracotta Server Array with High Availability

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:

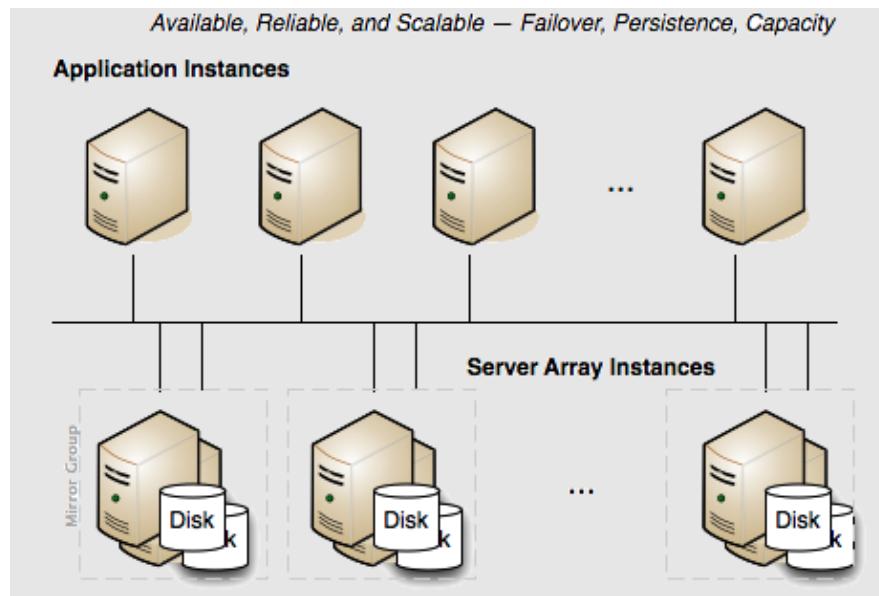
- No clients connect to the new active server – The original active server "zaps" the new active server, causing it to restart, wipe its database, and synchronize again as a standby.
- A minority of clients connect to the new active server – The original active server starts a reconnect timeout (based 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 you confirm the integrity of shared data after such an occurrence.

## 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>
```

## Scaling the Terracotta Server Array

```
</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>
```

## About Distributed Garbage Collection

```
<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.

## About Distributed Garbage Collection

Every Terracotta server instance runs a garbage collection algorithm to determine which clustered objects can be safely removed both from the memory and the disk store of the server instance. This garbage collection process, known as the Distributed Garbage Collector (DGC), is analogous to the JVM garbage collector but works only on clustered object data in the server instance.

The DGC is not a distributed process; it operates only on the distributed object data in the Terracotta server instance in which it lives. In essence, the DGC process walks all of the clustered object graphs in a single

## Types of DGC

Terracotta server instance to determine what is currently not garbage. Every object that is garbage is removed from the server instance's memory and from persistent storage (the object store on disk). If garbage objects were not removed from the object store, eventually the disk attached to a Terracotta server instance would fill up.

A clustered object is considered garbage when it is treated as garbage by Java GC. Typically, this happens when an object is not referenced by any other distributed object AND is not resident in any client's heap. Objects that satisfy only one of these conditions are safe from a DGC collection.

In distributed Ehcache, objects that exist in any cache are never collected by DGC. Objects that are removed from a distributed cache are eligible for collection.

## Types of DGC

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

Note that the inline DGC algorithm operates at intervals optimal to maximizing performance, and so does not necessarily collect distributed garbage immediately.

## Running the Periodic DGC

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

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

## Monitoring and Troubleshooting the DGC

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

If DGC does not seem to be collecting objects at the expected rate, one of the following issues may be the cause:

- Java GC is not able to collect objects fast enough. Client nodes may be under resource pressure, causing GC collection to run behind, which then causes DGC to run behind.
- Certain client nodes continue to hold references to objects that have become garbage on other nodes, thus preventing DGC from being able to collect those objects.

If possible, shut down all Terracotta clients to see if DGC then collects the objects that were expected to be collected.

# Working with Terracotta Configuration Files

## Introduction

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.

## 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.

### 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.

### 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.

### 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>
```

## Local or Remote Configuration File

where <path\_to\_configuration\_file> can be a URL or a relative directory path. In Microsoft Windows, use start-tc-server.bat.

# 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.

## 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.

## Terracotta Server

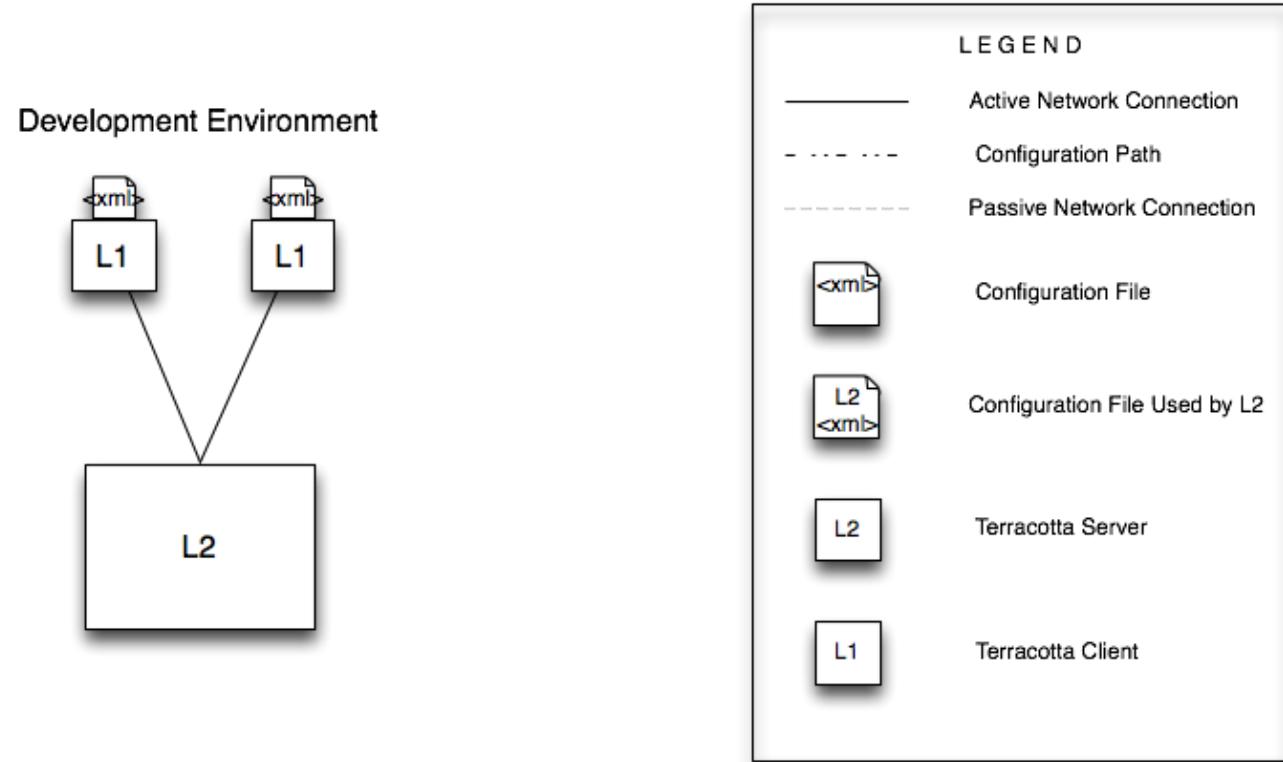
Terracotta clients can load configuration from an active Terracotta server by specifying its hostname and DSO port (see [Clients in Production](#)).

# 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.

## One-Server Setup in Development

For one Terracotta server, the default configuration is adequate.



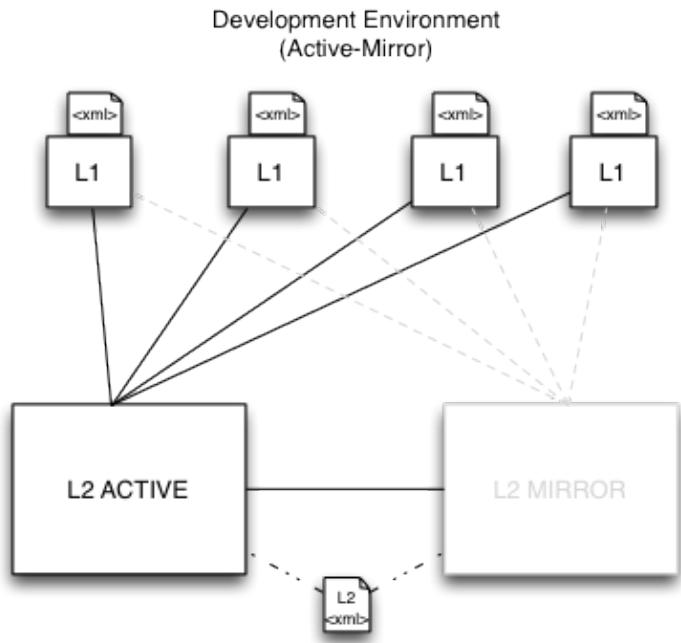
To use the default configuration settings, start your Terracotta server using the `start-tc-server.sh` (or `start-tc-server.bat`) script in a directory that does *not* contain the file `tc-config.xml`:

```
[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](#).

## Two-Server Setup in Development

## 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.

## Two-Server Setup in Development

### 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.

## 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.

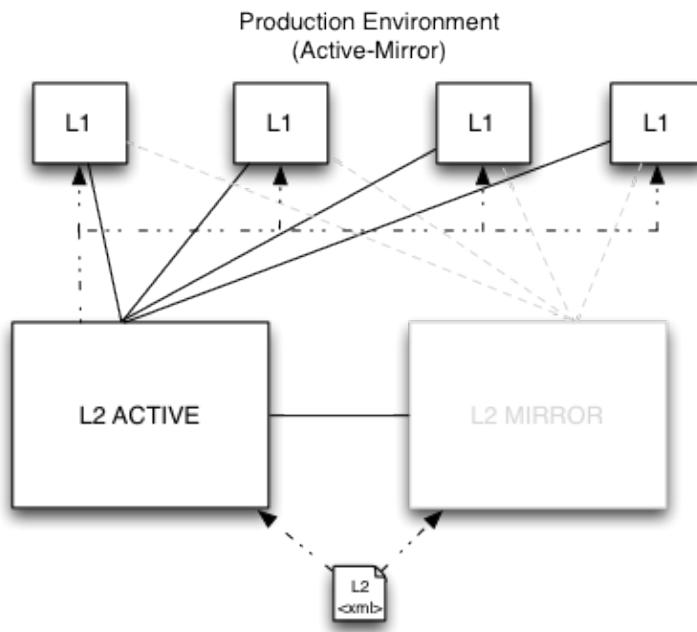
If `tc-config.xml` exists in the directory in which you run dso-java, it can be loaded without `-Dtc.config`.

#### 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](#).

## Configuration in a Production Environment

## 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">
```

## 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

## Clients in Production

```
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](#)).

## 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>
```

## Binding Ports to Interfaces

```
<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>
```

## 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.

## 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".

# Configuring Terracotta Clusters For High Availability

## Introduction

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 services
- Configurable automatic internode monitoring – Terracotta [HealthChecker](#)
- Automatic permanent storage of all current shared (in-memory) data – available to all server instances (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

Client reconnection refers to reconnecting clients that have not yet been disconnected from the cluster by the Terracotta Server Array. To learn about reconnecting Enterprise Ehcache clients that have been disconnected from their cluster, see [Using Rejoin to Automatically Reconnect Terracotta Clients](#).

### 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.

It is important to thoroughly test any High Availability setup before going to production. Suggestions for testing High Availability configurations are provided in [this section](#).

## 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.

## Common Causes of Failures in a Cluster

- Ensuring that the problem node has enough memory allocated to the heap.

Other sources of failures in a cluster are disks that are nearly full or are running slowly, and running other applications that compete for a node's resources. See [Best Practices](#) for more advice on preventing issues and failures in your cluster.

## Using BigMemory to Alleviate GC Slowdowns

Terracotta BigMemory allows L2s to use memory outside of the Java object heap. This also helps prevent another common cause of slowdowns: swapping. Swapping occurs when the system must use disk space to accomodate a data set that is too large for the heap. See [Improving Server Performance With BigMemory](#).

[HealthChecker](#) can also be tuned to allow for GC or network interruptions.

## 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

## Basic High-Availability Configuration

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 \(Servers Configuration Section\)](#)

# 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.

## 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:

## HealthChecker

- 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><br><code>l2.healthcheck.l2.ping.enabled</code><br><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><br><code>l2.healthcheck.l2.ping.idletime</code><br><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><br><code>l2.healthcheck.l2.ping.interval</code><br><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><br><code>l2.healthcheck.l2.ping.probes</code><br><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><br><code>l2.healthcheck.l2.socketConnect</code><br><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><br><code>l2.healthcheck.l2.socketConnectTimeout</code><br><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><br><code>l2.healthcheck.l2.socketConnectCount</code><br><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>  |   |

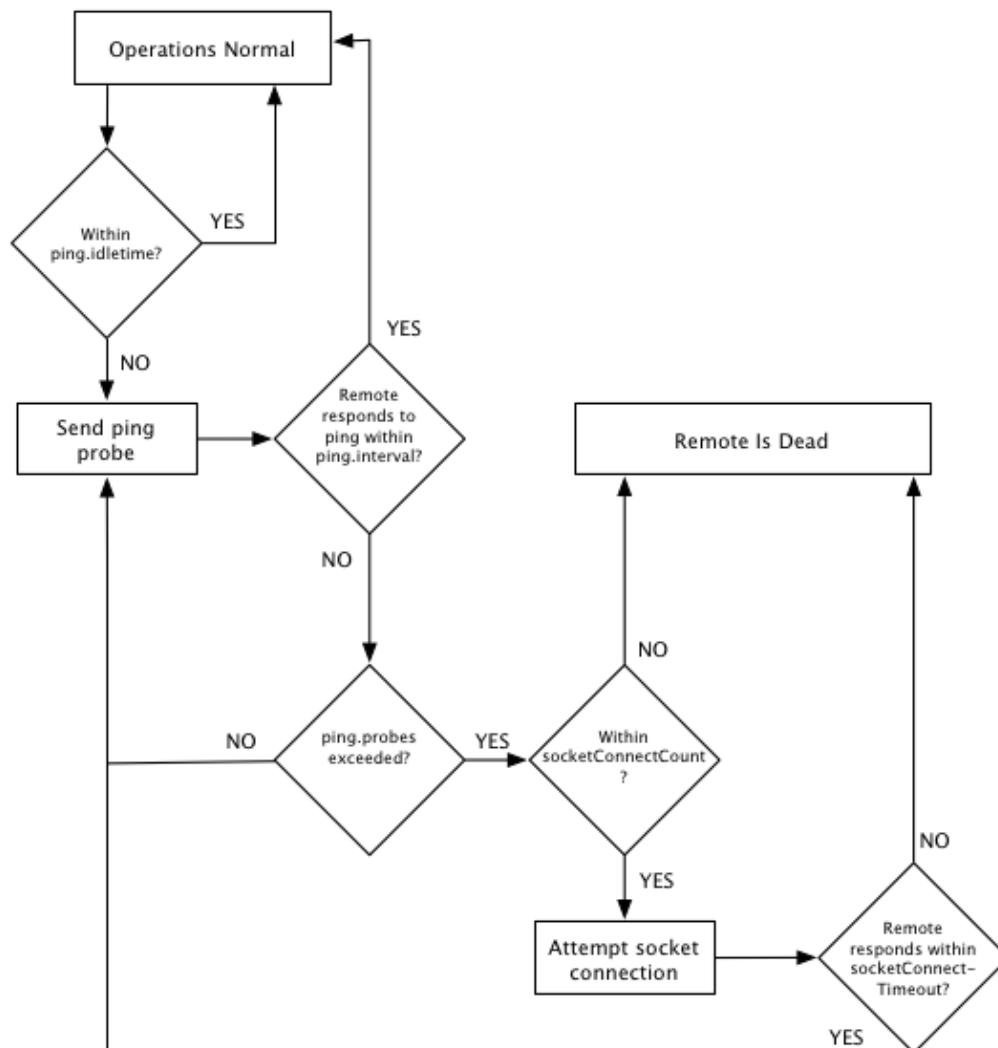
## HealthChecker

11.healthcheck.12.bindPort

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.

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

# 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.
- To prevent clients from disconnecting too quickly in a situation where an active server is temporarily disconnected from both the backup server and those clients, ensure that the Max Time for L1->L2 is approximately 8-12 seconds longer than for L2->L2. If the values are too close together, then in certain situations the active server may kill the backup and refuse to allow clients to reconnect.

## 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 a node dead in no more than 27 seconds.

## Calculating HealthChecker Maximum

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="11.healthcheck.12.ping.enabled" value="true" />
<property name="11.healthcheck.12.ping.idletime" value="5000" />
<property name="11.healthcheck.12.ping.interval" value="1000" />
<property name="11.healthcheck.12.ping.probes" value="3" />
<property name="11.healthcheck.12.socketConnect" value="true" />
<property name="11.healthcheck.12.socketConnectTimeout" value="5" />
<property name="11.healthcheck.12.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.

## Tuning HealthChecker to Allow for GC or Network Interruptions

GC cycles do not affect a node's ability to respond to socket-connection requests, while network interruptions do. This difference can be used to tune HealthChecker to work more efficiently in a cluster where one or the other of these issues is more likely to occur:

- To favor detection of network interruptions, adjust the socketConnectCount down (since socket connections will fail). This will allow HealthChecker to disconnect a client sooner due to network issues.
- To favor detection of GC pauses, adjust the socketConnectCount up (since socket connections will succeed). This will allow clients to remain in the cluster longer when no network disconnection has occurred.

## Automatic Server Instance Reconnect

The ping interval increases the time before socket-connection attempts kick in to verify health of a remote node. The ping interval can be adjusted up or down to add more or less tolerance in either of the situations listed above.

## 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` – (DEFAULT: false) When set to "true" enables a server instance to attempt reconnection with its peer server instance after a disconnection is detected. Most use cases should benefit from enabling this setting.
- `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.

## Automatic Client Reconnect

Clients disconnected from a Terracotta cluster normally require a restart to reconnect to 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.llreconnect.enabled` – (DEFAULT: false) 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

## Automatic Client Reconnect

emits a mismatch error and exits. Most use cases should benefit from enabling this setting.

- `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.

## 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>
    ...
  </dso>
  ...
</server>
</servers>
```

Clients which fail to connect to the new active server must be restarted if they are to successfully reconnect to 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:

## Special Client Connection Properties

```
# -1 == retry all configured servers eternally.  
11.max.connect.retries = -1  
# Must the client and server be running the same version of Terracotta?  
11.connect.versionMatchCheck.enabled = true  
# Time (in milliseconds) before a socket connection attempt is timed out.  
11.socket.connect.timeout=10000  
# Time (in milliseconds; minimum 10) between attempts to connect to a server.  
11.socket.reconnect.waitInterval=1000
```

A client with `11.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.

The property `11.max.connect.retries` controls connection attempts only *after* the client has obtained and resolved its configuration from the server. To control connection attempts *before* configuration is resolved, set the following property on the client:

```
-Dcom.tc.tc.config.total.timeout=5000
```

This property limits the time (in milliseconds) that a client spends attempting to make an initial connection.

## Effective Client-Server Reconnection Settings: An Example

To prevent unwanted disconnections, it is important to understand potentially complex interaction between HA settings and the environment in which your cluster runs. Settings that are not appropriate for a particular environment can lead to unwanted disconnections under certain circumstances.

In general, it is advisable to maintain an L1-L2 HealthChecker timeout that falls between the L2-L2 HealthChecker timeout as modified in the following inequality:

$$\begin{aligned} \text{L2-L2 HealthCheck + Election Time} \\ < \text{L1-L2 HealthCheck} \\ < \text{L2-L2 HealthCheck + Election Time + Client Reconnect Window} \end{aligned}$$

This allows a cluster's L1s to avoid disconnecting before a client reconnection window is opened (a backup L2 takes over), or to not disconnect if that window is never opened (the original active L2 is still functional). The Election Time and Client Reconnect Window settings, which are found in the Terracotta configuration file, are respectively 5 seconds and 120 seconds by default.

For example, in a cluster where the L2-L2 HealthChecker triggers at 55 seconds, a backup L2 can take over the cluster after 180 seconds ( $55 + 5 + 120$ ). If the L1-L2 HealthChecker triggers after a time that is greater than 180 seconds, clients may not attempt to reconnect until the reconnect window is closed and it's too late.

If the L1-L2 HealthChecker triggers after a time that is less than 60 seconds (L2-L2 HealthChecker + Election Time), then the clients may disconnect from the active L2 before its failure is determined. Should the active L2 win the election, the disconnected L1s would then be lost.

Beginning with Terracotta 3.6.1, a check is performed at server startup to ensure that L1-L2 HealthChecker settings are within the effective range. If not, a warning is printed with a prescription.

# Testing High-Availability Deployments

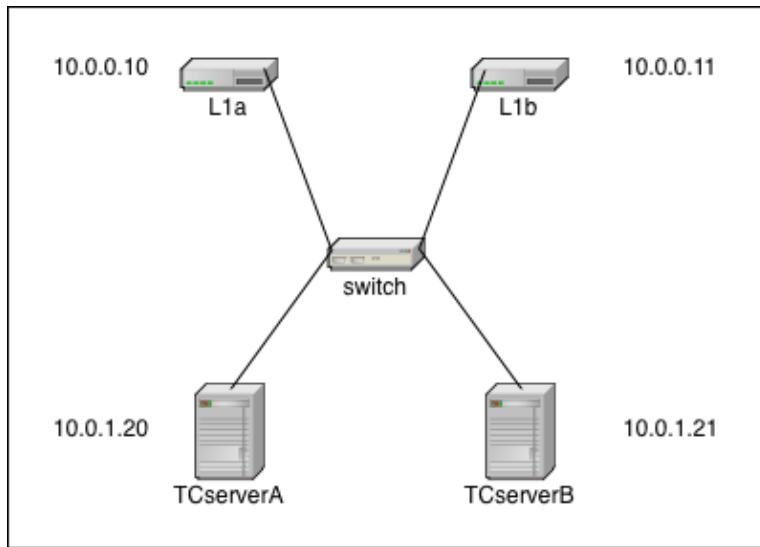
This section presents recommendations for designing and testing a robust cluster architecture. While these recommendations have been tested and shown to be effective under certain circumstances, in your custom environment additional testing is still necessary to ensure an optimal setup, and to meet specific demands related to factors such as load.

## High-Availability Network Architecture And Testing

To take advantage of the Terracotta active-passive server configuration, certain network configurations are necessary to prevent split-brain scenarios and ensure that Terracotta clients (L1s) and server instances (L2s) behave in a deterministic manner after a failure occurs. This is regardless of the nature of the failure, whether network, machine, or other type.

This section outlines two possible network configurations that are known to work with Terracotta failover. While it is possible for other network configurations to work reliably, the configurations listed in this document have been well tested and are fully supported.

### ***Deployment Configuration: Simple (no network redundancy)***



#### Description

This is the simplest network configuration. There is no network redundancy so when any failure occurs, there is a good chance that all or part of the cluster will stop functioning. All failover activity is up to the Terracotta software.

In this diagram, the IP addresses are merely examples to demonstrate that the L1s (L1a & L1b) and L2s (TCserverA & TCserverB) can live on different subnets. The actual addressing scheme is specific to your environment. The single switch is a single point of failure.

## Deployment Configuration: Simple (no network redundancy)

### Additional configuration

There is no additional network or operating-system configuration necessary in this configuration. Each machine needs a proper network configuration (IP address, subnet mask, gateway, DNS, NTP, hostname) and must be plugged into the network.

### Test Plan - Network Failures Non-Redundant Network

To determine that your configuration is correct, use the following tests to confirm all failure scenarios behave as expected.

| TestID | Failure                            | Expected Outcome   |
|--------|------------------------------------|--|
| FS1    | Loss of L1a (link or system)       | Cluster continues as normal using only L1b                       |
| FS2    | Loss of L1b (link or system)       | Cluster continues as normal using only L1a                       |
| FS3    | Loss of L1a & L1b                  | Non-functioning cluster  |
| FS4    | Loss of Switch                     | Non-functioning cluster  |
| FS5    | Loss of Active L2 (link or system) | Passive L2 becomes new Active L2, L1s fail over to new Active L2 |
| FS6    | Loss of Passive L2                 | Cluster continues as normal without TC redundancy                |
| FS7    | Loss of TCservers A & B            | Non-functioning cluster  |

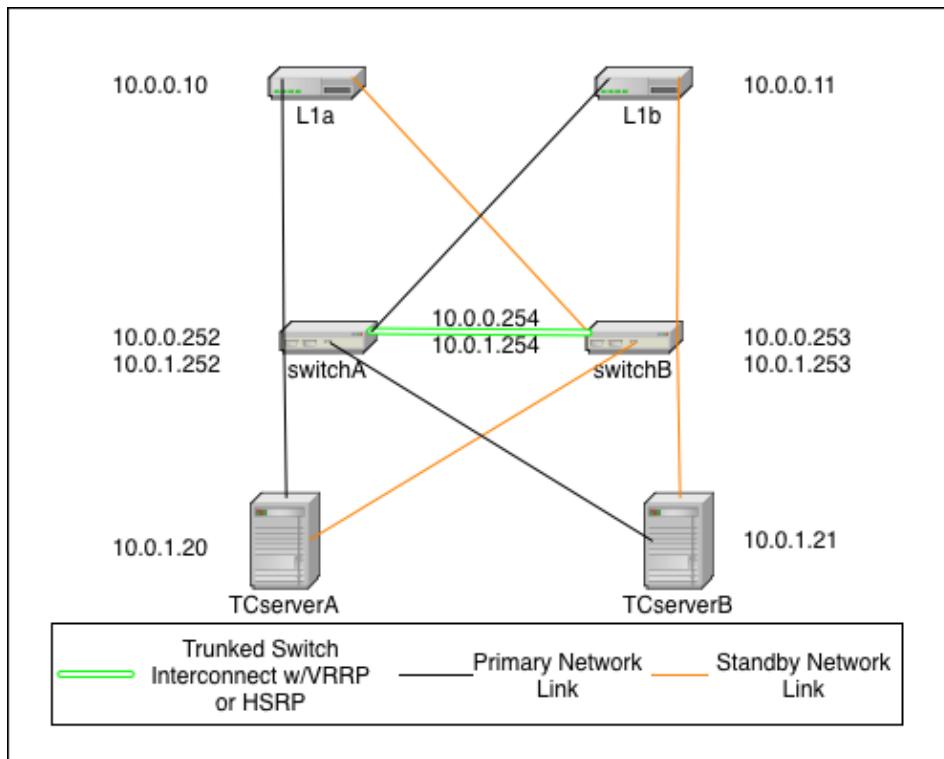
### Test Plan - Network Tests Non-redundant Network

After the network has been configured, you can test your configuration with simple ping tests.

| TestID | Host   | Action                                    | Expected Outcome   |
|--------|--------|---|--|
| NT1    | all    | ping every other host                     | successful ping  |
| NT2    | all    | pull network cable during continuous ping | ping failure until link restored                         |
| NT3    | switch | reload                                    | all pings cease until reload complete and links restored |

## Deployment Configuration: Fully Redundant

## Deployment Configuration: Fully Redundant



### Description

This is the fully redundant network configuration. It relies on the failover capabilities of Terracotta, the switches, and the operating system. In this scenario it is even possible to sustain certain double failures and still maintain a fully functioning cluster.

In this diagram, the IP addressing scheme is merely to demonstrate that the L1s (L1a & L1b) can be on a different subnet than the L2s (TCserverA & TCserverB). The actual addressing scheme will be specific to your environment. If you choose to implement with a single subnet, then there will be no need for VRRP/HSRP but you will still need to configure a single VLAN (can be VLAN 1) for all TC cluster machines.

In this diagram, there are two switches that are connected with trunked links for redundancy and which implement Virtual Router Redundancy Protocol (VRRP) or HSRP to provide redundant network paths to the cluster servers in the event of a switch failure. Additionally, all servers are configured with both a primary and secondary network link which is controlled by the operating system. In the event of a NIC or link failure on any single link, the operating system should fail over to the backup link without disturbing (e.g. restarting) the Java processes (L1 or L2) on the systems.

The Terracotta fail over is identical to that in the simple case above, however both NIC cards on a single host would need to fail in this scenario before the TC software initiates any fail over of its own.

### Additional configuration

- Switch - Switches need to implement VRRP or HSRP to provide redundant gateways for each subnet. Switches also need to have a trunked connection of two or more lines in order to prevent any single link failure from splitting the virtual router in two.

## Deployment Configuration: Fully Redundant

- Operating System - Hosts need to be configured with bonded network interfaces connected to the two different switches. For Linux, choose mode 1. More information about Linux channel bonding can be found in the [RedHat Linux Reference Guide](#). Pay special attention to the amount of time it takes for your VRRP or HSRP implementation to reconverge after a recovery. You don't want your NICs to change to a switch that is not ready to pass traffic. This should be tunable in your bonding configuration.

## Test Plan - Network Failures Redundant Network

The following tests continue the tests listed in Network Failures (Pt. 1). Use these tests to confirm that your network is configured properly.

| TestID | Failure                               | Expected Outcome   |
|--------|---------------------------------------|--|
| FS8    | Loss of any primary network link      | Failover to standby link   |
| FS9    | Loss of all primary links             | All nodes fail to their secondary link   |
| FS10   | Loss of any switch                    | Remaining switch assumes VRRP address and switches fail over NICs if necessary |
| FS11   | Loss of any L1 (both links or system) | Cluster continues as normal using only other L1                                |
| FS12   | Loss of Active L2                     | Passive L2 becomes the new Active L2, All L1s fail over to the new Active L2   |
| FS13   | Loss of Passive L2                    | Cluster continues as normal without TC redundancy                              |
| FS14   | Loss of both switches                 | non-functioning cluster  |
| FS15   | Loss of single link in switch trunk   | Cluster continues as normal without trunk redundancy                           |
| FS16   | Loss of both trunk links              | possible non-functioning cluster depending on VRRP or HSRP implementation      |
| FS17   | Loss of both L1s                      | non-functioning cluster  |
| FS18   | Loss of both L2s                      | non-functioning cluster  |

## Test Plan - Network Testing Redundant Network

After the network has been configured, you can test your configuration with simple ping tests and various failure scenarios.

The test plan for Network Testing consists of the following tests:

| TestID | Host       | Action   | Expected Outcome   |
|--------|------------|--|--|
| NT4    | any        | ping every other host                                      | successful ping  |
| NT5    | any        | pull primary link during continuous ping to any other host | failover to secondary link, no noticeable network interruption |
| NT6    | any        | pull standby link during continuous ping to any other host | no effect  |
| NT7    | Active L2  | pull both network links                                    | Passive L2 becomes Active, L1s fail over to new Active L2      |
| NT8    | Passive L2 | pull both network links                                    | no effect  |

## Terracotta Cluster Tests

|      |         |                        |   |
|------|---------|------------------------|---|
| NT9  | switchA | reload                 | nodes detect link down and fail to standby link, brief network outage if VRRP transition occurs |
| NT10 | switchB | reload                 | brief network outage if VRRP transition occurs  |
| NT11 | switch  | pull single trunk link | no effect   |

## Terracotta Cluster Tests

All tests in this section should be run after the Network Tests succeed.

### Test Plan - Active L2 System Loss Tests - verify Passive Takeover

The test plan for Passive takeover consists of the following tests:

| TestID | Test                            | Setup   | Steps   | Expected Result   |
|--------|---------------------------------|---|---|---|
| TAL1   | Active L2 Loss - Kill           | L2-A is active, L2-B is passive. All systems are running and available to take traffic. | 1. Run app<br>2. Kill -9 Terracotta PID on L2-A (Active)  | L2-B(passive) becomes active. Takes the load. No drop in TPS on Failover. |
| TAL2   | Active L2 Loss - clean shutdown | L2-A is active, L2-B is passive. All systems are running and available to take traffic. | 1. Run app 2.Run ~/bin/stop-tc-server.sh on L2-A (Active) | L2-B(passive) becomes active. Takes the load. No drop in TPS on Failover. |
| TAL3   | Active L2 Loss - Power Down     | L2-A is Active, L2-B is passive. All systems are running and available to take traffic  | 1. Run app 2. Power down L2-A (Active)                    | L2-B(passive) becomes active. Takes the load. No drop in TPS on Failover. |
| TAL4   | Active L2 Loss - Reboot         | L2-A is Active, L2-B is passive. All systems are running and available to take traffic  | 1. Run app 2. Reboot L2-A (Active)                        | L2-B(passive) becomes active. Takes the load. No drop in TPS on Failover. |
| TAL5   | Active L2 Loss - Pull Plug      | L2-A is Active, L2-B is passive. All systems are running and available to take traffic  | 1. Run app 2. Pull the power cable on L2-A (Active)       | L2-B(passive) becomes active. Takes the load. No drop in TPS on Failover. |

### Test Plan - Passive L2 System Loss Tests

System loss tests confirms High Availability in the event of loss of a single system. This section outlines tests for testing failure of the Terracotta Passive server.

The test plan for testing Terracotta Passive Failures consist of the following tests:

| TestID | Test                   | Setup   | Steps   | Expected Result  |
|--------|------------------------|---|---|--|
| TPL1   | Passive L2 loss - kill | L2-A is active, L2-B is passive. All systems are running and available to take traffic. | 1. Run app 2. Kill -9 L2-B (Passive)                        | data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server. |
| TPL2   | Passive L2 loss -clean | L2-A is active, L2-B is passive. All systems are running and available to take traffic. | 1. Run app 2. Run ~/bin/stop-tc-server.sh on L2-B (passive) | data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state                     |

## Terracotta Cluster Tests

|      |                             |  |   |  |
|------|-----------------------------|--|---|--|
|      |                             | take traffic   |   | from Active Server.  |
| TPL3 | Passive L2 loss -power down | L2-A is active, L2-B is passive. All systems are running and available to take traffic | 1. Run app 2. Power down L2-B (Passive)   | data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server. |
| TPL4 | Passive L2 loss -reboot     | L2-A is active, L2-B is passive. All systems are running and available to take traffic | 1. Run app 2. Reboot L2-B (Passive)       | data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server. |
| TPL5 | Passive L2 loss -Pull Plug  | L2-A is active, L2-B is passive. All systems are running and available to take traffic | 1. Run app 2. Pull plug on L2-B (Passive) | data directory needs to be cleaned up, then when L2-B is restarted, it re-synchs state from Active Server. |

### Test Plan - Failover/Failback Tests

This section outlines tests to confirm the cluster ability to fail-over to the Passive Terracotta server, and fail back.

The test plan for testing fail over and fail back consists of the following tests:

| TestID | Test              | Setup  | Steps  | Expected Result   |
|--------|-------------------|--|--|---|
| TFO1   | Failover/Failback | L2-A is active, L2-B is passive. All systems are running and available to take traffic | 1. Run application 2. Kill -9 (or run stop-tc-server) on L2-A (Active) 3. After L2-B takes over as Active, start-tc-server on L2-A. (L2-A is now passive) 4. Kill -9 (or run stop-tc-server) on L2-B. (L2-A is now Active) | After first failover L2-A->L2-B, txns should continue. L2-A should come up cleanly in passive mode when tc-server is run. When second failover occurs L2-B->L2-A, L2-A should process txns. |

### Test Plan - Loss of Switch Tests

{tip} This test can only be run on a redundant network {tip}

This section outlines testing the loss of a switch in a redundant network, and confirming that no interrupt of service occurs.

The test plan for testing failure of a single switch consists of the following tests:

| TestID | Test             | Setup  | Steps   | Expected Result                            |
|--------|------------------|--|---|--|
| TSL1   | Loss of 1 Switch | 2 Switches in redundant configuration. L2-A is active, L2-B is passive. All systems are running and available to take traffic. | 1. Run application 2. Power down/pull plug moves to switch 2 with on Switch | All traffic transparently no interruptions |

### Test Plan - Loss of Network Connectivity

This section outlines testing the loss of network connectivity.

The test plan for testing failure of the network consists of the following tests:

## Terracotta Cluster Tests

| <b>TestID</b> | <b>Test</b>                  | <b>Setup</b>  | <b>Steps</b>                                       | <b>Expected Result</b>  |
|---------------|------------------------------|---|--|---|
| TNL1          | Loss of NIC wiring (Active)  | L2-A is active, L2-B is passive. All systems are running and available to traffic | 1. Run application 2. Remove Network Cable on L2-A | All traffic transparently moves to L2-B with no interruptions |
| TNL2          | Loss of NIC wiring (Passive) | L2-A is active, L2-B is passive. All systems are running and available to traffic | 1. Run application 2. Remove Network Cable on L2-B | No user impact on cluster                                     |

### Test Plan - Terracotta Cluster Failure

This section outlines the tests to confirm successful continued operations in the face Terracotta Cluster failures.

The test plan for testing Terracotta Cluster failures consists of the following tests:

| <b>TestID</b> | <b>Test</b>              | <b>Setup</b>  | <b>Steps</b>  | <b>Expected Result</b>                                     |
|---------------|--------------------------|---|---|--|
| TF1           | Process Failure Recovery | L2-A is active, L2-B is passive. All systems are running and available to traffic | 1. Run application 2. Bring down all L1s and L2s 3. Start L2s then L1s  | Cluster should come up and begin taking txns again         |
| TF2           | Server Failure Recovery  | L2-A is active, L2-B is passive. All systems are running and available to traffic | 1. Run application 2. Power down all machines 3. Start L2s and then L1s | Should be able to run application once all servers are up. |

### Client Failure Tests

This section outlines tests to confirm successful continued operations in the face of Terracotta client failures.

The test plan for testing Terracotta Client failures consists of the following tests:

| <b>TestID</b> | <b>Test</b>  | <b>Setup</b>  | <b>Steps</b>                        | <b>Expected Result</b>  |
|---------------|--------------|---|-------------------------------------|---|
| TCF1          | L1 Failure - | L2-A is active, L2-B is passive. 2 L1s L1-A and L1-B All systems are running and available to traffic | 1. Run application 2. kill -9 L1-A. | L1-B should take all incoming traffic. Some timeouts may occur due to txns in process when L1 fails over. |

# Cluster Security

## Introduction

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.

You can choose one of the following to secure servers:

- SSL-based security – Provides certificate-based authentication for all nodes (including clients) and secures the entire cluster with encrypted connections. Can be used with role-based authorization.
- LDAP-based authentication – Uses your organization's authentication database to secure access to Terracotta servers.

## Configure SSL-based Security

See the [advanced security page](#) to learn how to use Secure Sockets Layer (SSL) encryption and certificate-based authentication to secure enterprise versions of Terracotta clusters.

## Configure Security Using LDAP (via JAAS)

Lightweight Directory Access Protocol (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;  
};
```

Edit the values for `userProvider` (LDAP server), `authIdentity` (user identity), and `bindCredential` (encrypted password) to match the values for your environment.

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>
```

## Configure Security Using LDAP (via JAAS)

```
<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.

## User Roles

There are two roles available for Terracotta servers and clients:

- admin — The user with the "admin" role is the initial user who sets up security. Thereafter, the "admin" user can perform system functions such as shutting down servers, clearing or deleting caches and cache managers, and reloading configurations.
- terracotta — This is the operator role. The default username for the operator role is "terracotta". The "terracotta" user can connect to the TMC and access the read-only areas. In addition, the "terracotta" user can start a secure server. But a user must have the "admin" role in order to run the stop-tc-server script.

## 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.

# Securing Terracotta Clusters with SSL

## Introduction

Secure Sockets Layer (SSL) encryption and certificate-based authentication can be used to secure enterprise versions of Terracotta clusters.

Security in a Terracotta cluster includes both server-server connections and client-server connections. Note that all connections in a secured cluster must be secure; no insecure connections are allowed, and therefore security must be enabled globally in the cluster. It is not possible to enable security for certain nodes only.

## Overview

Security is set up using Terracotta configuration, tools provided in the Terracotta kit, standard Java tools, and public key infrastructure (via standard digital X.509 digital certificates).

## Security-Related Files

Each Terracotta server utilizes the following types of files to implement security:

- Java keystore – Contains the server's private key and public-key certificate. Protected by a keystore/certificate-entry password.
- Truststore – A keystore file containing only the certificates' public keys. Needed only if using self-signed certificates, not a Certificate Authority (CA).
- Keychain – Stores passwords, including to the server's keystore and to entries in these other files. The tools for creating and managing the Terracotta keychain file are provided with the Terracotta kit.
- Authorization – A .ini file with password-protected user accounts and their roles for servers and clients that connect to the server.

### TIP: Secure cacerts File

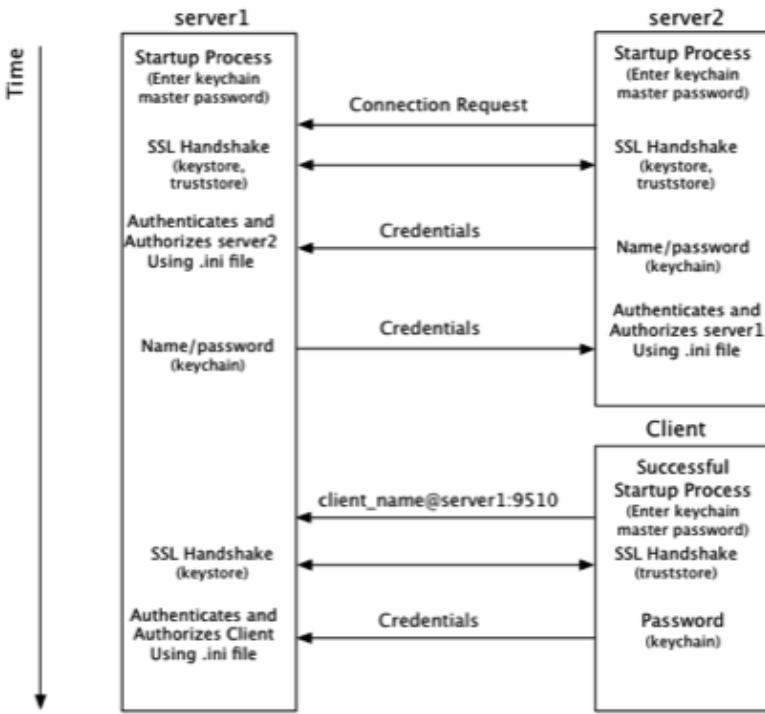
The standard Java cacerts file, located in  `${JAVA_HOME}/java.home/lib/security` by default, is a system-wide repository for CA root certificates included with the JDK. These certificates can play a part in certificate chains. [Java documentation](#) recommends that the cacerts file be protected by changing its default password and file permissions.

Each Terracotta client also has a keychain file that stores the password it uses to authenticate with the server.

All files are read on startup. Changes made to the files after startup cannot be read unless the cluster is restarted.

The following diagram illustrates the flow of security information—and that information's origin in terms of the files described above—during initial cluster connections:

## Security-Related Files



From a Terracotta server's point of view, security checks take place at startup and at the time a connection is made with another node on the cluster:

1. At startup, server1 requires a password to be entered directly from the console to complete its startup process.
2. A connection request from server2 initiates the process of establishing a secure connection using SSL.
3. Server1 authenticates server2 using stored credentials. Credentials are also associated with a role that authorizes server2. The process is symmetrical: server2 authenticates and authorizes server1.
4. A connection request from a Terracotta client initiates the process of establishing a secure connection using SSL.
5. Server1 authenticates and authorizes the client using stored credentials and associated roles. Because clients may be communicating with any active server in the cluster during their lifetimes, the client must be able to authenticate with any active server. Clients should be able to authenticate against *all* servers in the cluster, since active servers may fail over to mirror servers.

From a Terracotta client's point of view, security checks occur at the time the client attempts to connect to an active server in the cluster:

1. The client uses a server URI that includes the client username.

A typical (non-secure) URI is <server-address>:<port>. A URI that initiates a secure connection takes the form <client-username>@<server-address>:<port>.

2. A secure connection using SSL is established with the server.
3. The client sends a password fetched from a local keychain file. The password is associated with the client username.

The following configuration snippet is an example of how security could be set up for the servers in the illustration above:

## Setting Up Server Security

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
<servers secure="true">
    <server host="172.16.254.1" name="server1">
        ...
        <security>
            <ssl>
                <certificate>jks:server1alias@/the/path/keystore-file.jks</certificate>
            </ssl>
            <keychain>
                <class>com.terracotta.management.keychain.FileStoreKeyChain</class>
                <url>file:///%(user.dir)/server1keychain.tkc</url>
            </keychain>
            <auth>
                <realm>com.tc.net.core.security.ShiroIniRealm</realm>
                <url>file:///%(user.dir)/my_auth.ini</url>
                <user>server1username</user>
            </auth>
        </security>
    ...
</server>
    <server host="172.16.254.2" name="server2">
        ...
        <security>
            <ssl>
                <certificate>jks:server2alias@/the/path/keystore-file.jks</certificate>
            </ssl>
            <keychain>
                <class>com.terracotta.management.keychain.FileStoreKeyChain</class>
                <url>file:///%(user.dir)/server2keychain.tkc</url>
            </keychain>
            <auth>
                <realm>com.tc.net.core.security.ShiroIniRealm</realm>
                <url>file:///%(user.dir)/my_auth.ini</url>
                <user>server2username</user>
            </auth>
        </security>
    ...
</server>
...
</servers>
...
</tc:tc-config>
```

See the [configuration section](#) below for more information on the configuration elements in the example.

## Setting Up Server Security

To set up security on a Terracotta server, follow the steps in each of these procedures:

{toc-zone|3:3}

**NOTE:** Script names in the examples given below are for \*NIX systems. Equivalent scripts are available for Microsoft Windows in the same locations. Simply substitute the .bat extension for the .sh extension shown and convert path delimiters as appropriate.

Create the Server Certificate and Add It to the Keystore

## Create the Server Certificate and Add It to the Keystore

Each Terracotta server must have a keystore file containing a digital certificate and the associated private key. Your organization may already have policies and procedures in place regarding the generation and use of digital certificates and certificate chains, including the use of certificates signed by a CA.

**IMPORTANT SECURITY CONSIDERATION!:** Self-signed certificates might be less safe than CA-signed certificates because they lack third-party identity verification and do not carry a digital signature from an official CA. Your organization might already have policies and procedures in place regarding the generation and use of digital certificates and certificate chains, including the use of certificates signed by a Certificate Authority (CA). To follow your organization's policies and procedures regarding using digital certificates, you might need to adjust the procedures outlined in this document.

When used for a Terracotta server, the following conditions must be met for certificates and their keystores:

- The keystore must be a Java keystore (JKS) compatible with JDK 1.6 or higher.
- The certificate must be keyed with the alias named in the value of the `<certificate>` element of the [server's configuration](#).
- The Common Name (CN) field in the Distinguished Name must contain the hostname of the server, as configured in the [server's configuration](#).
- The password securing the certificate must match the keystore's main password. In other words, **the store password and key passwords must be identical**.
- When using a self-signed certificate (not one signed by a trusted CA), create a [custom truststore](#) for storing public keys.
- If security is not the most important consideration, you can set the following Java system properties as shown:
  - ◆ `-Dtc.ssl.trustAllCerts=true` (use any certificates)
  - ◆ `-Dtc.ssl.disableHostnameVerifier=true` (bypass hostname verification)

If you have a keystore in place, but the server certificate is not already stored in the keystore, you must import it into the keystore. If the keystore does not already exist, you must [create it](#).

### Creating Self-Signed Certificates Using Java Keytool

For testing purposes, or if you intend to use [self-signed certificates](#), use the Java keytool command to create the public-private key pair. You also use this command to create keystores and truststores, but note that keytool refers to truststores as "keystores" since there is only a logical difference.

#### Specifying a Custom Truststore

Note that if you are **not** using cacerts, the default Java truststore, the custom truststore must be specified with the `javax.net.ssl.trustStore` system property. In this case, you can choose to reset the custom truststore's default password with `javax.net.ssl.trustStorePassword`.

The following could be used to create both public-private keys (including a certificate) and a keystore file for the server called "server1" in the configuration example above:

```
keytool -genkey -keystore keystore-file.jks -dname "CN=172.16.254.1, OU=Terracotta, O=SAG, L=San
```

Note that the values passed to `-storepass` and `-keypass` match. Also, the field designating the Common Name (CN) must match the server's hostname, which matches the value entered in the server's configuration.

## Create the Server Certificate and Add It to the Keystore

This hostname can be an IP address or a resolvable domain name. If the `-dname` option is left out, a series of identity prompts (distinguished-name fields based on the X.500 standard) will appear before the server's entry is created in the keystore. The CN prompt appears as shown:

```
What is your first and last name?  
[Unknown]:
```

There are a number of other keytool options to consider, including `-keyalg` (cryptographic algorithm; default is DSA) and `-validity` (number of days until the certificate expires; default is 90). These and other options are dependent on your environment and security requirements. See the JDK documentation for more information on using the keytool.

Create a keystore and entry on each Terracotta server.

## Exporting and Importing Certificates

Each server should have a copy of each other server's public-key certificate in its truststore.

The following could be used to export the certificate of the server called "server1" in the configuration example above:

```
keytool -export -alias server1alias -keystore keystore-file.jks -file server1SelfSignedCert.cert
```

This "cert" file can now be used to import server1's certificate into the truststore of every other server. For example, to create a truststore and import server1's certificate on server2, copy the cert file to the working directory on server2 and use the following command:

```
keytool -import -alias server1alias -file server1SelfSignedCert.cert -keystore truststore.jks
```

After the password prompt, information about the certificate appears, and you are prompted to trust it. You must repeat this process until every server has a truststore containing the public-key certificate of every other server in the cluster.

### TIP: Use a Single Truststore

Instead of recreating the truststore on every server, create a single truststore containing every server's public key, then copy that to every server. This same truststore can also be used for [clients..](#)

## Set Up the Server Keychain

The keystore and each certificate entry are protected by passwords stored in the server keychain file. The location of the keychain file is specified in the value of the `<url>` element under the `<keychain>` element of the server's configuration file.

For example, with this [server configuration](#), when the server starts up, the keychain file would be searched for in the user's (process owner's) home directory. In the configuration example, a keychain file called `server1keychain.tkc` is searched for when server1 is started.

The keychain file should have the following entries:

- An entry for the local server's keystore entry.

## Set Up the Server Keychain

- An entry for every server that the local server will connect to.

Entries are created using the keychain script found in the Terracotta kit's `bin` directory.

### Creating an Entry for the Local Server

Create an entry for the local server's keystore password:

```
bin/keychain.sh <keychain-file> <certificate-URI>
```

where `<keychain-file>` is the file named in the server configuration's `<keychain>/<url>` element, and `<certificate-URI>` is the URI value in the server configuration's `<ssl>/<certificate>` element. **Note:** The `<certificate-URI>` must match the server configuration's `<ssl>/<certificate>` element exactly, including the path to the keystore.

If the keychain file does not exist, add the `-c` option to create it:

```
bin/keychain.sh -c <keychain-file> <certificate-URI>
```

You will be prompted for the keychain file's password, then for a password to associate with the URI. **For the URI, you must enter the same password used to secure the server's certificate in the keystore.**

For example, to create an entry for `server1` from the configuration example above, enter:

```
bin/keychain.sh server1keychain.tkc jks:server1alias@/the/path/keystore-file.jks  
Terracotta Management Console - Keychain Client  
Open the keychain by entering its master key: xxxxxxxx  
Enter the password you wish to associate with this URL: server1pass  
Confirm the password to associate with this URL: server1pass  
Password for jks:server1alias@/the/path/keystore-file.jks successfully stored
```

### Creating Entries for Remote Servers

Entries for remote servers have the format `tc://<user>@<host>:<group-port>`. Note that the value of `<user>` is specified in each server configuration's `<security>/<auth>/<user>` and is *not* related to the user running as process owner. If a value for `<security>/<auth>/<user>` is not specified, the username "terracotta" is used by default.

For example, to create an entry for `server2` in `server1`'s keychain, use:

```
bin/keychain.sh server1keychain.tkc tc://server2username@172.16.254.2:9530
```

If the keychain file does not exist, use the `-c` option:

```
bin/keychain.sh -c server1keychain.tkc tc://server2username@172.16.254.2:9530
```

You will be prompted for the keychain file's password, then for a password to associate with the entry `server2username@172.16.254.2:9530`.

An entry for `server1` must also be added to `server2`'s keychain:

```
bin/keychain.sh server2keychain.tkc tc://server1@172.16.254.1:9530
```

## Set Up Authentication

To use a copy of the same keychain file on every server, be sure to create an entry for the local server as well as every server that will connect to it.

## Set Up Authentication

Servers and clients that connect to a secured server must have authentication (usernames/passwords) and authorization (roles) defined. By default, this is set up using the usermanagement script, located in the Terracotta kit's `bin` directory. This script also creates the `.ini` file that contains the required usernames and roles (associated passwords are stored in the keychain file).

All nodes in a secured Terracotta cluster must have an entry in the server's `.ini` file:

- The local server itself
- All other servers
- All clients

Use the usermanagement script with the following format:

```
bin/usermanagement.sh -c <file> <username> terracotta
```

where `-c` is required only if the file does not already exist. For servers, the `<username>` will be used as the value configured in `<security>/<auth>/<user>`. For clients, the username must match the one used to start the client.

**Note:** While the "terracotta" role is appropriate for Terracotta servers and clients, the "admin" role is necessary for performing system functions such as stopping servers. For more information about roles, refer to the [User Roles](#) section.

For example:

```
# Create the .ini file and add a server username and role.  
bin/usermanagement.sh -c my_auth.ini server1username terracotta  
  
# Add another server.  
bin/usermanagement.sh my_auth.ini server2username terracotta  
  
# Add a client.  
bin/usermanagement.sh my_auth.ini client1username terracotta  
  
# Add a user with an admin (read/write) role.  
bin/usermanagement.sh my_auth.ini admin1username admin  
  
# Add a user with an operator (read) role.  
bin/usermanagement.sh my_auth.ini consoleusername operator
```

The correct Shiro realm must be specified in the [server configuration](#), along with the path to the `.ini` file:

```
...  
<auth>  
  <realm>com.tc.net.core.security.ShiroIniRealm</realm>  
  <url>file:///%(user.dir)/my_auth.ini</url>  
  <user>server1username</user>  
</auth>  
...
```

## Configure Server Security

Security for the Terracotta Server Array is configured in the Terracotta configuration file (`tc-config.xml` by default). The following is an example security configuration:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
...
<servers secure="true">
  <server host="172.16.254.1" name="server1">
    ...
    <security>
      <ssl>
        <certificate>jks:server1alias@/the/path/keystore-file.jks</certificate>
      </ssl>
      <keychain>
        <class>com.terracotta.management.keychain.FileStoreKeyChain</class>
        <url>file:///%(user.dir)/server1keychain.tkc</url>
      </keychain>
      <auth>
        <realm>com.tc.net.core.security.ShiroIniRealm</realm>
        <url>file:///%(user.dir)/my_auth.ini</url>
        <user>server1username</user>
      </auth>
    </security>
  ...
</server>
...
</servers>
...
</tc:tc-config>
```

Every server participating in an SSL-based secured cluster must have a `<security>` block wherein security-related information is encapsulated and defined. The keystore, keychain, and .ini files named in the configuration must be available to every server in the cluster.

The following table defines selected security-related elements and attributes shown in the configuration example.

| Name        | Definition   | Notes   |
|-------------|--|---|
| secure      | Attribute in <code>&lt;servers&gt;</code> element. Enables SSL security for the cluster.<br>DEFAULT: false.  | Enables/disables SSL-based security globally.                             |
| certificate | Element specifying the location of the server's authentication certificate and its containing keystore file. The format for the certificate-keystore location is <code>jks:alias@/path/to/keystore</code> . "alias" must match the value used to key the certificate in the keystore file. | Only the JKS type of keystore is supported.                               |
| class       | Element specifying the class defining the keychain file. If a class is not specified, <code>com.terracotta.management.keychain.FileStoreKeyChain</code> is used.   |   |
| url         | The URI for the keychain file (when under <code>&lt;keychain&gt;</code> ) or for the Realm configuration (.ini) file (when under <code>&lt;auth&gt;</code> ). These URIs are passed to the keychain or realm class to specify the keychain or authentication file, respectively.           | These files are created and managed with the <a href="#">keychain</a> and |

## Enabling SSL on Terracotta Clients

|             |  |                           |
|-------------|--|---------------------------|
|             |  | usermanagement scripts.   |
| user        | The username that represents this server and is authenticated by other servers.<br>This name is part of the credentials stored in the .ini file. | Defaults to "terracotta". |
| {/toc-zone} |  |                           |

# Enabling SSL on Terracotta Clients

Terracotta clients do not require any specific configuration to enable SSL connections to a Terracotta Server Array.

**NOTE:** Script names in the examples given below are for \*NIX systems. Equivalent scripts are available for Microsoft Windows in the same locations. Simply substitute the .bat extension for the .sh extension shown and convert path delimiters as appropriate.

To enable SSL security on the client, prepend the client username part to the Terracotta server URL the client uses to connect to the cluster. This should be the URL of an active server that has the client's credentials in its [.ini file](#). The format of this security URL is <client-username>@<host>:<dso-port>. Prepending the username automatically causes the client to initiate an SSL connection.

If the client has username `tcclient`, for example, and attempts to connect to the server in the configuration example, the URI would be:

```
tcclient@172.16.254.1:9510
```

This type of URL replaces the non-secure URL (<host>:<dso-port>) used to start clients in non-SSL clusters.

Both the client username and the corresponding password must match those in the [server's .ini file](#). The username is included in the URL, but the password must come from a [local keychain-file entry](#) that you create.

**NOTE:** If Terracotta servers are using self-signed certificates (not certificates signed by a well-known CA), then you must [specify a truststore for the client](#) which contains the public key of every server in the cluster.

## Create a Keychain Entry

The Terracotta client should have a keychain file with an entry for every Terracotta server in the cluster. The format for the entry uses the "tc" scheme:

```
tc://<client-username>@<host>:<dso-port>
```

An entry for the server in the example configuration should look like the following:

```
tc://client1@172.16.254.1:9510
```

Use the keychain script in the Terracotta kit to add the entry:

```
bin/keychain.sh clientKeychainFile tc://client1@172.16.254.1:9510
```

## Create a Keychain Entry

If the keychain file does not already exist, use the `-c` flag to create it:

```
bin/keychain.sh -c clientKeychainFile tc://client1@172.16.254.1:9510
```

If creating the keychain file, you will be prompted for a master password.

This entry in the client's keychain file serves as the key for the client's password, and will be provided to the server along with the client username ("client1" in the example). These credentials must match those in the [server's .ini file](#).

The Terracotta client searches for the keychain file in the following locations:

- `%(user.home)/.tc/mgmt/keychain`
- `%(user.dir)/keychain.tkc`
- The path specified by the system property `com.tc.security.keychain.url`

When you run the keychain script, the following prompt should appear:

```
Terracotta Management Console - Keychain Client
KeyChain file successfully created in clientKeychainFile
Open the keychain by entering its master key:
```

Enter the master key, then answer the prompts for the secret to be associated with the server URI:

```
Enter the password you wish to associate with this URL:
Password for tc://client1@172.16.254.1:9510 successfully stored
```

Note that the script does not verify the credentials or the server address.

## Using a Client Truststore

If Terracotta servers are using self-signed certificates (not certificates signed by a well-known CA), create a truststore on the client and import each server's public-key certificate into that truststore.

If you have already [created a truststore](#) for a server in the TSA, you can copy that file to each client after first importing that server's public-key certificate into the copy.

For the client to find the truststore, you must set the Java system property `javax.net.ssl.trustStore` to the location of the truststore file. In this case, note the existing secrets for opening the truststore and accessing each certificate.

**TIP: Changing the Truststore Password**

To change the existing truststore master password, use the Java system property `javax.net.ssl.trustStorePassword`.

## Restricting Clients to Specified Servers (Optional)

By default, clients are not restricted to authenticate a specific set of servers when responding to REST requests. However, it is possible to explicitly list the servers that a client may respond to by using the `<managementRESTService>` element's `securityServiceLocation` attribute in the Ehcache configuration.

## Restricting Clients to Specified Servers (Optional)

When this attribute is empty (or missing), no such restriction exists and the client will authenticate against any server in the cluster that meets the established security requirements. This is the recommended setting, as SSL connections and the authentication/authorization mechanism provide sufficient security.

In the case where an extra layer of security is required for the client's REST service, you can configure a list of allowed servers as follows:

```
<managementRESTService ...  
    securityServiceLocation=" https://my-l2-node1/tmc/api/assertIdentity ,  
    https://my-l2-node2/tmc/api/assertIdentity ">
```

where my-l2-node1 and my-l2-node2 are the servers' hostnames. Note, however, that any of the servers in a client's cluster can forward a REST request to that client at any time, and so all servers should be listed if this feature is used.

## Running a Secured Server

Start a server in a secure Terracotta cluster using the start-tc-server script as usual. After initial startup messages, you will be prompted for the password to the server's keychain. Once the password is entered, the server should continue its startup as normal.

## Confirm Security Enabled

You can confirm that a server in a number of ways:

- Look for the startup message `Security enabled, turning on SSL`.
- Search for log messages containing "SSL keystore", "HTTPS Authentication enabled", and "Security enabled, turning on SSL".
- Attempt to make JMX connections to the server—these should fail.

## Stopping a Secured Server

Stop a server in a secure Terracotta cluster using the stop-tc-server script with the following arguments:

- `-f <tc-config-file>` — A valid path to the self-signed certificate must have been specified in the server's configuration file.
- `-u <username>` — The user specified must have the "admin" role.
- `-w <password>`

For more information about the stop script, refer to [Start and Stop Server Scripts](#).

## Troubleshooting

You may encounter any of the following errors at startup:

### **RuntimeException: Couldn't access a Console instance to fetch the password from!**

This results from using "nohup" during startup. The startup process requires a console for reading password entry.

## Troubleshooting

### **TCRuntimeException: Couldn't create KeyChain instance ...**

The keychain file specified in the Terracotta configuration cannot be found. Check for the existence of the file at the location specified in <keychain>/<url>.

### **RuntimeException: Couldn't read from file ...**

This error appears just after an incorrect password is entered for the keychain file.

### **RuntimeException: No password available in keyChain for ...**

This error appears if no keychain password entry is found for the server's certificate. You must explicitly [store the certificate password](#) in the keychain file.

This error could also appear if the resolved hostname or IP address is different from the one in the keychain entry:

- tc://terracotta@localhost:9530 is the entry, but when the server configuration is read then localhost is resolved to an IP address. The entry searched for becomes tc://terracotta@<a.certain.ip.address>:9530.
- tc://terracotta@<a.certain.ip.address>:9530 is the entry, but when the server configuration is read then <a.certain.ip.address> is resolved to a host name. The entry searched for becomes tc://terracotta@my.host.com:9530.

## **Two Active Servers (Split Brain)**

Instead of an active-mirror 2-server stripe, both servers assert active status after being started. This error can be caused by the failure of the SSL handshake. An entry similar to the following may appear in the server log:

```
2013-05-17 12:10:24,805 [L2_L2:TCWorkerComm # 1_W] ERROR com.tc.net.core.TCConnection - SSL
```

For each server, ensure that all keychain entries are accurate, and that the required certificates are available from the appropriate truststores.

## **No Messages Indicating Security Enabled**

If servers start with no errors, but there are no messages indicating that security is enabled, ensure that the <servers> element contains secure="true".

# Changing Cluster Topology in a Live Cluster

## Introduction

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.

Servers that are part of the same server array but do not share the edited configuration file must have their configuration file edited and reloaded as shown below. Clients that do not load their configuration from the servers must have their configuration files edited to exactly match that of the servers.

**Note:** Changing the topology of a live cluster will not affect the distribution of data that is already loaded in the TSA. For example, if you added a stripe to a live cluster, the data in the server array would not be redistributed to utilize it. Instead, the new stripe could be used for adding new caches, while the original servers would continue to manage the original data.

## 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
```

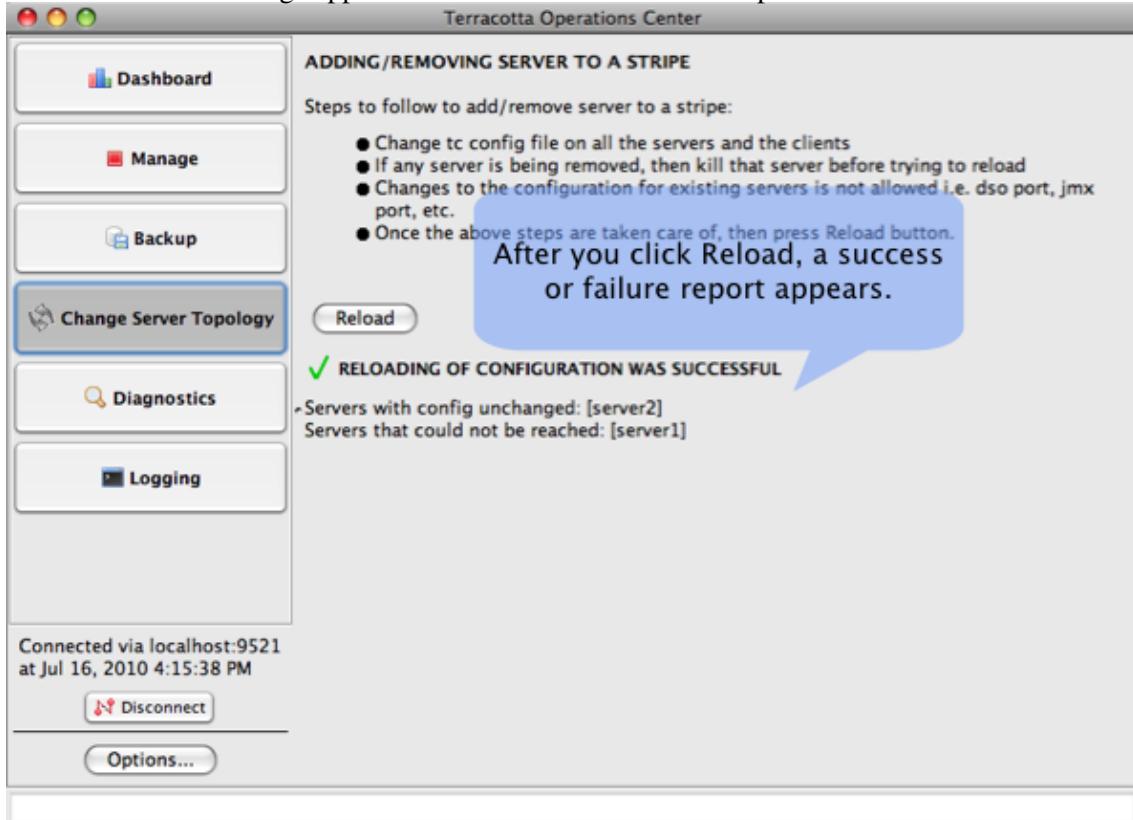
### MICROSOFT WINDOWS

```
%TERRACOTTA_HOME%\bin\ops-center.bat
```

4. After connecting to the cluster with the Operations Center, open the **Change Server Topology** panel.

## Adding a New Server

5. Click **Reload**. A message appears with the result of the reload operation.



6. Start the new server.

## 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 server 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 removed 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.

## Editing the Configuration of an Existing Server

If you edit the configuration of an existing ("live") 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.

**NOTE:** To be able to edit the configuration of an existing server, all clients must load their configuration from the Terracotta Server Array (TSA). Clients that load configuration from another source will fail to remain connected to the TSA due to a configuration mismatch.

# Terracotta Configuration Reference

## Introduction

This document is a reference to all of the Terracotta configuration elements found in the Terracotta configuration file. The Terracotta configuration file is named `tc-config.xml` by default.

## Configuration Schema

The documentation for the Terracotta configuration XML document can be found [here](#).

## Reference and Sample Configuration Files

The reference configuration is a sample configuration file with inline comments describing each configuration element. The Terracotta kit contains a reference configuration file under the `/config-samples` directory.

While this file serves as a good reference for the Terracotta configuration, do not use it as the basis for your Terracotta config. For that purpose, you should start with one of the sample configurations provided in the Terracotta kit. Typically the sample configurations are named `tc-config.xml` and can be found under the various samples provided with the Terracotta kit.

## Configuration Structure

The Terracotta configuration is an XML document that has these main sections: system, servers, clients. Each of these sections provides a number of configuration options relevant to its particular configuration topic.

## Configuration Variables

There are a few variables that will be interpolated by the configuration subsystem:

| Variable                        | Interpolated Value                     |
|---------------------------------|--|
| <code>%h</code>                 | The hostname                           |
| <code>%i</code>                 | The ip address                         |
| <code>%D</code>                 | Time stamp (yyyyMMddHHmmssSSS)         |
| <code>%(system property)</code> | The value of the given system property |

These variables will be interpolated in the following places:

- the "name", "host" and "bind" attributes of the `<server>` element
- the password file location for JMX authentication
- client logs location
- server logs location
- server data location

NOTE: Value of `%i`

## Configuration Variables

The variable `%i` is expanded into a value determined by the host's networking setup. In many cases that setup is in a `hosts` file containing mappings that may influence the value of `%i`.

## Overriding tc.properties

Every Terracotta installation has a default `tc.properties` file containing system properties. Normally, the settings in `tc.properties` are pre-tuned and should not be edited.

If tuning is required, you can override certain properties in `tc.properties` using `tc-config.xml`. This can make a production environment more efficient by allowing system properties to be pushed out to clients with `tc-config.xml`. Those system properties would normally have to be configured separately on each client.

### Setting System Properties in tc-config

To set a system property with the same value for all clients, you can add it to the Terracotta server's `tc-config.xml` file using a configuration element with the following format:

```
<property name="<tc_system_property>" value="<new_value>" />
```

All `<property />` tags must be wrapped in a `<tc-properties>` section placed at the beginning of `tc-config.xml`.

For example, to override the values of the system properties `l1.cachemanager.enabled` and `l1.cachemanager.leastCount`, add the following to the beginning of `tc-config.xml`:

```
<tc-properties>
  <property name="l1.cachemanager.enabled" value="false" />
  <property name="l1.cachemanager.leastCount" value="4" />
</tc-properties>
```

## Override Priority

System properties configured in `tc-config.xml` override the system properties in the default `tc.properties` file provided with the Terracotta kit. The default `tc.properties` file should *not* be edited or moved.

If you create a *local* `tc.properties` file in the Terracotta `lib` directory, system properties set in that file are used by Terracotta and will override system properties in the *default* `tc.properties` file. System properties in the local `tc.properties` file are *not* overridden by system properties configured in `tc-config.xml`.

System property values passed to Java using `-D` override all other configured values for that system property. In the example above, if `-Dcom.tc.l1.cachemanager.leastcount=5` was passed at the command line or through a script, it would override the value in `tc-config.xml` and `tc.properties`. The order of precedence is shown in the following list, with highest precedence shown last:

1. default `tc.properties`
2. `tc-config.xml`
3. local, or user-created `tc.properties` in Terracotta `lib` directory

## Override Priority

4. Java system properties set with -D

## Failure to Override

If system properties set in `tc-config.xml` fail to override default system properties, a warning is logged to the Terracotta logs. The warning has the following format:

```
The property <system_property_name> was set by local settings to <value>.  
This value will not be overridden to <value> from the tc-config.xml file.
```

System properties used early in the Terracotta initialization process may fail to be overridden. If this happens, a warning is logged to the Terracotta logs. The warning has the following format:

```
The property <system_property_name> was read before initialization completed.
```

The warning is followed by the value assigned to `<system_property_name>`.

The property `tc.management.mbeans.enabled` is known to load before initialization completes and cannot be overridden.

## System Configuration Section

### /tc:tc-config/system/configuration-model

The configuration-model element is for informational purposes. The two configuration-model options are '*development*' and '*production*'. These values have no effect on the functioning of Terracotta servers or clients, but instead allow you to designate the intended use of a configuration file using one of two recommended modes.

In development, you may want each client might have its own configuration, independent of the server or any other client. This approach is useful for development, but should not be used in production as it can result in unpredictable behavior should conflicts arise. To note that a configuration file is intended for direct use by a client, set the configuration-model to '*development*'.

In production, each client should obtain its configuration from a Terracotta server instance. To note that a configuration file is intended to be fetched from a server, set the configuration-model to '*production*'.

In general, a client can specify that its configuration come from a server by setting the `tc.config` system property:

```
-Dtc.config=serverHost:dsoPort
```

## Servers Configuration Section

### /tc:tc-config/servers

This section defines the Terracotta server instances present in your cluster. One or more entries can be defined. If this section is omitted, Terracotta configuration behaves as if there's a single server instance with default values.

## /tc:tc-config/servers

The attribute `secure` is a global control for enabling ("true") or disabling ("false" DEFAULT) [SSL-based security](#) for the entire cluster.

Each Terracotta server instance needs to know which configuration it should use as it starts up. If the server's configured name is the same as the hostname of the host it runs on and no host contains more than one server instance, then configuration is found automatically.

For more information on how to use the Terracotta configuration file with servers, see the [Terracotta configuration guide](#).

## /tc:tc-config/servers/server

A server stanza encapsulates the configuration for a Terracotta server instance. The `server` element takes three optional attributes (see table below).

| Attribute         | Definition  | Value  | Default Value             |
|-------------------|---|--|---------------------------|
| <code>host</code> | The ID of the machine hosting the Terracotta server   | Host machine's IP address or resolvable hostname | Host machine's IP address |
| <code>name</code> | The ID of the Terracotta server; can be passed to Terracotta scripts such as <code>start-tc-server</code> using <code>n &lt;name&gt;</code> | user-defined string                              | :                         |
| <code>bind</code> | The network interface on which the Terracotta server listens for Terracotta clients; 0.0.0.0 specifies all interfaces                       | interface's IP address                           | 0.0.0.0                   |

*Sample configuration snippet:*

```
<server>
  <!-- my host is '%i', my name is '%i:dso-port', my bind is 0.0.0.0 -->
  ...
</server>
<server host="myhostname">
  <!-- my host is 'myhostname', my name is 'myhostname:dso-port', my bind is 0.0.0.0 -->
  ...
</server>
<server host="myotherhostname" name="server1" bind="192.168.1.27">
  <!-- my host is 'myotherhostname', my name is 'server1', my bind is 192.168.1.27 -->
  ...
</server>
```

## /tc:tc-config/servers/server/data

This section lets you declare where the server should store its data.

*Sample configuration snippet:*

```
<!-- Where should the server store its persistent data? (This includes stored object data for DSO.) This value undergoes parameter substitution before being used; this allows you to use placeholders like '%h' (for the hostname) or '%(com.mycompany.propname)' (to substitute in the value of Java system property 'com.mycompany.propname'). Thus, a value of 'server-data-%h' would expand to 'server-data-artichoke' if running on host 'artichoke'. -->
```

## /tc:tc-config/servers/server/data

If this is a relative path, then it is interpreted relative to the location of this file. It is thus recommended that you specify an absolute path here.

Default: 'data'; this places the 'data' directory in the same directory as this config file.

-->  
<data>/opt/terracotta/server-data</data>

## /tc:tc-config/servers/server/logs

This section lets you declare where the server should write its logs.

*Sample configuration snippet:*

```
<!-- In which directory should the server store its log files? Again,  
this value undergoes parameter substitution before being used;  
thus, a value like 'server-logs-%h' would expand to  
'server-logs-artichoke' if running on host 'artichoke'. See the  
Product Guide for more details.
```

If this is a relative path, then it is interpreted relative to the location of this file. It is thus recommended that you specify an absolute path here.

Default: 'logs'; this places the 'logs' directory in the same directory as this config file.

-->  
<logs>/opt/terracotta/server-logs</logs>

You can also specify stderr : or stdout : as the output destination for log messages. For example:

```
<logs>stdout:</logs>
```

To set the logging level, see [this FAQ entry](#).

## /tc:tc-config/servers/server/statistics

This section lets you declare where the server should store buffered statistics data.

*Sample configuration snippet:*

```
<!-- In which directory should the server store statistics  
data that is being buffered? Again, this value undergoes  
parameter substitution before being used; thus, a value  
like 'statistics-data-%h' would expand to 'statistics-data'  
if running on host 'artichoke'. See the Product Guide for  
more details.
```

If this is a relative path, then it is interpreted relative to the current working directory of the server (that is, the directory you were in when you started server). It is thus recommended that you specify an absolute path here.

Default: 'statistics'; this places the 'statistics' directory in the directory you were in when you invoked 'start-tc-server'.

-->

```
/tc:tc-config/servers/server/statistics  
<statistics>/opt/terracotta/server-statistics</statistics>
```

## **/tc:tc-config/servers/server/dso-port**

This section lets you set the port that the Terracotta server listens to. It is called 'dso-port' for historical reasons—it is not related to Distributed Shared Objects.

The default value of 'dso-port' is 9510.

*Sample configuration snippet:*

```
<dso-port>9510</dso-port>
```

## **/tc:tc-config/servers/server/jmx-port**

This section lets you set the port that the Terracotta server's JMX Connector listens to.

The default value of 'jmx-port' is 9520.

*Sample configuration snippet:*

```
<jmx-port>9520</jmx-port>
```

## **/tc:tc-config/servers/server/l2-group-port**

This section lets you set the port that the Terracotta server uses to communicate with other Terracotta servers when the servers are run in network-based active-passive mode.

The default value of 'l2-group-port' is 9530.

*Sample configuration snippet:*

```
<l2-group-port>9530</l2-group-port>
```

## **/tc:tc-config/servers/server/security**

This section contains the data necessary for running a secure cluster based on SSL, digital certificates, and node authentication and authorization.

See the [advanced security page](#) for a configuration example.

### **/tc:tc-config/servers/server/security/ssl/certificate**

The element specifying certificate entry and location of the certificate store. The format is:

```
<store-type>:<certificate-alias>@</path/to/keystore.file>
```

The Java Keystore (JKS) type is supported by Terracotta 3.7 and higher.

/tc:tc-config/servers/server/security

## **/tc:tc-config/servers/server/security/keychain**

This element contains the following subelements:

- <class> – Element specifying the class defining the keychain file. If a class is not specified, com.terracotta.management.keychain.FileStoreKeyChain is used.
- <url> – The URI for the keychain file. It is passed to the keychain class to specify the keychain file.

## **/tc:tc-config/servers/server/security/auth**

This element contains the following subelements:

- <realm> – Element specifying the class defining the security realm. If a class is not specified, com.tc.net.core.security.ShiroIniRealm is used.
- <url> – The URI for the Realm configuration (.ini) file. It is passed to the realm class to specify authentication file.
- <user> – The username that represents the server and is authenticated by other servers. This name is part of the credentials stored in the .ini file.

## **/tc:tc-config/servers/server/authentication**

In this section, you can configure security using the Lightweight Directory Access Protocol (LDAP) or JMX authentication. Enabling one of these methods causes a Terracotta server to require credentials before allowing a JMX connection to proceed.

For more information on how to configure authentication, including secure SSL-based connections, see [Terracotta Cluster Security](#).

## **/tc:tc-config/servers/server/http-authentication/user-realm-file**

Turn on authentication for the embedded Terracotta HTTP Server. This requires a properties file that contains the users and passwords that have access to the HTTP server.

The format of the properties file is:

```
username: password [,rolename ...]
```

The supported roles and protected sections are: \* statistics (for the statistics gatherer at /statistics-gatherer.)

Passwords may be clear text, obfuscated or checksummed. The class com.mortbay.Util.Password should be used to generate obfuscated passwords or password checksums.

By default, HTTP authentication is turned off.

*Sample configuration snippet:*

```
<http-authentication>
  <user-realm-file>/opt/terracotta/realm.properties</user-realm-file>
</http-authentication>
```

/tc:tc-config/servers/server/dso

## **/tc:tc-config/servers/server/dso**

This section contains configuration specific to the functioning of the Terracotta server. It is called 'dso' for historical reasons—it is not related to Distributed Shared Objects.

### **/tc:tc-config/servers/server/dso/client-reconnect-window**

This section lets you declare the client reconnect-time window. The value is specified in seconds and the default is 120 seconds. If adjusting the value in <client-reconnect-window>, note that a too-short reconnection window can lead to unsuccessful reconnections during failure recovery, and a too-long window lowers the efficiency of the network.

*Sample configuration snippet:*

```
<client-reconnect-window>120</client-reconnect-window>
```

*Further reading:* For more information on client and server reconnection is executed in a Terracotta cluster, and on tuning reconnection properties in a high-availability environment, see [Configuring and Testing Terracotta For High Availability](#).

### **/tc:tc-config/servers/server/dso/persistence**

This section lets you configure whether Terracotta operates in persistent (permanent-store) or non-persistent mode (temporary-swap-only).

Temporary-swap-only mode is the default mode. This mode uses the disk as a temporary backing store. Data is not preserved across server restarts and cluster failures, although shared in-memory data may survive if a backup Terracotta server is set up. This mode maximizes performance and should be used where restoring data after failures is not dependent on the server.

Permanent-store mode backs up all shared in-memory data to disk. This backed-up data survives server restarts and cluster failures, allowing all application state to be restored. This mode is recommended for ensuring a more robust failover architecture, but will not perform as well as temporary-swap-mode.

*Further reading:* For more information on Terracotta cluster persistence and restarting servers, see the [architecture guide](#).

*Sample configuration snippet:*

```
<persistence>
  <!--
    Default: 'temporary-swap-only'
  -->
  <mode>permanent-store</mode>
</persistence>
```

### **/tc:tc-config/servers/server/dso/garbage-collection**

This section lets you configure the periodic distributed garbage collector (DGC) that runs in the Terracotta server.

## /tc:tc-config/servers/server/dso/garbage-collection

*Further reading:*

- [TSA Architecture](#) – More on how DGC functions.
- [Terracotta Tools Catalog](#) – See the section on the run-dgc shell script.
- [Terracotta Developer Console](#) - How to execute DGC operations and view current status, history, and statistics

*Configuration snippet:*

```
<garbage-collection>

    <!-- If 'true', distributed garbage collection is enabled.
        You should only set this to 'false' if you are
        absolutely certain that none of the data underneath
        your roots will ever become garbage; certain
        applications, such as those that read a large amount
        of data into a Map and never remove it (merely look up
        values in it) can safely do this.

        Default: true
    -->
    <enabled>true</enabled>

    <!-- If 'true', the DSO server will emit extra information when
        it performs distributed garbage collection; this can
        be useful when trying to performance-tune your
        application.

        Default: false
    -->
    <verbose>false</verbose>

    <!-- How often should the DSO server perform distributed
        garbage collection, in seconds?

        Default: 3600 (60 minutes)
    -->
    <interval>3600</interval>
</garbage-collection>
```

## /tc:tc-config/servers/ha

Use this section to indicate properties associated with running your servers in active-passive mode. The properties apply to all servers defined. You can omit this section, in which case your servers, running in persistent mode, will run in networked active-passive mode.

To allow for at most 1 `<ha>` section to be defined along with multiple `<server>` sections, define multiple `<server>` sections followed by one `<ha>` section (or no `<ha>` section).

*Sample configuration snippet:*

```
<servers>
    <server host="%i" name="sample1">
    </server>
    <server host="%i" name="sample2">
    </server>
```

## /tc:tc-config/servers/ha

```
<ha>
  <mode>networked-active-passive</mode>
</ha>
</servers>
```

## /tc:tc-config/servers/ha mode

This section allows you to configure whether servers run in network-based (default) or disk-based active-passive High Availability (HA) mode. Network-based servers have separate databases and sync over the network. Disk-based servers share the database, which is locked by the active server. Disk-based HA is appropriate only in special use-cases.

There are two corresponding value options: 'networked-active-passive' and 'disk-based-active-passive'.

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.

*Sample configuration snippet:*

```
<mode>networked-active-passive</mode>
```

## /tc:tc-config/servers/ha/networked-active-passive

This section allows you to declare the election time window, which is used when servers run in network-based active-passive mode. An active server is elected from the servers that cast a vote within this window. The value is specified in seconds and the default is 5 seconds. Network latency and work load of the servers should be taken into consideration when choosing an appropriate window.

*Sample configuration snippet:*

```
<networked-active-passive>
  <election-time>5</election-time>
</networked-active-passive>
```

## /tc:tc-config/servers/mirror-groups

Use the `<mirror-groups>` section to bind groups of Terracotta server instances into a server array. The server array is built from sets of Terracotta server instances with one active server instance and one or more mirrors ("hot standbys" or back-ups).

The following table summarizes the elements contained in a `<mirror-groups>` section.

| Element                            | Definition   | Attributes  |
|------------------------------------|--|---|
| <code>&lt;mirror-groups&gt;</code> | Encapsulates any number of <code>&lt;mirror-group&gt;</code> sections. Only one <code>&lt;mirror-groups&gt;</code> section can exist in a Terracotta configuration file. | None  |
| <code>&lt;mirror-group&gt;</code>  | Encapsulates one <code>&lt;members&gt;</code> element and one <code>&lt;ha&gt;</code> element.   | <code>&lt;group-name&gt;</code> can be assigned a unique non-empty string value. If not set, Terracotta automatically creates a unique name for the mirror group. |

## /tc:tc-config/servers/mirror-groups

|           |   |      |
|-----------|---|------|
| <members> | Encapsulates <member> elements. Only one <members> section can exist in a <mirror-group> section.   | None |
| <ha>      | Applies high-availability settings to the encapsulating mirror group only. Settings are the same as those available in the <a href="#">main high-availability section</a> . Mirror groups without an <ha> section take their high-availability settings from the <a href="#">main high-availability section</a> . | None |
| <member>  | Contains the value of the server-instance name attribute configured in a <server> element. Each <member> element represents a server instance assigned to the mirror group.   | None |

For examples and more information, see the [Terracotta Configuration Guide](#).

## Clients Configuration Section

The clients section contains configuration about how clients should behave.

### /tc:tc-config/clients/logs

This section lets you configure where the Terracotta client writes its logs.

*Sample configuration snippet:*

```
<!--
This value undergoes parameter substitution before being used;
thus, a value like 'client-logs-%h' would expand to
'client-logs-banana' if running on host 'banana'. See the
Product Guide for more details.

If this is a relative path, then it is interpreted relative to
the current working directory of the client (that is, the directory
you were in when you started the program that uses Terracotta
services). It is thus recommended that you specify an absolute
path here.

Default: 'logs-%i'; this places the logs in a directory relative
to the directory you were in when you invoked the program that uses
Terracotta services (your client), and calls that directory, for example,
'logs-10.0.0.57' if the machine that the client is on has assigned IP
address 10.0.0.57.
-->
<logs>logs-%i</logs>
```

You can also specify `stderr:` or `stdout:` as the output destination for log messages. For example:

```
<logs>stdout:</logs>
```

To set the logging level, see [this FAQ entry](#).

/tc:tc-config/clients/dso/fault-count

## **/tc:tc-config/clients/dso/fault-count**

Sets the object fault count. Fault count is the maximum number of reachable objects that are pre-fetched from the Terracotta server to the Terracotta client when an object is faulted from that server to that client.

*Pre-fetching* an object does not fault the entire object, only the minimum amount of metadata needed to construct the object on the client if necessary. Unused pre-fetched objects are eventually cleared from the client heap.

Objects qualify for pre-fetching either by being referenced by the original object (being part of original object's object graph) or because they are peers. An example of peers are primitives found in the same map.

In most applications, pre-fetching improves locality of reference, which benefits performance. But in some applications, for example a queue fed by producer nodes that shouldn't pre-fetch objects, it may be advantageous to set the fault count to 0.

*Sample configuration snippet:*

```
<fault-count>500</fault-count>
```

*Default:* 500

## **/tc:tc-config/clients/dso/debugging**

The client debugging options give you control over various logging output options to gain more visibility into what Terracotta is doing at runtime.

### **/tc:tc-config/clients/dso/debugging/runtime-logging/new-object-debug**

When set to true, the addition of every new clustered object is logged.

*Default:* false

### **/tc:tc-config/clients/dso/debugging/runtime-output-options**

The runtime output options are modifiers to the runtime logging options

#### **/tc:tc-config/clients/dso/debugging/runtime-output-options/caller**

When set to true, this option logs the call site in addition to the runtime output.

*Default:* false

#### **/tc:tc-config/clients/dso/debugging/runtime-output-options/full-stack**

When set to true, this option provides a full stack trace of the call site in addition to the runtime output.

*Default:* false

# Terracotta Developer Console

## Introduction

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.

## 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

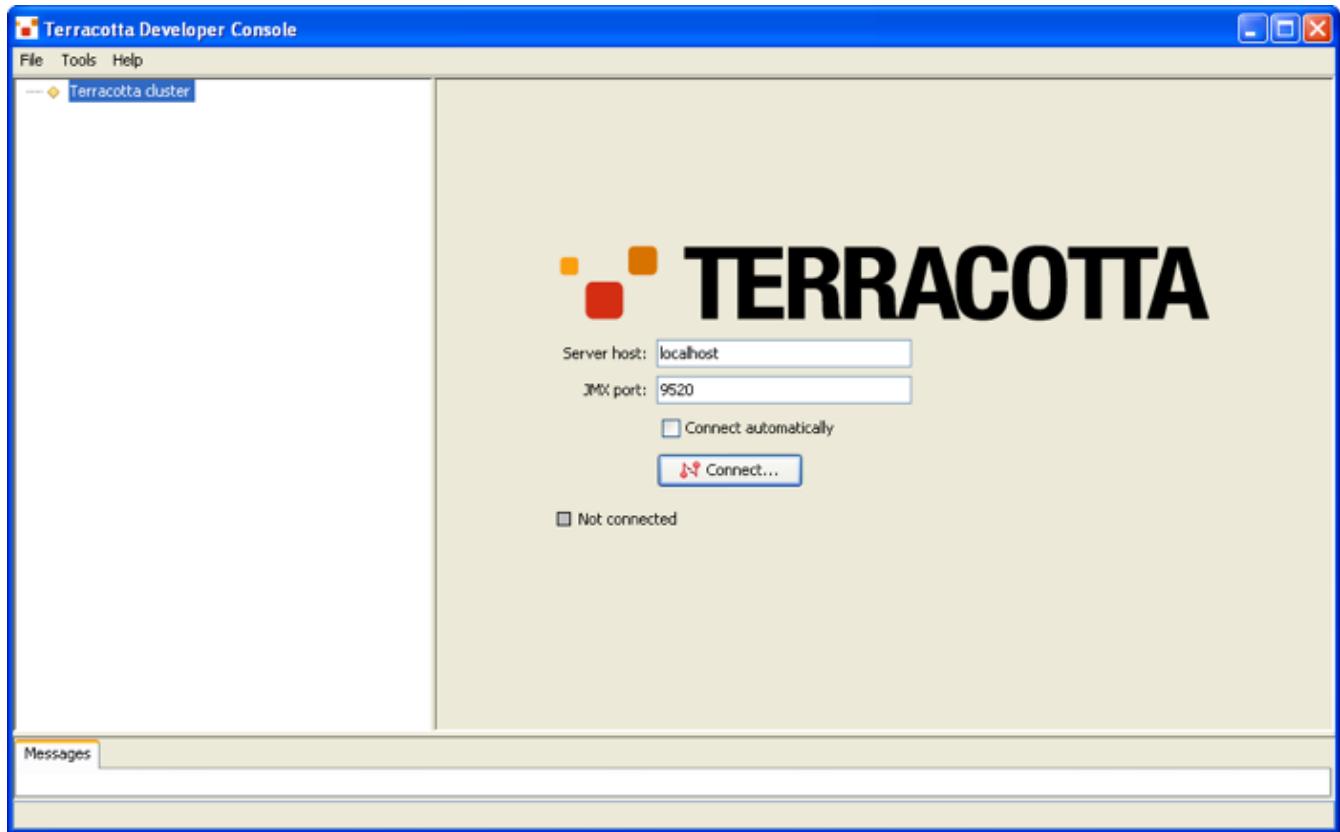
```
[PROMPT] %TERRACOTTA_HOME%\bin\dev-console.bat
```

#### 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.

## 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.

A cluster node is created with a default name ("Terracotta cluster"). However, you can rename a cluster node by choosing **Rename** from its context menu.

## Console Messages

Click **Messages** in the Status panel to view messages from the console about its operations.

## 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

- 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.

## 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.

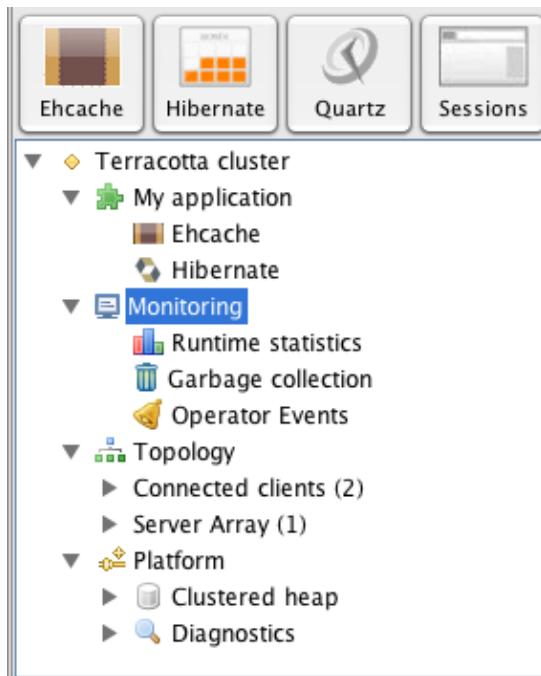
## Context Menus

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.

## 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.

## Working with Clusters



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.

## Adding and Removing Clusters

To add a new cluster reference, choose **New cluster** from the **File** or context menu.

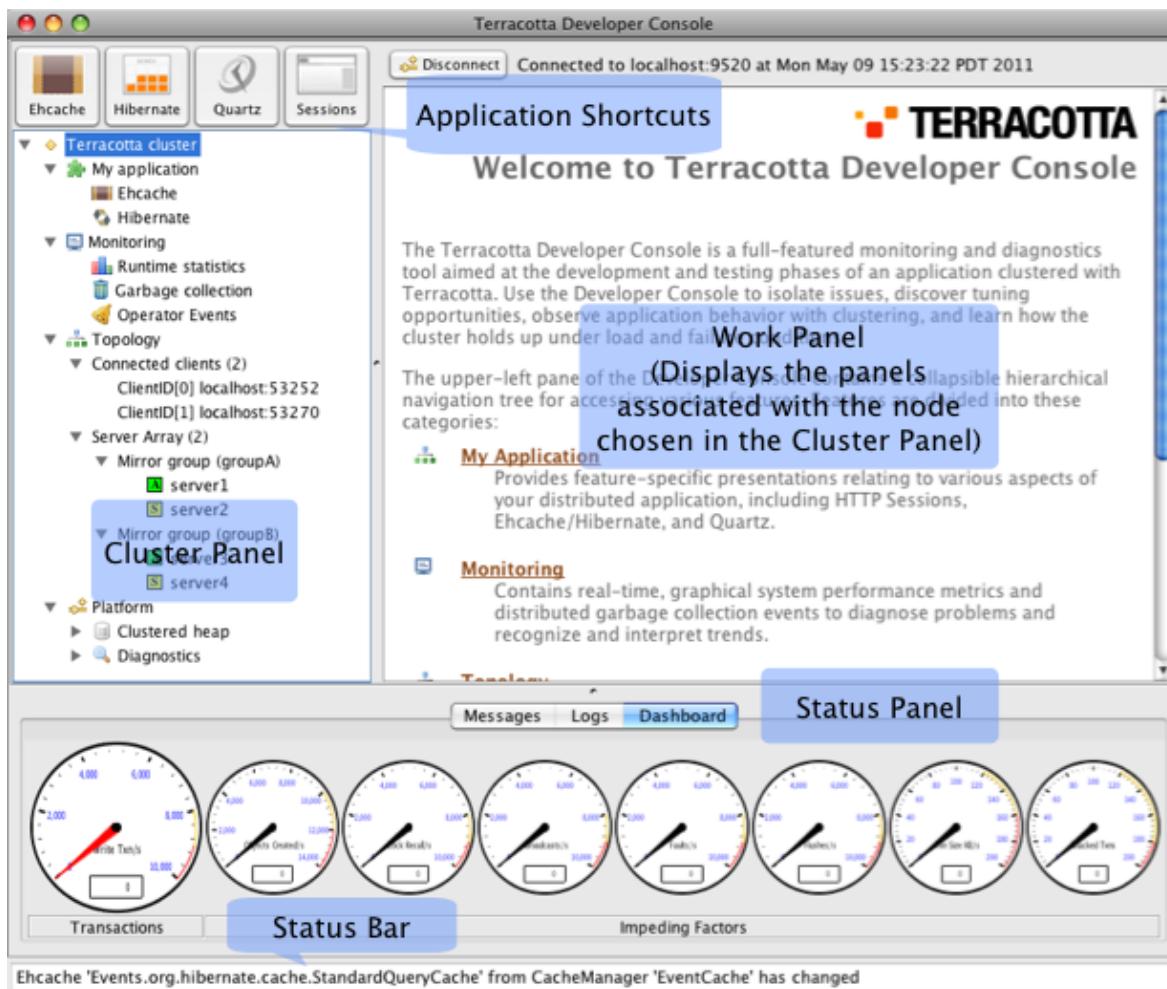
The cluster topology is determined from the server specified in the connection panel's **Server Host** and **JMX Port** fields. These fields are editable when the console is not connected to the cluster.

To remove an existing cluster reference, right-click the cluster in the cluster list to open the context menu, then choose **Delete**.

## Connecting to a cluster

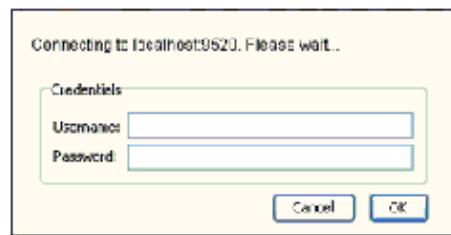
To connect to an existing cluster, select the cluster node in the cluster list, then click the **Connect** button in the connect/disconnect panel. You can also connect to a specific cluster by choosing **Connect** from its context menu. After a successful connection, the cluster node becomes expandable and a connection message appears in the status bar.

## Connecting to a cluster



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.

## 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.

- Note that the flag `-s` must be passed to the launch script.
- Be aware of the [security issue with self-signed certificates](#).

For instructions on how to secure your Terracotta cluster for JMX, see [Cluster Security](#).

Disconnecting from a Cluster

## Disconnecting from a Cluster

To disconnect from a cluster, select that cluster's node in the clusters panel and either click the **Disconnect** button above the help panel or select **Disconnect** from the cluster context menu.

## 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.

### 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.

The screenshot shows the Terracotta Developer Console interface. At the top, a header bar displays "Cache Manager: ColorCache" and "has 2 clustered instances containing 1 total cache instances". Below this is a toolbar with buttons for "Overview", "Performance", "Statistics", "Manage Active Caches...", "Cache BulkLoading...", "Cache Statistics...", "Clear Cache Contents...", and "Cache Configuration...". A "View by:" dropdown is set to "CacheManager Instances".

The main area is titled "Instances of CacheManager ColorCache" and contains a table:

| Node             | Caches | Enabled | BulkLoad | Statistics |
|------------------|--------|---------|----------|------------|
| 10.2.0.133:57925 | 1      | 1       | 0        | 1          |
| 10.2.0.133:57936 | 1      | 1       | 0        | 1          |

At the bottom right of this section is a checkbox labeled "Select/De-select All".

Below this is a "Summary of CacheManager ColorCache on node 10.2.0.133:57936" table:

| Cache  | Terracotta-clustered | Enabled | BulkLoad | Consistency | Statistics |
|--------|----------------------|---------|----------|-------------|------------|
| colors | ✓                    | ✓       | ✗        | STRONG      | ✓          |

## Overview Panel

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.
- Mode – Indicates whether the cache is in bulk-load mode ("bulk loading") or not ("normal"). Caches that cannot be put into bulk-loading mode display "na". For more on bulk loading, see [Bulk-Load API](#).
- Consistency – Indicates what mode of data consistency the cache is in (or "na" if no consistency is in effect). Available modes are STRONG and EVENTUAL. For more on cache consistency, see [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.
- Pinned – Indicates the type of cache pinning in effect (or "na" if no pinning is in effect). For more information on pinning, see [this page](#).

### 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: ... 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.

## Overview Panel

### 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.

## Overview Panel

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

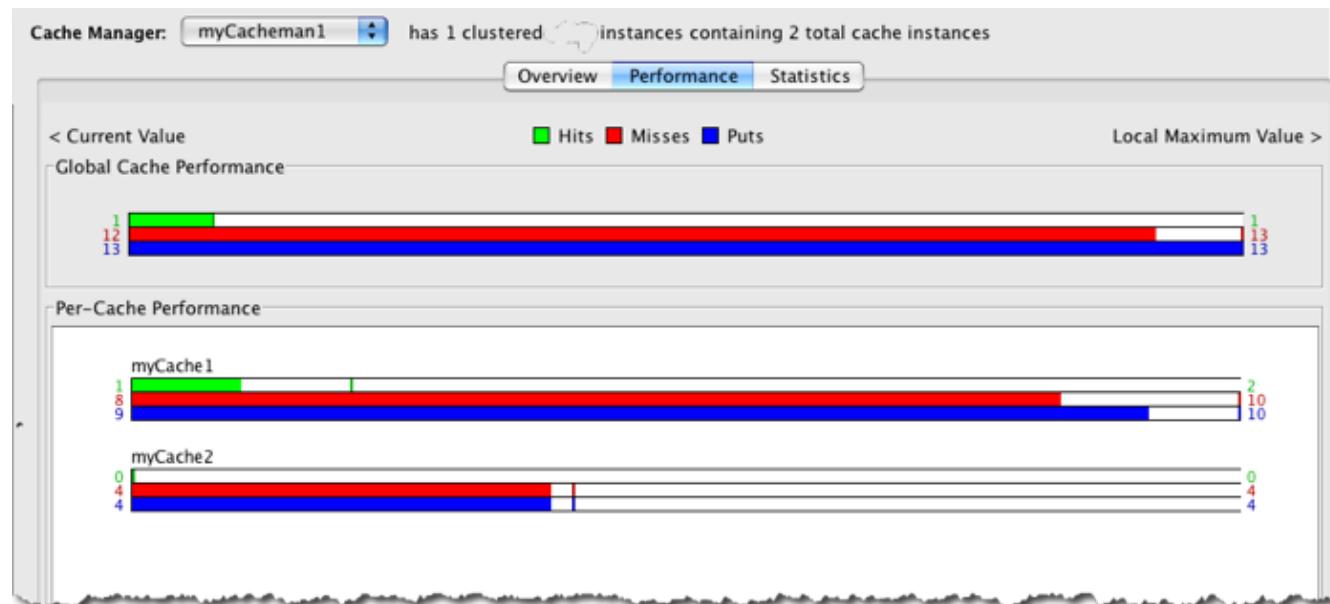
Click **Cache Configuration** to open a dialog with editable cache configuration. See [Editing Cache Configuration](#) for more information on editing cache configurations in the Terracotta Developer Console.

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.

## 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 Panel

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 [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.

## 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).
- View statistics for specific caches cluster-wide or on selected clients.

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:

## Statistics Panel

- 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. Or expand either of those menus and choose a cache to view. To navigate long cache lists, highlight the list and type the first letter of the cache's name.
- 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.

## Statistics Panel

- 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.

You can change the type of statistic a graph displays by opening its context menu, then choosing **Change chart to....** Choices include CacheInMemoryHitRate and CacheAverageGetTime.

## Cache Statistics Table

The cache statistics table displays a snapshot of statistics for each cache, including the following:

- 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.

To display additional (or remove current) cache statistics:

1. Click **Customize Columns....**
2. Select statistics to add to the table, or clear checkboxes to remove statistics from the table.

To add or remove all statistics, select or clear **Select/De-select**. Float the mouse pointer over a statistic name to view its definition.

3. Click **OK** to save and apply changes, or click **Cancel** to leave the table as is.

Click **Reset to defaults** to return the original statistics to the table.

## 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

## Statistics Panel



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.

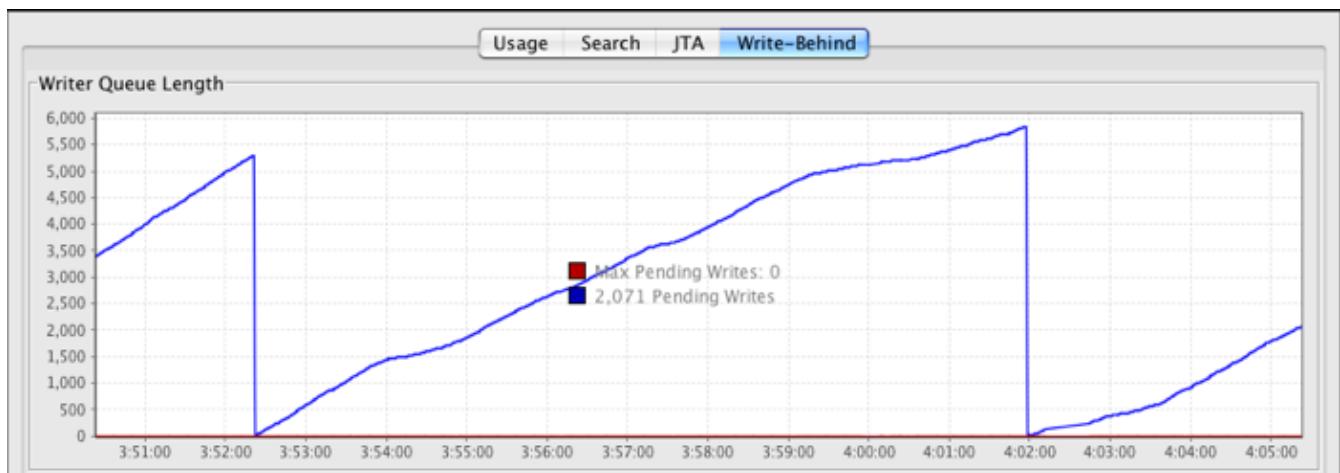
## Cache Statistics JTA Graphs



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 [Working With Transactional Caches](#).

## Cache Statistics Write-Behind Graphs

## Sizing Panel



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 [Write-Behind Queue in Enterprise Ehcache](#).

## Sizing Panel

The **Sizing** panel displays graphical resource-tier usage statistics for the current CacheManager and caches. Only the tiers available to the CacheManager and its caches are displayed. For example, if BigMemory is available for Ehcache, local offheap statistics are shown.

The available tiers are determined by the CacheManager and cache configuration. Possible tiers are:

- Local Heap – Local client on-heap memory.
- Local OffHeap – Local client off-heap memory, available with BigMemory for Ehcache.
- Local Disk – For versions prior to 4.x, refers either to the local client disk store of a non-clustered cache or to Terracotta storage of a clustered cache.
- Remote – Storage on the Terracotta Server Array. This is a combination of on-heap, off-heap (available with BigMemory for the Terracotta Server Array), and server disk store. Only caches that *are* distributed can utilize this tier.

### NOTE: Terracotta Server Array Sizing Statistics

The \*\*Remote\*\* tier reflects the amount of CacheManager and cache data stored on the Terracotta Server Array. This statistic is a client-side view and therefore is only estimate. For more accurate statistics on the Terracotta Server Array, use the [Monitoring panels](#).

The display is automatically refreshed every 30 seconds or manually by clicking the **Refresh** button.

### CacheManager Utilization by Tier

A set of bar graphs showing the selected CacheManager's current usage of each tier. A red vertical line marks the limit at that amount allotted for that tier. A message above the graph indicates whether the CacheManager is configured with all size-based caches ("size-base pooling") or mixed caches (only size-based caches are displayed).

## Sizing Panel

### CacheManager Relative Cache Sizes

Gives a pie-chart of tier usage by cache. Select the tier to display from the **Tier** menu.

A table provides the following information for each cache:

- Cache – The name of the cache.
- Size in Bytes – The size of the cache on the selected tier.
- % of Used – Percentage taken up by the cache of the portion of the tier that is being used by all of the caches.
- Entries - The number of cache entries on the selected tier.
- Mean Value Size – The estimated size of each cache entry on the selected tier.

### Cache Utilization by Tier

A set of bar graphs showing the selected cache's current usage of each tier. A red vertical line marks the limit at that amount allotted for that tier. A message above the graph indicates whether the cache takes its size configuration from the CacheManager (pool-base) or is configured with its own sizing (size-based).

### Cache Misses by Tier

A set of bar graphs showing the total cache misses in each tier for the selected cache.

## 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

## Editing Cache Configuration

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](#).

## 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

## Enterprise Ehcache for Hibernate Applications

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

The screenshot shows the Terracotta Developer Console interface with the following details:

- Persistence Unit:** EventCache
- Select Feature:** Hibernate
- Second-Level Cache:** (disabled)
- Hibernate Statistics:** Current View Type: Aggregate View, Select View: Cluster stats
- Entity Statistics Table:**

| Name          | Loads | Updates | Inserts | Deletes | Fetches | Optimistic Failures |
|---------------|-------|---------|---------|---------|---------|---------------------|
| o.h.t.d.Event | 4     | 0       | 2       | 0       | 0       | 0                   |
- Buttons:** Refresh, Clear Stats

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

## Hibernate View

- 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)

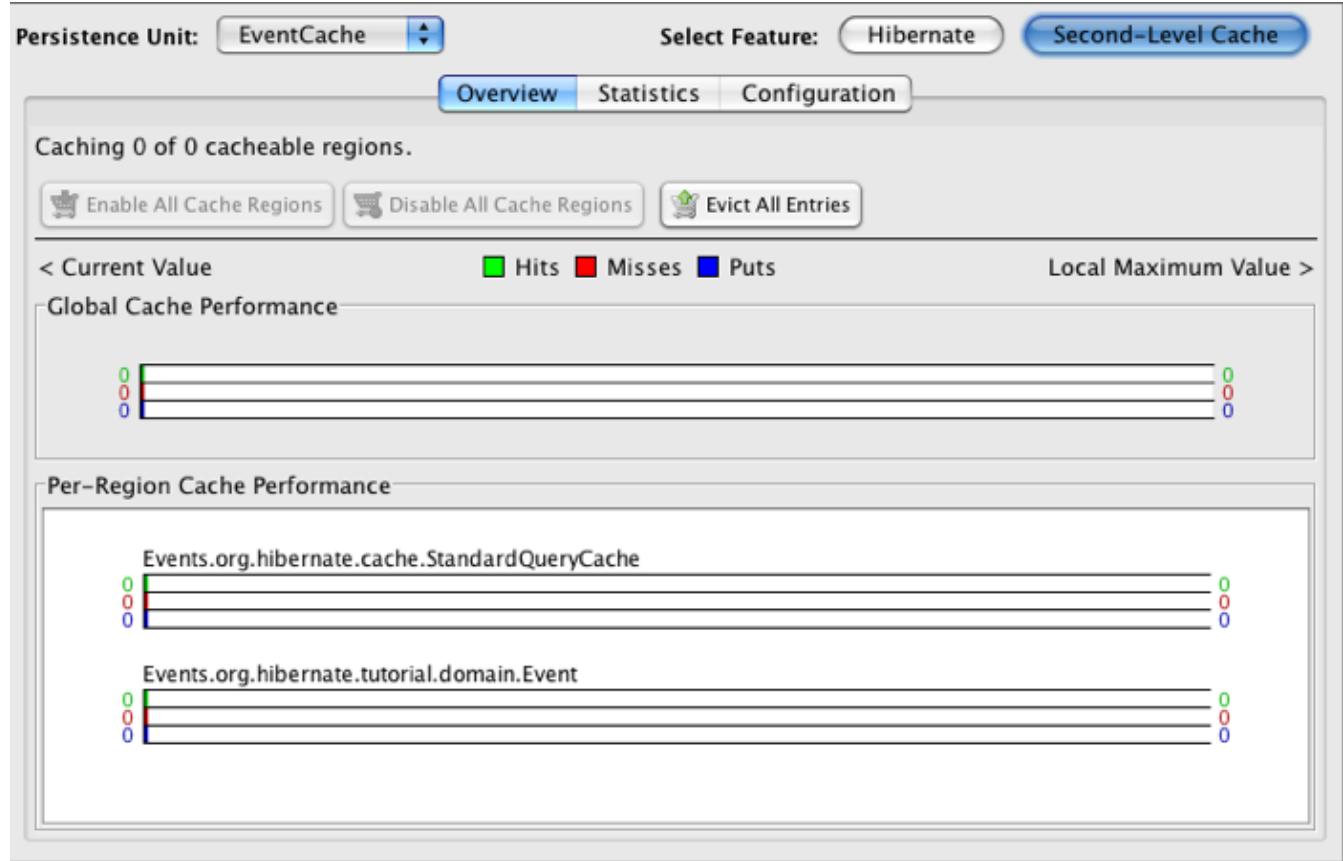
## Second-Level Cache View

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

## Second-Level Cache View



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).

## Second-Level Cache View

### 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

## Second-Level Cache View

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 ration 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 screenshot shows the Terracotta Developer Console interface. At the top, there are dropdown menus for 'Persistence Unit' (set to 'EventCache') and 'Select Feature' (set to 'Hibernate'). A blue button labeled 'Second-Level Cache' is highlighted. Below these are three tabs: 'Overview', 'Statistics', and 'Configuration', with 'Configuration' being the active tab. A message 'Caching 0 of 0 cacheable regions.' is displayed. On the right, there is a button 'Generate Cache Configuration...'. A table below lists configuration properties for cache regions:

| Region | Cached | TTI | TTL | Target Max In-Memo... | Target Max... |
|--------|--------|-----|-----|-----------------------|---------------|
|        |        |     |     |                       |               |

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).

## Second-Level Cache View

- 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](#).

# Clustered Quartz Scheduler Applications

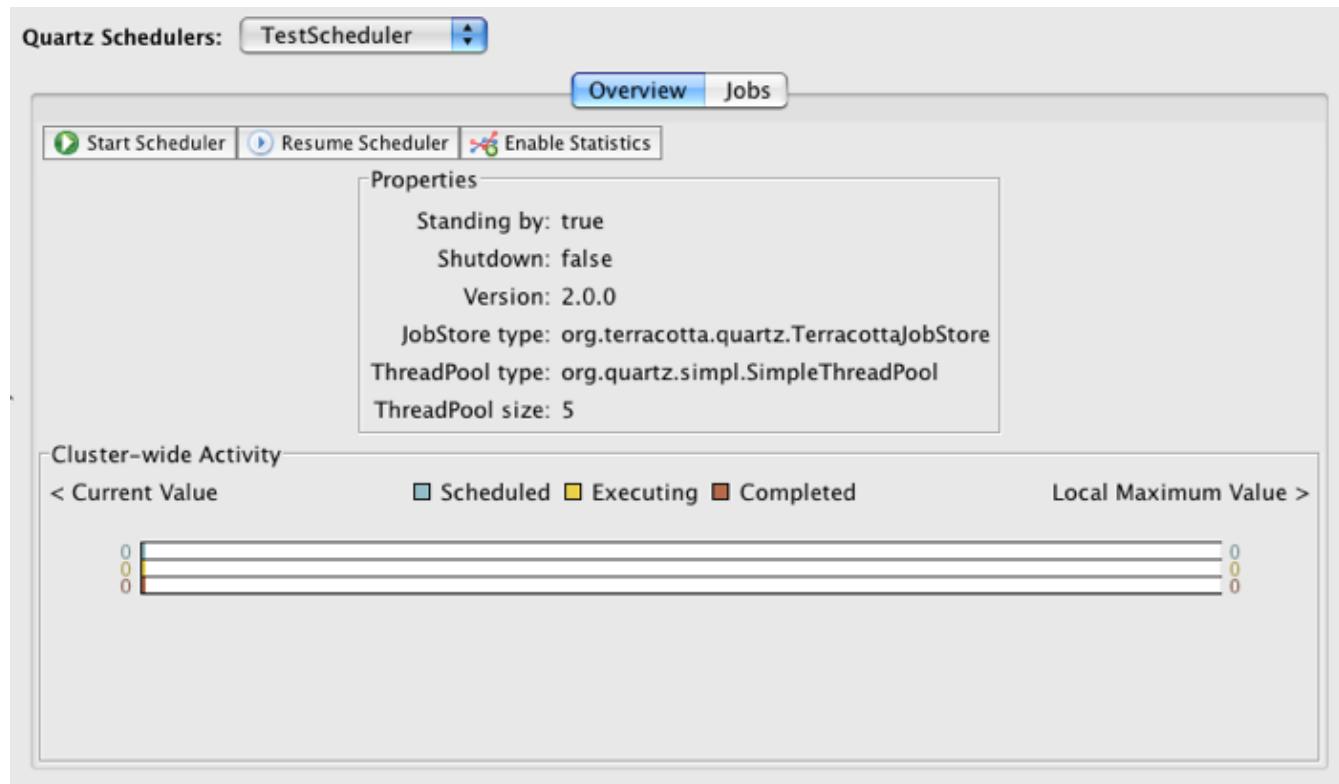
The Terracotta JobStore for Quartz Scheduler clusters the Quartz job-scheduling service. If you are clustering Quartz Scheduler, the Quartz view is available. The Quartz view offers the following features:

- 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).

## Overview

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 screenshot shows the 'Jobs' panel of the Quartz Scheduler interface. At the top, there is a dropdown menu labeled 'Quartz Schedulers' with 'TestScheduler' selected. Below the dropdown are two tabs: 'Overview' (which is active) and 'Jobs'. The main area is divided into two subpanels: 'Job Details' on the left and 'Triggers' on the right. The 'Job Details' subpanel contains a tree view of job groups and their triggers. The 'Triggers' subpanel lists individual triggers. At the bottom, there is a table titled 'Job Execution History' with columns for Job Name, Trigger Name, Refires, Fire Time, Scheduled Fire Time, Previous Fire, and Next Fire. The table has one row of data.

| Job Name   | Trigger Name         | Refires | Fire Time              | Scheduled Fire Time   | Previous Fire         | Next Fire             |
|--|----------------------|---------|------------------------|-----------------------|-----------------------|-----------------------|
| instance_one,node=6942231b67474eda971cb42923178582 | instance_one,node... | 0       | Mar 1, 2011 2:37:13 PM | Mar 1, 2011 2:37:1... | Mar 1, 2011 2:37:1... | Mar 1, 2011 2:37:1... |

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

## Overview

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.

## 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.

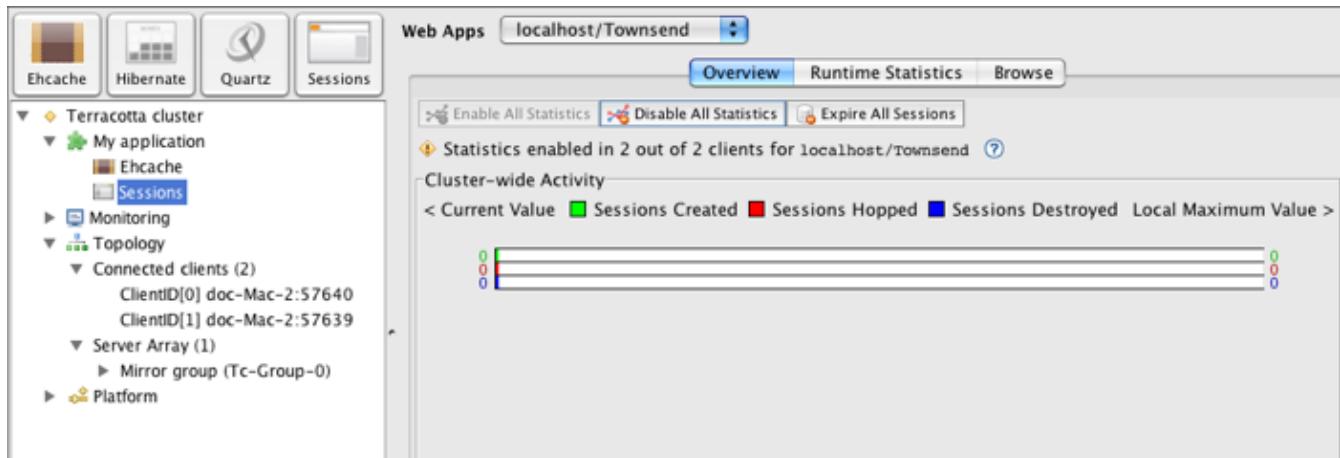
NOTE: Statistics and Performance

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

### 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.

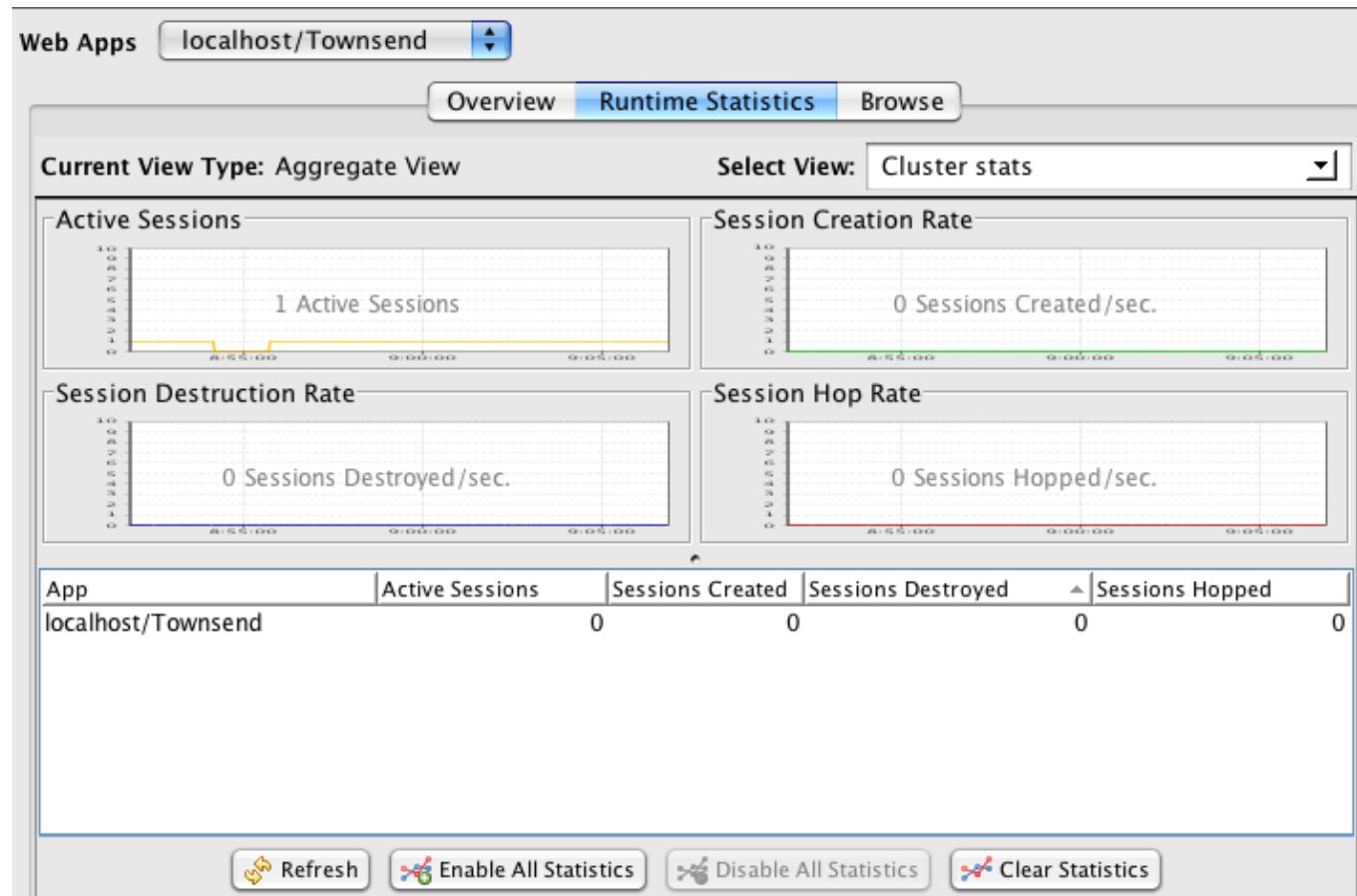
These statistics are displayed as color-coded bar graphs with current values shown on the left end and "high-water" (current maximum) values shown on the right end.

The **Overview** panel provides the following controls:

- Expire All Sessions – Close all sessions.
- Enable Statistics – Turns on the gathering of statistics. *Gathering statistics may have a negative impact on performance.*
- Disable Statistics – Turns off the gathering of statistics.

### Runtime (Sessions) Statistics

## Overview



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.
- Enable Statistics – Turns on the gathering of statistics. *Gathering statistics may have a negative impact on performance.*
- Disable Statistics – Turns off the gathering 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.

## Overview

### 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 Hop 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

## Working with Terracotta Server Arrays

The screenshot shows the Terracotta Developer Console interface. At the top, there's a header bar with "Web Apps" and "localhost/Townsend". Below it is a navigation bar with tabs: "Overview", "Runtime Statistics", and "Browse" (which is highlighted). Underneath the navigation bar, there are buttons for "Get Sessions" and "Find Session". A dropdown menu indicates a "Limited to a batch size of: 10". The main content area displays a message "Retrieved all 1 sessions" followed by a list of session IDs: "XEjPKnKvij0mGbbNG6et". Below this list, a table provides detailed information for the selected session:

| Attribute           | Value  |
|---------------------|--|
| displayUserListForm | DynaActionForm[dynaClass=displayUserListForm,listLen...] |
| datakeeper          | Casio EX-Z850[id=0008, quantity=14, details='8.0 M...    |

The **Browse** panel lists active sessions and has the following controls:

- Get Sessions – Obtain or refresh the list of session IDs of all active sessions. The session IDs are sorted from most recently created sessions at the top of the list to oldest sessions at the bottom.
- Batch-size menu – Choose a maximum number of sessions to return with **Get Sessions**.
- Find Session – Find sessions in the sessions table. To locate specific sessions, enter a string to match, then click **Find Session**.

To view the attributes and attribute values for a session, select the session ID in the sessions table. The session's existing attributes and values are displayed in the panel below

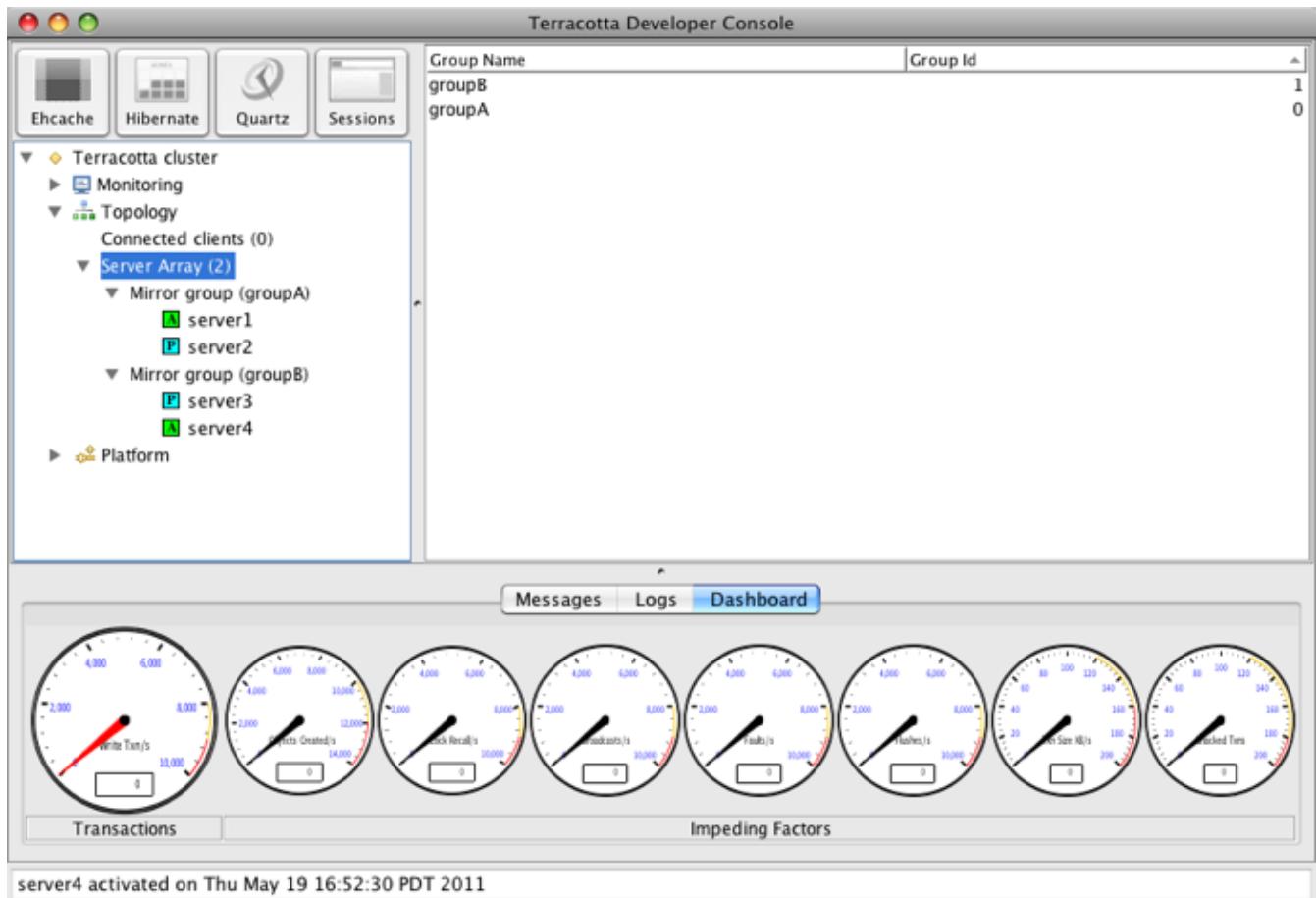
To manually expire a session, select the session's ID and open its context menu (for example, right-click on the session ID), then choose **Expire** from the context menu.

## Working with Terracotta Server Arrays

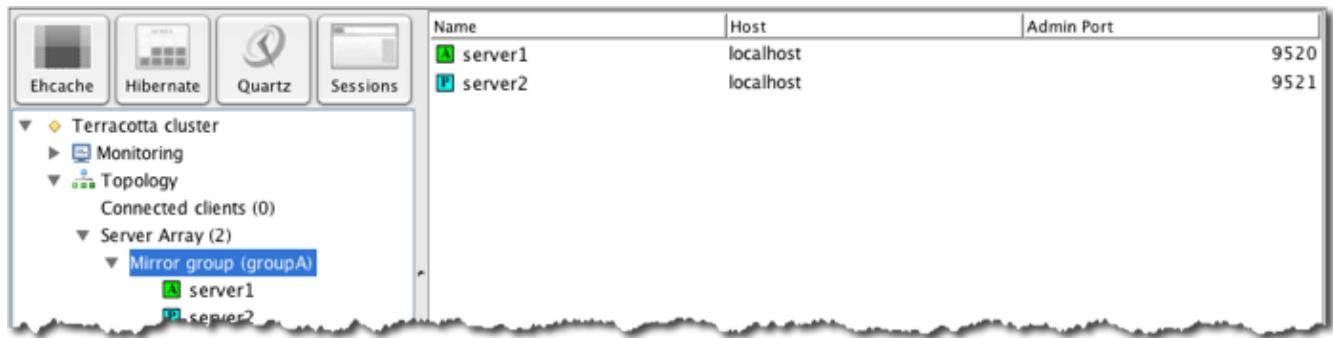
Terracotta servers are arranged in *mirror groups*, each of which contains at least one active server instance. A High Availability mirror group also contains one backup server instance, sometimes called a passive server or "hot standby." Under the **Topology** node, a **Server Array** node lists all 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**.

## Server Panel



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.

## Server Panel

Selecting a specific server's node displays that server's panel, with the following 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.

## Server Panel

Main   Environment   TCPProperties   Process Arguments   Config   Logging Settings

Activated on Thu May 19 16:50:42 PDT 2011

| Field            | Value   |
|------------------|---|
| Name             | server1   |
| Host             | 10.2.0.133  |
| Address          | 10.2.0.133  |
| JMX port         | 9520  |
| DSO port         | 9510  |
| Version          | Terracotta Enterprise 3.5.1                       |
| Build            | 20110415-160415 (Revision 10912-17477 b...)       |
| License          | DCV2, authentication, ehcache, ehcache monitor... |
| Persistence mode | permanent-store                                   |
| Failover mode    | networked-active-passive                          |

The following tabs display environment and configuration information, and provide a **Find** tool for performing string searches on that information:

- **Environment** – The server's JVM system properties.
- **TCPProperties** – The Terracotta properties that the server is using.
- **Process Arguments** – The JVM arguments that the server was started with.

Main   Environment   TCPProperties   **Process Arguments**   Config   Logging Settings

```
-n server1
-Xserver
-XX:MaxDirectMemorySize=64g
-Xms512m
-Xmx512m
-XX:+HeapDumpOnOutOfMemoryError
-Dcom.sun.management.jmxremote
-Dtc.install-root=bin/..
-Dsun.rmi.dgc.server.gcInterval=31536000
```

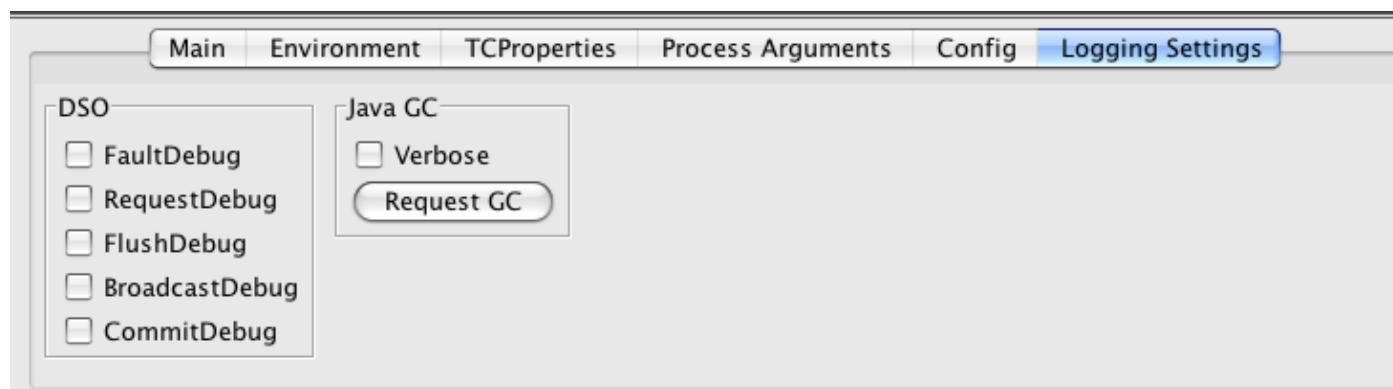
Find:  Next Previous

- **Config** – The Terracotta configuration that the server is using.

The **Logging Settings** tab provides controls for setting 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.

## Connecting and Disconnecting from a Server



The **Logging Settings** tab also provides Java garbage-collection controls.

## 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](#).

## 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   |
|----------------|---------------------|---|
| <i>GREEN*</i>  | Active              | The server is connected and ready for work.   |
| <i>RED*</i>    | Unreachable         | The server, or the network connection to the server, is down.   |
| <i>YELLOW*</i> | 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. |

## Server Connection Status

ORANGE\* Initializing

Normally the file lock is released only when the active server fails. The passive will then move to ACTIVE state (green status light).

CYAN\* Standby

In a network-based multi-server cluster, a passive server must initialize its state before going into standby mode.

In a network-based multi-server cluster, a passive server is ready to become active if the active server fails.

## 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.

The screenshot shows the Terracotta Developer Console interface. On the left is a navigation tree with icons for Ehcache, Hibernate, Quartz, and Sessions. Below these are nodes for Terracotta cluster, My application (with Ehcache), Monitoring, Topology (with Connected clients (2) selected), Server Array (1), and Platform. The main panel shows a table titled 'Connected clients' with the following data:

| Host       | Port  | Client ID | Live Object Count |
|------------|-------|-----------|-------------------|
| 10.2.0.133 | 50749 | 0         | 2,608             |
| 10.2.0.133 | 50771 | 2         | 2,516             |

To view the client nodes that appear in the **Connected clients** panel, expand the **Connected clients** node.

## Client Panel

Selecting a specific client's node displays that client's panel, with the following tabs.

The **Main** tab displays a list of client properties such as hostname and DSO port.

The following tabs display environment and configuration information, and provide a **Find** tool for performing string searches on that information:

- **Environment** – The client's JVM system properties.
- **TCPProperties** – The Terracotta properties that the client is using.
- **Process Arguments** – The JVM arguments that the client was started with.
- **Config** – The Terracotta configuration the client is using.

The **Logging Settings** tab provides options to set logging items corresponding to DSO client debugging as well as Java garbage-collection controls. See the [Configuration Guide and Reference](#) for details on the various debug logging options.

Connecting and Disconnecting Clients

## 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](#).

## 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

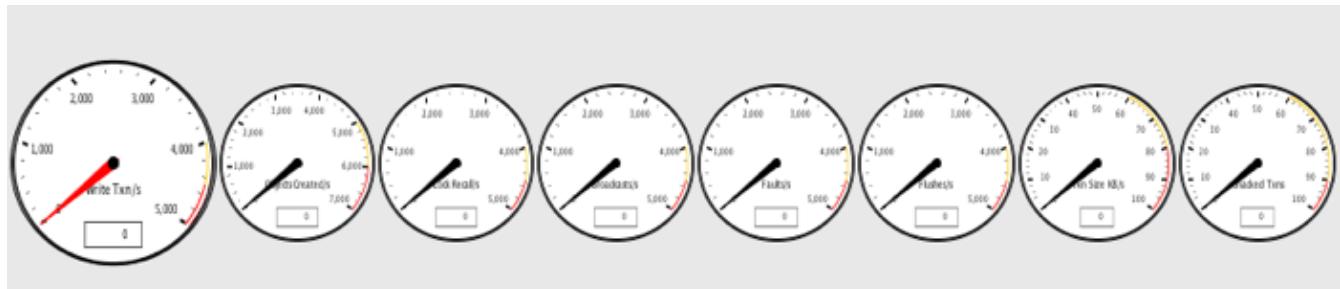
The Cache Miss Rate measures the number of client requests for an object that cannot be met by a server's cache and must be faulted in from disk. An upward trend in this graph can expose a bottleneck in your cluster. For more information, see [onheap fault/flush Rate \(Cluster, Server\)](#).

## Real-Time Performance Monitoring

Real-time cluster monitoring allows you to spot issues as they develop in the cluster.

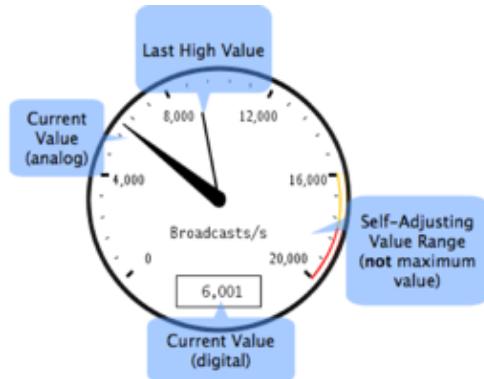
### Dashboard

The cluster activity gauges provide real-time readings of critical cluster metrics.



Each gauge has the following characteristics:

## Real-Time Performance Monitoring



- 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:

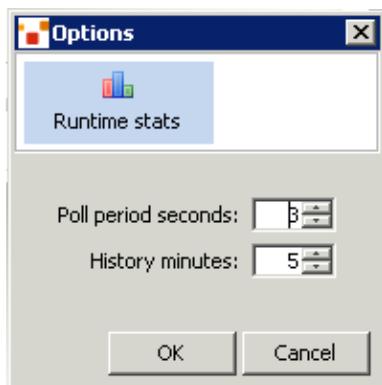
- **Objects Created/s** -- The rate of shared objects being created. A rising trend can have a negative impact on performance by reducing available memory and necessitating more garbage collection.
- **Lock Recalls/s** -- The number of locks being recalled by Terracotta servers. Growing lock recalls 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.

## Real-Time Performance Monitoring

- **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 or 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.

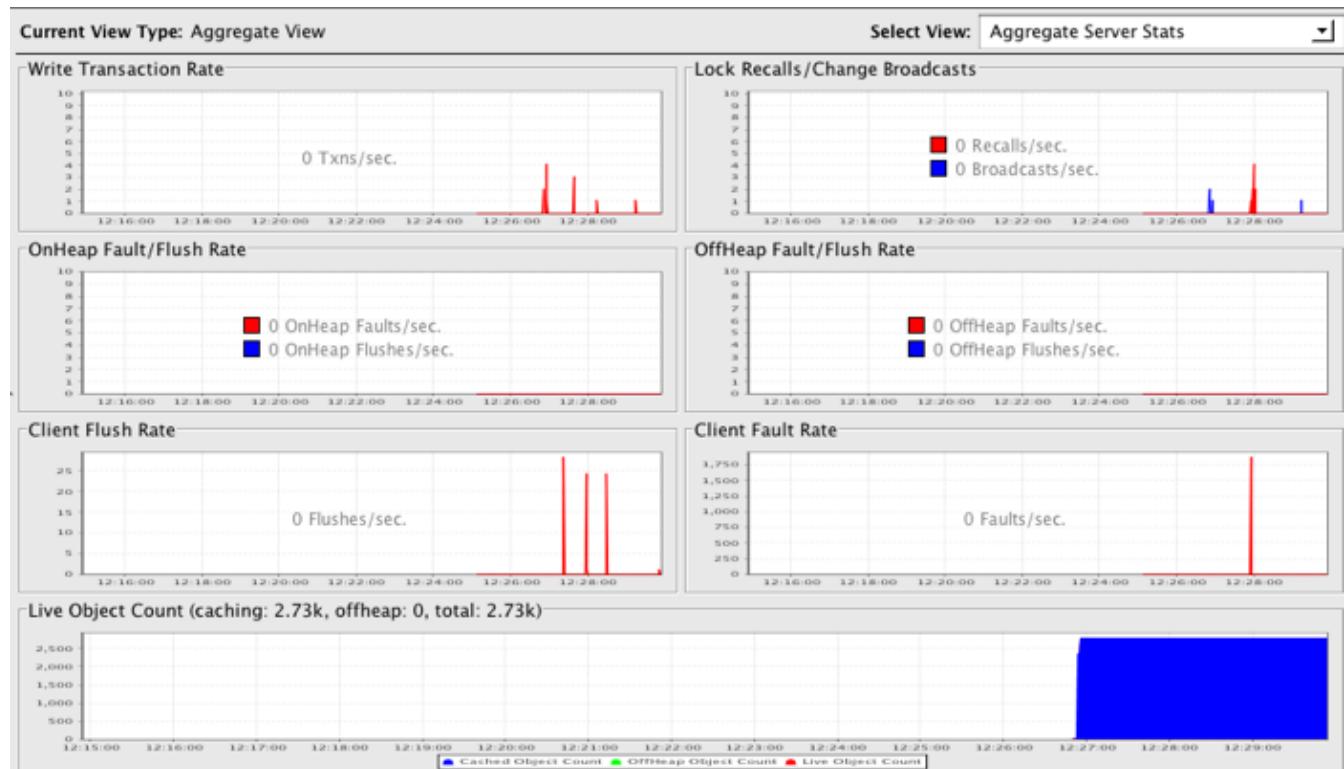
To record and save historical data, see [Cluster Statistics Recorder](#).

To view runtime statistics for a cluster, expand the cluster's **Monitoring** node, then click the **Runtime statistics** node.

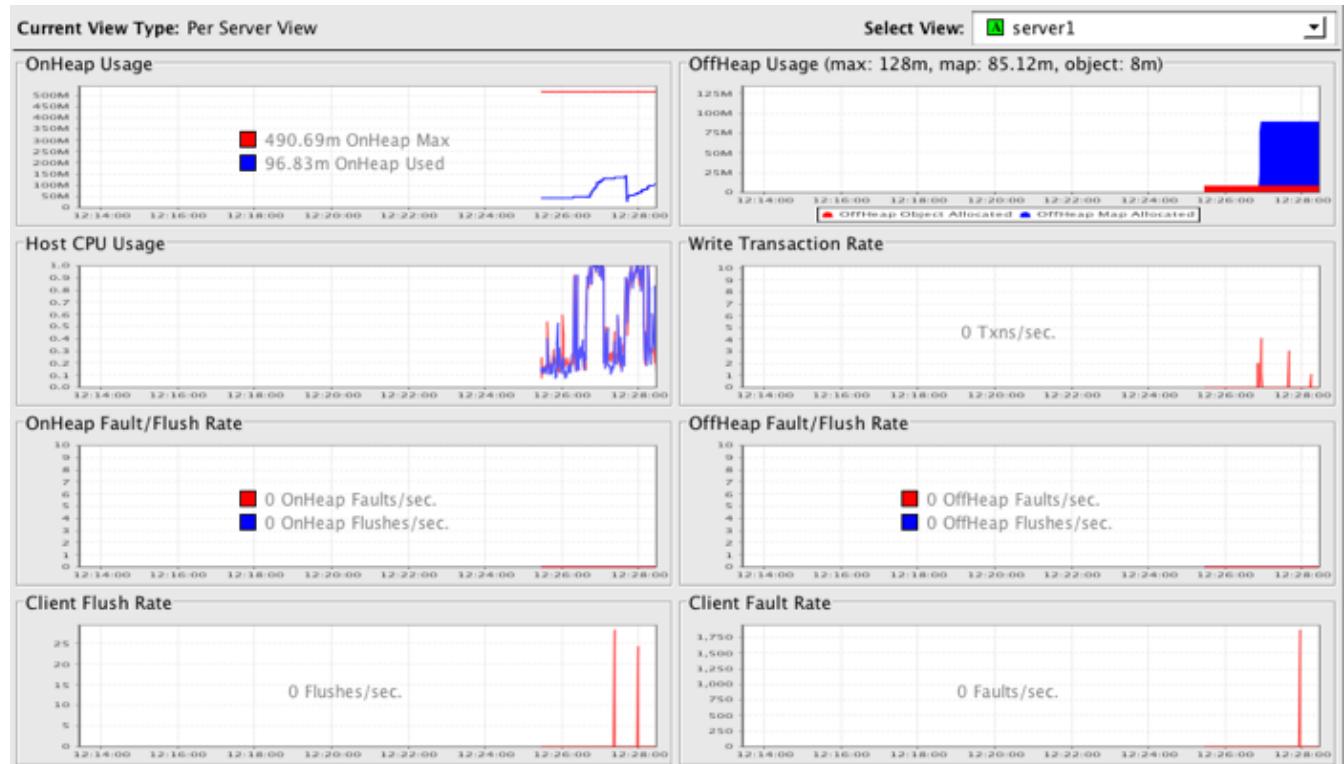
Use the **Select View** menu to set the runtime statistics view to one of the following:

- Aggregate View – Choose **Aggregate Server Stats** to display cluster-wide statistics.

## Real-Time Performance Monitoring



- 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.

## Real-Time Performance Monitoring

### 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.

## Real-Time Performance Monitoring

### **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 indicated poor locality of reference.

Change broadcasts tracks the number of object-change notifications that Terracotta servers are sending. See [12 changes per broadcast](#), [12 broadcast count](#), and [12 broadcast per transaction](#) for more information on

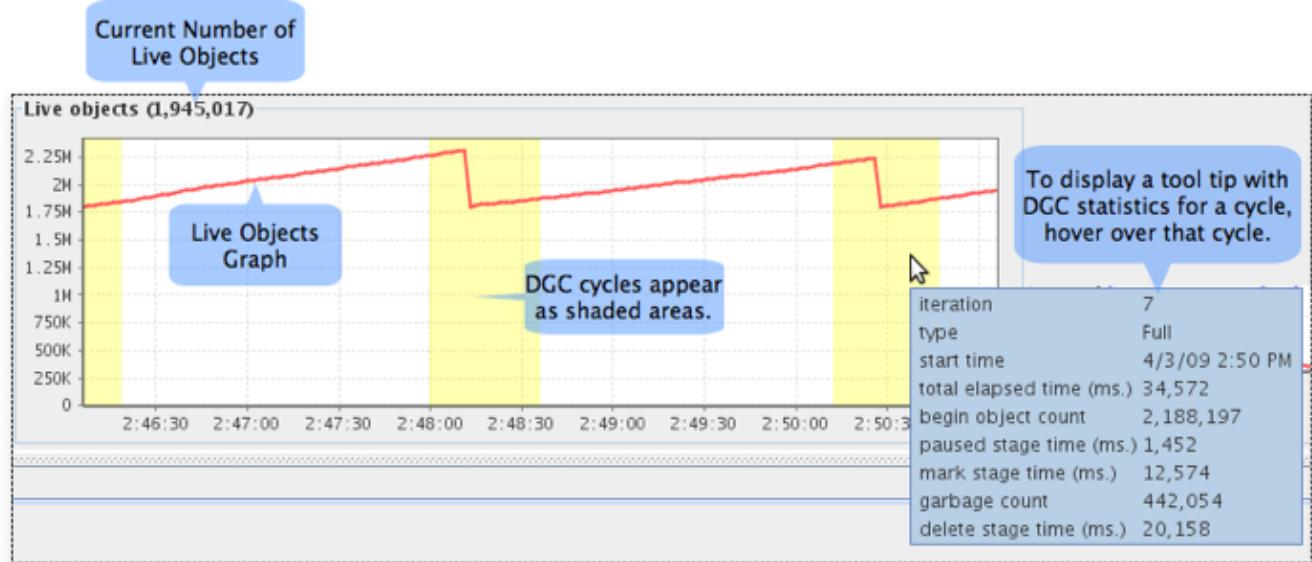
## Real-Time Performance Monitoring

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                 | Full – Running a full collection cycle targeting all eligible objects. Young – Running a collection cycle targeting Young Generation objects. |
| Status    |                                   |   |

## Real-Time Performance Monitoring

The collection cycle's START – Monitoring for object reference changes and collecting statistics such as the object begin count. MARK – Determining which objects should be collected and which should not. PAUSE – Determining if any marked objects should not be collected. MARK COMPLETED – Stops checking for reference changes (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**.

NOTE: Periodic and Inline DGC

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

## 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.

## 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.

## 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.

## Shared Objects

Applications clustered with Terracotta use shared objects to keep data coherent. Monitoring shared objects

## 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:

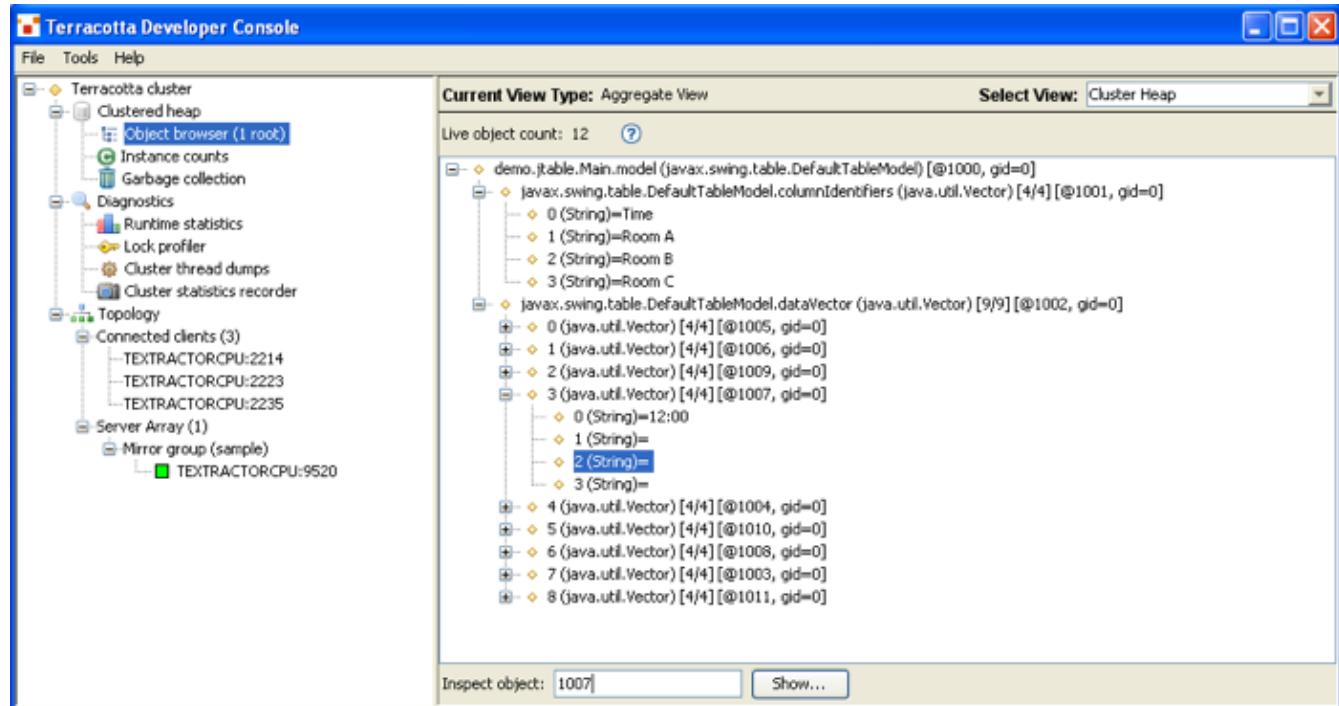
## Shared Objects

1. Find the object ID of the object at the root of the portion you want to inspect. An object ID has the format @. 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 clients are sharing them.



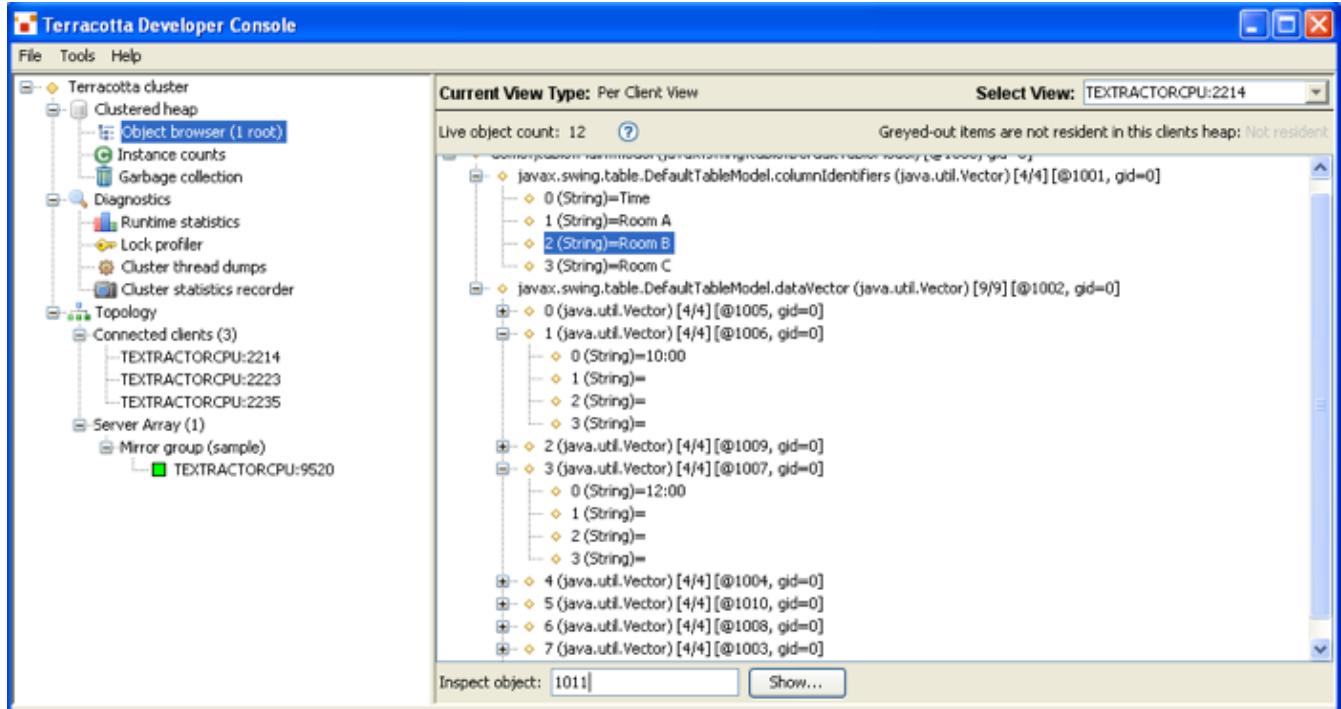
To see object graphs specific to a client, see [Client Object Browsing](#).

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.

## Shared Objects



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.

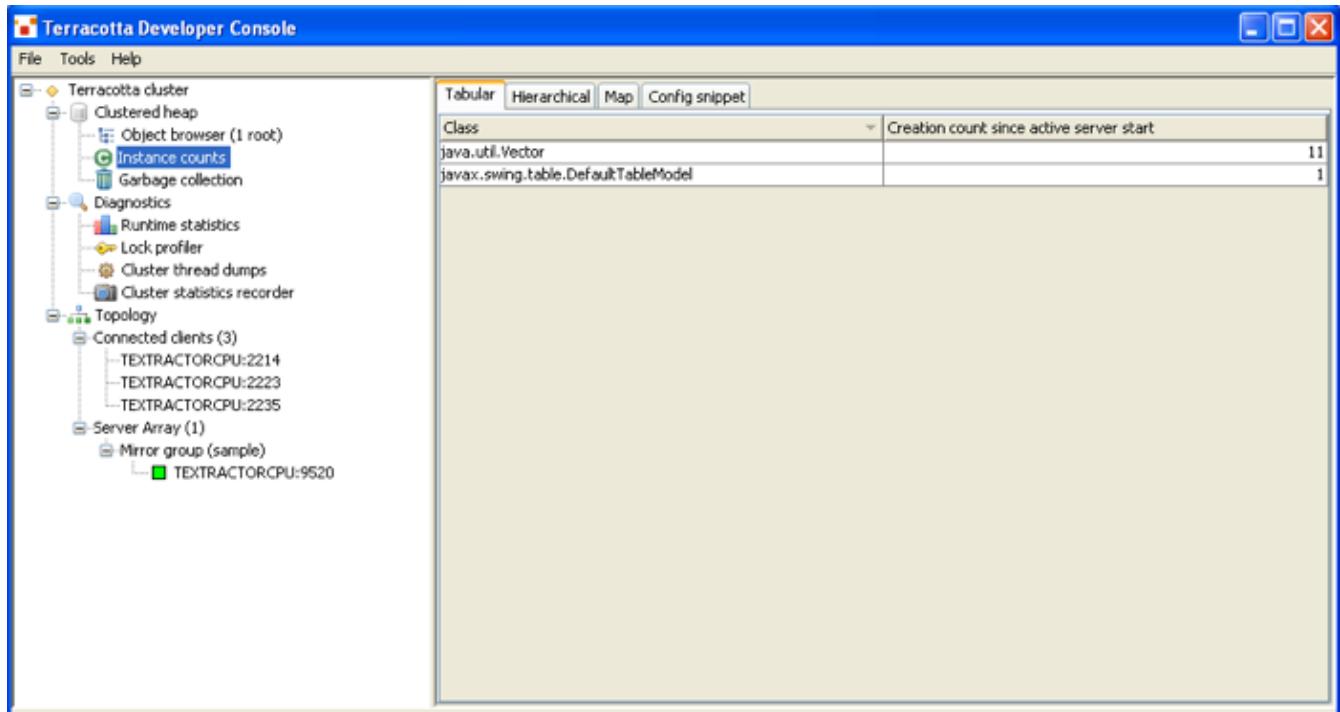
## Classes Browser

Terracotta allows for transparent, clustered object state synchronization. To accomplish this, 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.

## Shared Objects



## 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.

## 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-contended access to shared state. Statistics are displayed in a table (see [The Statistics Table](#)).

## Lock Profiler

The screenshot shows the Lock Profiler interface. At the top, there are buttons for 'Enable lock profiling' (Off, On), 'Trace depth' (set to 4), and 'Refresh'. Below this is a table titled 'Clients' showing lock statistics:

| Lock                     | Times Requested | Times Hopped | Average Contenders | Average Acquire Time | Average Held Time | Average Nested |
|--------------------------|-----------------|--------------|--------------------|----------------------|-------------------|----------------|
| ▼ @DistributedMethodCall | 4               | 0            | 0                  | 0                    | 1                 | 0              |
| ▼ demo.chatter.CI        | 4               | 0            | 0                  | 0                    | 1                 | 0              |
| ▼ demo.chatte            | 4               | 0            | 0                  | 0                    | 1                 | 0              |
| demo.ch                  | 4               | 0            | 0                  | 0                    | 1                 | 0              |

Below the table is a search bar labeled 'Find: demo' with 'Next' and 'Previous' buttons. At the bottom, there are two panes: 'Trace' on the left showing code snippets like 'demo.chatter.ChatManager.sendNewM' and 'demo.chatter.ChatManager.send(Unkn', and 'Distributed Invoke' on the right.

## 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

## Lock Profiler

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. |

## Lock Profiler

|                      |   |
|----------------------|---|
| 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.

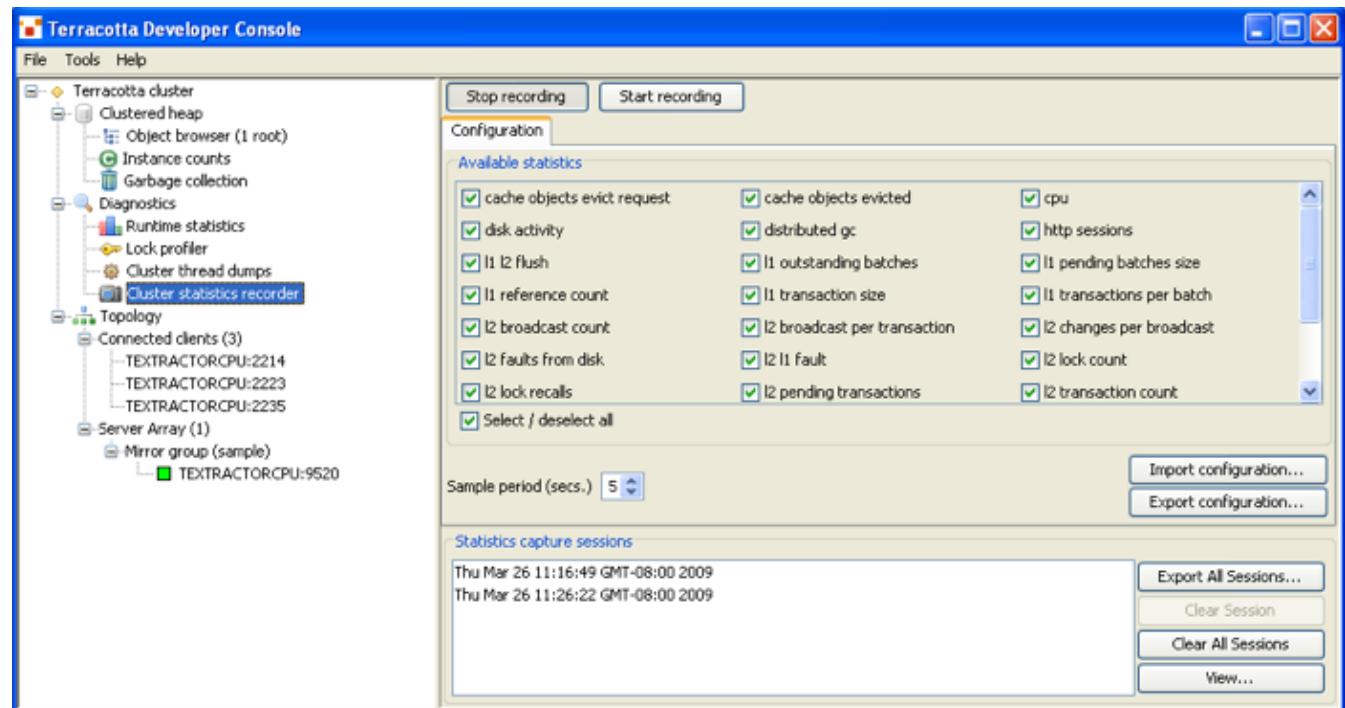
## Lock Element Details

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.

# Recording and Viewing Statistics

## 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.



## Cluster Statistics Recorder

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.

## Troubleshooting the Console

This section provides solutions for common issues that can affect both the Terracotta Developer Console and Operations Center.

### 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.

### 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](#).

### 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.

## Console Runs Very Slowly

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 1.6, be sure to set property `sun.java2d.pmoffscreen` to "false" to allow Swing buffers to reside in memory:

```
-Dsun.java2d.pmoffscreen=false
```

For information about this Java system property, see  
<http://download.oracle.com/javase/1.5.0/docs/guide/2d/flags.html#pmoffscreen>.

You can add these properties to the Developer Console start-up script (`dev-console.sh` or `dev-console.bat`).

## Console Logs and Configuration File

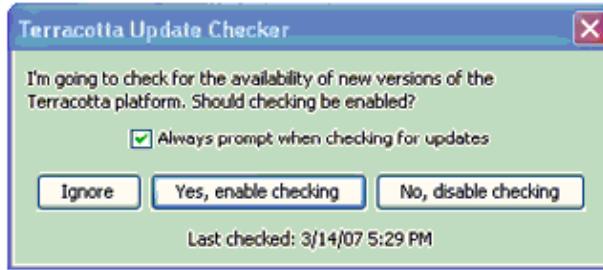
The Terracotta Developer Console stores a log file (`devconsole.log.<number>`) and a configuration file (`.AdminClient.xml`) in the home directory of the user who ran the console. The log file contains a record of errors encountered by the console.

## 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 \(backup-data\)](#)).

## 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](http://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.

## 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

### 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. High counts imply that free memory could be low.

### 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.

### 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. The objects are not literally copied to the L2 — this data already exists on the L2 — they are removed from the L1.

### 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.

### 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.

### memory (usage)

The amount of memory (heap) usage over time.

### vm garbage collector

The standard Java garbage collector's behavior, tracked on all JVMs in the cluster.

### 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.

**I2 pending transactions**

## I2 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.

## 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.

## server transaction sequencer stats

Statistics on the Terracotta server-instance transaction sequencer, which sequences transactions as resources become available while maintaining transaction order.

## network activity

The amount of data transmitted and received by a Terracotta server instance in bytes per second.

## I2 changes per broadcast

The number of updates to objects on disk per broadcast message (see [I2 broadcast count](#)).

## message monitor

The network message count flowing over TCP from Terracotta clients to the Terracotta server.

## I2 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.

## I2 transaction count

The number of [Terracotta transactions](#) being processed (per second) by a Terracotta server instance.

## I2 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.

system properties

## **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.

## **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.

## **cpu (usage)**

The percent of CPU resources being used. Dual-core processors are broken out into CPU0 and CPU1.

# The Terracotta Operations Console

The Operations Console is documented [here](#).

# JMX Management and Monitoring

## Introduction

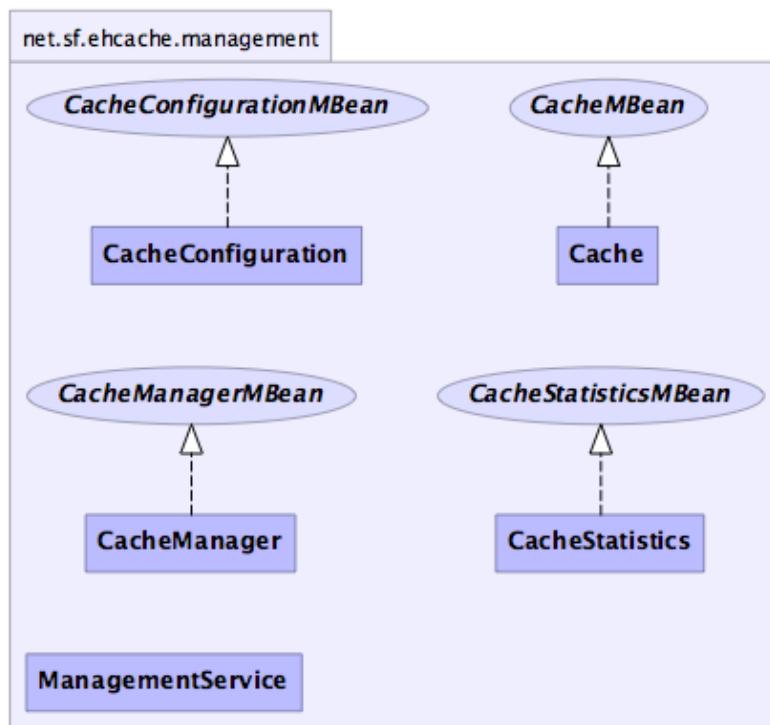
JMX creates a standard way of instrumenting classes and making them available to a management and monitoring infrastructure. This provides an alternative to the dev console for custom or third-party tools.

## JMX Overview

The `net.sf.ehcache.management` package contains MBeans and a ManagementService for JMX management of BigMemory Max. It is in a separate package so that JMX libraries are only required if you want to use it - there is no leakage of JMX dependencies into the core Ehcache package.

Use `net.sf.ehcache.management.ManagementService.registerMBeans(...)` static method to register a selection of MBeans to the MBeanServer provided to the method. If you wish to monitor Ehcache but not use JMX, use the existing public methods on Cache and CacheStatistics.

The Management package is illustrated in the following image.



generated by yDoc

## MBeans

BigMemory Max supports Standard MBeans. MBeans are available for the following:

## MBeans

- CacheManager
- Cache
- CacheConfiguration
- CacheStatistics

All MBean attributes are available to a local MBeanServer. The CacheManager MBean allows traversal to its collection of Cache MBeans. Each Cache MBean likewise allows traversal to its CacheConfiguration MBean and its CacheStatistics MBean.

## JMX Remoting

The JMX Remote API allows connection from a remote JMX Agent to an MBeanServer via an MBeanServerConnection. Only Serializable attributes are available remotely. The following Ehcache MBean attributes are available remotely:

- limited CacheManager attributes
- limited Cache attributes
- all CacheConfiguration attributes
- all CacheStatistics attributes

Most attributes use built-in types. To access all attributes, add ehcache.jar to the remote JMX client's classpath. For example, jconsole -J-Djava.class.path=ehcache.jar.

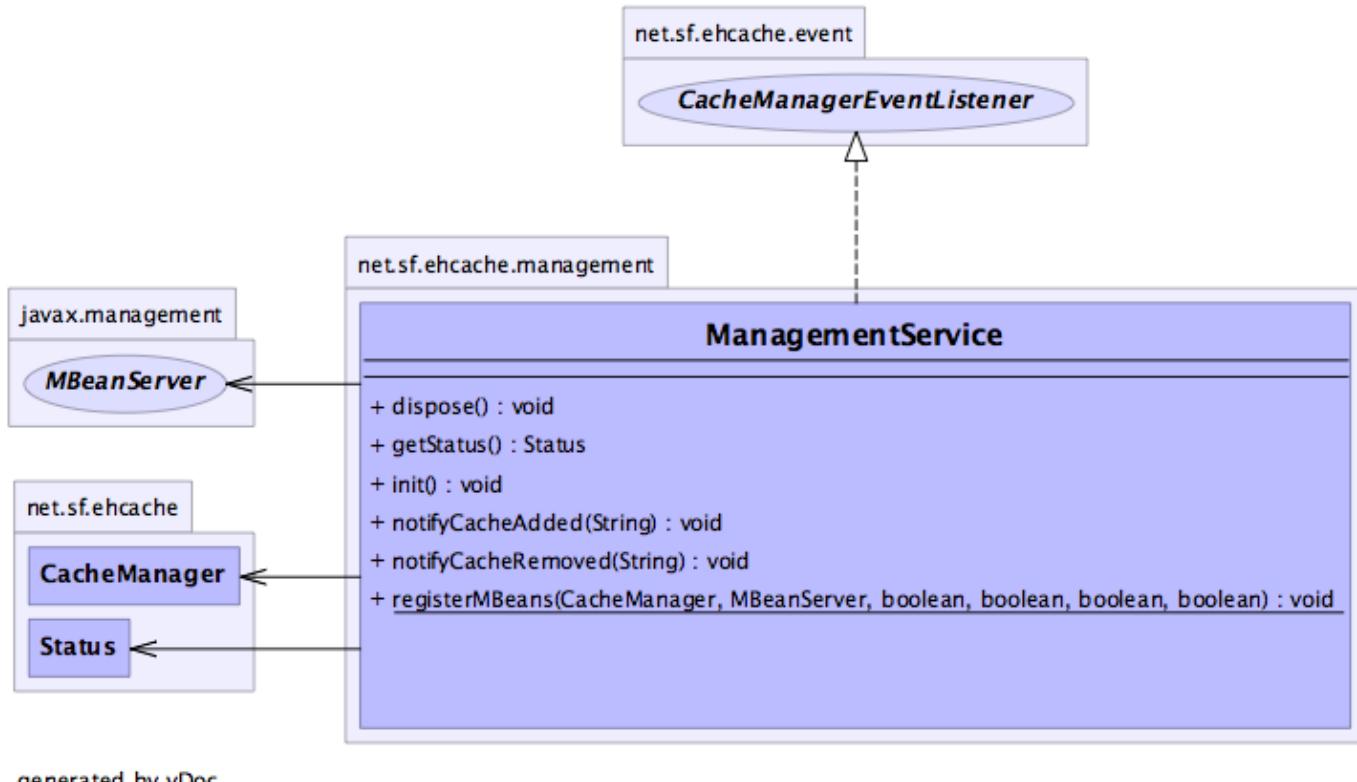
## ObjectName naming scheme

- CacheManager - "net.sf.ehcache:type=CacheManager,name=<CacheManager>"
- Cache - "net.sf.ehcache:type=Cache,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheConfiguration
- "net.sf.ehcache:type=CacheConfiguration,CacheManager=<cacheManagerName>,name=<cacheName>"
- CacheStatistics -  
"net.sf.ehcache:type=CacheStatistics,CacheManager=<cacheManagerName>,name=<cacheName>"

## The Management Service

The ManagementService class is the API entry point.

## The Management Service



There is only one method, `ManagementService.registerMBeans` which is used to initiate JMX registration of a CacheManager's instrumented MBeans. The `ManagementService` is a `CacheManagerEventListener` and is therefore notified of any new Caches added or disposed and updates the `MBeanServer` appropriately. Initiated MBeans remain registered in the `MBeanServer` until the CacheManager shuts down, at which time the MBeans are deregistered. This ensures correct behavior in application servers where applications are deployed and undeployed.

```
/**  
 * This method causes the selected monitoring options to be registered  
 * with the provided MBeanServer for caches in the given CacheManager.  
 *  
 * While registering the CacheManager enables traversal to all of the other  
 * items, this requires programmatic traversal. The other options allow entry points closer  
 * to an item of interest and are more accessible from JMX management tools like JConsole.  
 * Moreover CacheManager and Cache are not serializable, so remote monitoring is not  
 * possible for CacheManager or Cache, while CacheStatistics and CacheConfiguration are.  
 * Finally CacheManager and Cache enable management operations to be performed.  
 *  
 * Once monitoring is enabled caches will automatically added and removed from the  
 * MBeanServer as they are added and disposed of from the CacheManager. When the  
 * CacheManager itself shutdowns all registered MBeans will be unregistered.  
 *  
 * @param cacheManager the CacheManager to listen to  
 * @param mBeanServer the MBeanServer to register MBeans to  
 * @param registerCacheManager Whether to register the CacheManager MBean  
 * @param registerCaches Whether to register the Cache MBeans  
 * @param registerCacheConfigurations Whether to register the CacheConfiguration MBeans  
 * @param registerCacheStatistics Whether to register the CacheStatistics MBeans  
 */  
public static void registerMBeans(
```

## JConsole Example

```
net.sf.ehcache.CacheManager cacheManager,
MBeanServer mBeanServer,
boolean registerCacheManager,
boolean registerCaches,
boolean registerCacheConfigurations,
boolean registerCacheStatistics) throws CacheException {
```

## JConsole Example

This example shows how to register CacheStatistics in the JDK platform MBeanServer, which works with the JConsole management agent.

```
CacheManager manager = new CacheManager();
MBeanServer mBeanServer = ManagementFactory.getPlatformMBeanServer();
ManagementService.registerMBeans(manager, mBeanServer, false, false, false, true);
```

CacheStatistics MBeans are then registered.

The screenshot shows the JConsole interface. In the top navigation bar, the title is "J2SE 5.0 Monitoring & Management Console: 3075@localhost". Below it, the "Connection" tab is selected. In the main area, there are several tabs: Summary, Memory, Threads, Classes, MBeans, and VM. The "MBeans" tab is active. On the left, a tree view shows the following structure under "Tree": JMImplementation, java.lang, java.util.logging, net.sf.ehcache, CacheStatistics, net.sf.ehcache.CacheManager@881cb3, CachedLogin, FooterPageCache, SimplePageCachingFilter, SimplePageCachingFilterWithBlankPageProblem, SimplePageFragmentCachingFilter, net.sf.ehcache.constructs.asynchronous.MessageCache, persistentLongExpiryIntervalCache, sampleCache1, sampleCache2, sampleCacheNoldle, sampleCacheNotEternalButNoldleOrExpiry, sampleIdlingExpiringCache. The "sampleCache1" node is currently selected. To the right of the tree view, there is a table titled "Attributes" with columns "Name" and "Value". The table contains the following data:

| Name                          | Value        |
|-------------------------------|--------------|
| AssociatedCacheName           | sampleCache1 |
| CacheHits                     | 1            |
| CacheMisses                   | 0            |
| InMemoryHits                  | 1            |
| ObjectCount                   | 1            |
| OnDiskHits                    | 0            |
| StatisticsAccuracy            | 1            |
| StatisticsAccuracyDescription | Best Effort  |

A "Refresh" button is located at the bottom right of the table panel.

*CacheStatistics MBeans in JConsole*

## Hibernate statistics

If you are running Terracotta clustered caches as hibernate second-level cache provider, it is possible to access the hibernate statistics and Ehcache stats via JMX. EhcacheHibernateMBean is the main interface that exposes all the APIs via JMX. It basically extends two interfaces -- EhcacheStats and HibernateStats. Please look into the specific interface for more details. You may also refer to this [online tutorial](#).

# Performance

Collection of cache statistics is not entirely free of overhead, however, the statistics API switches on/off automatically according to usage. If you need few statistics, you incur little overhead; on the other hand, as you use more statistics, you can incur more. Statistics are off by default.

## SSL-Secured JMX Monitoring

This page documents setting up an SSL-enabled connection for remote monitoring of a Terracotta Server from a simple Java client using Java Management Extensions (JMX) technology.

### Compile the Client

Use the sample client code below, but adapt the host, port, username, and password variables according to your setup.

```

import java.util.HashMap;
import java.util.Map;

import javax.management.remote.JMXConnector;
import javax.management.remote.JMXConnectorFactory;
import javax.management.remote.JMXServiceURL;
import javax.management.remote.rmi.RMIC_connectorServer;
import javax.rmi.ssl.SslRMIClientSocketFactory;
import javax.rmi.ssl.SslRMIServerSocketFactory;

public class Main {
    public static void main(String[] args) throws Exception {
        String host = "terracotta-server-host";
        String port = "9520";
        String username = "terracotta";
        String password = "terracotta-user-password";

        Object[] credentials = { username, password.toCharArray() };

        SslRMIClientSocketFactory csf = new SslRMIClientSocketFactory();
        SslRMIServerSocketFactory ssf = new SslRMIServerSocketFactory();

        Map<String, Object> env = new HashMap<String, Object>();
        env.put(RMIC_connectorServer.RMI_CLIENT_SOCKET_FACTORY_ATTRIBUTE, csf);
        env.put(RMIC_connectorServer.RMI_SERVER_SOCKET_FACTORY_ATTRIBUTE, ssf);
        env.put("com.sun.jndi.rmi.factory.socket", csf);
        env.put("jmx.remote.credentials", credentials);

        JMXServiceURL serviceURL = new JMXServiceURL("service:jmx:rmi://" + host + ":" + port +
            "/jndi/rmi://" + host + ":" + port + "/jmxrmi");
        JMXConnector jmxConnector = JMXConnectorFactory.connect(serviceURL, env);

        // do some work with the JMXConnector

        jmxConnector.close();
    }
}

```

## Run the Client

# Run the Client

After compiling your client, configure the JVM with a truststore containing your Terracotta Server's certificate. You can simply re-use the one created for the Terracotta Server.

```
% java -Djavax.net.ssl.trustStore=/your/path/to/truststore.jks \
-Djavax.net.ssl.trustStorePassword=your_truststore_password \
Main
```

## About the Credentials

In the above example, the client's credentials are encoded as an array of Objects. The Object array contains the username as a String in the array's first slot, and the password as a char[] in the array's second slot. The Object array is then passed to the connection as the "jmx.remote.credentials" entry. Passing the credentials in this format is necessary to avoid an authentication failure, except for the following exception. If you are using the JConsole tool, the credentials are sent as String[] {String, String} instead of String[] {String, char[] }.

## Troubleshooting

### Password stack trace

The stack trace below indicates that the password you specified in the `javax.net.ssl.trustStorePassword` system property is not the same as in the truststore you specified in the `javax.net.ssl.trustStore` system property.

```
Exception in thread "main" java.io.IOException: Failed to retrieve RMIServer stub: javax.naming.CommunicationException [Root exception is java.rmi.ConnectIOException]
        java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing implementation (Algorithm Name: [null])
        at javax.management.remote.rmi.RMIClientImpl.connect(RMIClientImpl.java:338)
        at javax.management.remote.JMXConnectorFactory.connect(JMXConnectorFactory.java:248)
        at Main.main(Main.java:31)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:597)
        at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
Caused by: javax.naming.CommunicationException [Root exception is java.rmi.ConnectIOException]
        java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing implementation (Algorithm Name: [null])
        at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:101)
        at com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:185)
        at javax.naming.InitialContext.lookup(InitialContext.java:392)
        at javax.management.remote.rmi.RMIClientImpl.findRMIServerJNDI(RMIClientImpl.java:1886)
        at javax.management.remote.rmi.RMIClientImpl.findRMIServer(RMIClientImpl.java:1856)
        at javax.management.remote.rmi.RMIClientImpl.connect(RMIClientImpl.java:255)
        ... 7 more
Caused by: java.rmi.ConnectIOException: Exception creating connection to: localhost; nested exception is:
        java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing implementation (Algorithm Name: [null])
        at sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:614)
        at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:198)
        at sun.rmi.transport.tcp.TCPChannel.newConnection(TCPChannel.java:184)
        at sun.rmi.server.UnicastRef.newCall(UnicastRef.java:322)
        at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
        at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:97)
        ... 12 more
Caused by: java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing implementation (Algorithm Name: [null])
        at javax.net.ssl.DefaultSSLocketFactory.throwException(SSLocketFactory.java:179)
        at javax.net.ssl.DefaultSSLocketFactory.createSocket(SSLocketFactory.java:192)
```

## Troubleshooting

```
at javax.rmi.ssl.SslRMIClientSocketFactory.createSocket (SslRMIClientSocketFactory.java:100)
at sun.rmi.transport.tcp.TCPEndpoint.newSocket (TCPEndpoint.java:595)
...
... 17 more
Caused by: java.security.NoSuchAlgorithmException: Error constructing implementation (algorithm: [PROVIDED])
at java.security.Provider$Service.newInstance (Provider.java:1245)
at sun.security.jca.GetInstance.getInstance (GetInstance.java:220)
at sun.security.jca.GetInstance.getInstance (GetInstance.java:147)
at javax.net.ssl.SSLContext.getInstance (SSLContext.java:125)
at javax.net.ssl.SSLContext.getDefault (SSLContext.java:68)
at javax.net.ssl.SSLSocketFactory.getDefault (SSLSocketFactory.java:102)
at javax.rmi.ssl.SslRMIClientSocketFactory.getDefaultClientSocketFactory (SslRMIClientSocketFactory.java:100)
at javax.rmi.ssl.SslRMIClientSocketFactory.createSocket (SslRMIClientSocketFactory.java:100)
...
... 18 more
Caused by: java.io.IOException: Keystore was tampered with, or password was incorrect
at sun.security.provider.JavaKeyStore.engineLoad (JavaKeyStore.java:771)
at sun.security.provider.JavaKeyStore$JKS.engineLoad (JavaKeyStore.java:38)
at java.security.KeyStore.load (KeyStore.java:1185)
at com.sun.net.ssl.internal.ssl.TrustManagerFactoryImpl.getCacertsKeyStore (TrustManagerFactoryImpl.java:100)
at com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl.getDefaultTrustManager (DefaultSSLContextImpl.java:40)
at com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl.<init> (DefaultSSLContextImpl.java:40)
at sun.reflect.NativeConstructorAccessorImpl.newInstance0 (Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance (NativeConstructorAccessorImpl.java:61)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance (DelegatingConstructorAccessorImpl.java:45)
at java.lang.reflect.Constructor.newInstance (Constructor.java:513)
at java.lang.Class.newInstance0 (Class.java:357)
at java.lang.Class.newInstance (Class.java:310)
at java.security.Provider$Service.newInstance (Provider.java:1221)
...
... 25 more
Caused by: java.security.UnrecoverableKeyException: Password verification failed
at sun.security.provider.JavaKeyStore.engineLoad (JavaKeyStore.java:769)
...
... 37 more
```

## Truststore stack trace

The stack trace below indicates that the truststore you specified in the `javax.net.ssl.trustStore` system property does not exist or cannot be read.

```
Exception in thread "main" java.io.IOException: Failed to retrieve RMIServer stub: javax.naming.CommunicationException [Root exception is java.rmi.ConnectIOException]
java.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty
at javax.management.remote.rmi.RMIClient.connect (RMIClient.java:338)
at javax.management.remote.JMXConnectorFactory.connect (JMXConnectorFactory.java:248)
at Main.main (Main.java:31)
at sun.reflect.NativeMethodAccessorImpl.invoke0 (Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke (NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke (DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke (Method.java:597)
at com.intellij.rt.execution.application.AppMain.main (AppMain.java:120)
Caused by: javax.naming.CommunicationException [Root exception is java.rmi.ConnectIOException]
java.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty
at com.sun.jndi.rmi.registry.RegistryContext.lookup (RegistryContext.java:101)
at com.sun.jndi.toolkit.url.GenericURLContext.lookup (GenericURLContext.java:185)
at javax.naming.InitialContext.lookup (InitialContext.java:392)
at javax.management.remote.rmi.RMIClient.findRMIServerJNDI (RMIClient.java:1886)
at javax.management.remote.rmi.RMIClient.findRMIServer (RMIClient.java:1856)
at javax.management.remote.rmi.RMIClient.connect (RMIClient.java:255)
...
... 7 more
Caused by: java.rmi.ConnectIOException: error during JRMP connection establishment; nested exception is
java.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty
at sun.rmi.transport.tcp.TCPChannel.createConnection (TCPChannel.java:286)
at sun.rmi.transport.tcp.TCPChannel.newConnection (TCPChannel.java:184)
at sun.rmi.server.UnicastRef.newCall (UnicastRef.java:322)
```

## Troubleshooting

```
at sun.rmi.registry.RegistryImpl_Stub.lookup(Unknown Source)
at com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:97)
... 12 more
Caused by: javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty
at sun.security.validator.PKIXValidator.<init>(PKIXValidator.java:57)
at sun.security.validator.Validator.getInstance(Validator.java:161)
at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.getValidator(X509TrustManagerImpl.java:118)
at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:135)
at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerImpl.java:147)
at com.sun.net.ssl.internal.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:386)
at com.sun.net.ssl.internal.ssl.ClientHandshaker.processMessage(ClientHandshaker.java:135)
at com.sun.net.ssl.internal.ssl.Handshaker.processLoop(Handshaker.java:593)
at com.sun.net.ssl.internal.ssl.Handshaker.process_record(Handshaker.java:529)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.readRecord(SSLSocketImpl.java:943)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.performInitialHandshake(SSLSocketImpl.java:847)
at com.sun.net.ssl.internal.ssl.SSLSocketImpl.writeRecord(SSLSocketImpl.java:654)
at com.sun.net.ssl.internal.ssl.AppOutputStream.write(AppOutputStream.java:100)
... 20 more
Caused by: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParameterException: the trustAnchors parameter must be non-empty
at java.security.cert.PKIXParameters.setTrustAnchors(PKIXParameters.java:183)
at java.security.cert.PKIXParameters.<init>(PKIXParameters.java:103)
at java.security.cert.PKIXBuilderParameters.<init>(PKIXBuilderParameters.java:87)
at sun.security.validator.PKIXValidator.<init>(PKIXValidator.java:55)
... 32 more
```

## Conditional stack trace

The stack trace below indicates that one of the following conditions is true:

- The username you specified does not exist.
- The password you specified is incorrect.
- The credentials field is not an Object array containing the username as a String in the array's first slot and the password as a char[] in the array's second slot.

```
Exception in thread "main" java.lang.SecurityException: Username and/or password is not valid
at com.tc.management.EnterpriseL2Management$1.authenticate(EnterpriseL2Management.java:114)
at javax.management.remote.rmi.RMIServerImpl.doNewClient(RMIServerImpl.java:213)
at javax.management.remote.rmi.RMIServerImpl.newClient(RMIServerImpl.java:180)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:303)
at sun.rmi.transport.Transport$1.run(Transport.java:159)
at java.security.AccessController.doPrivileged(Native Method)
at sun.rmi.transport.Transport.serviceCall(Transport.java:155)
at sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:535)
at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.java:790)
```

## Troubleshooting

```
at sun.rmi.transport.tcp.TCPTTransport$ConnectionHandler.run(TCPTTransport.java:649)
at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:895)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:918)
at java.lang.Thread.run(Thread.java:680)
at sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer(StreamRemoteCall.java:142)
at sun.rmi.transport.StreamRemoteCall.executeCall(StreamRemoteCall.java:233)
at sun.rmi.server.UnicastRef.invoke(UnicastRef.java:142)
at javax.management.remote.rmi.RMIServerImpl_Stub.newClient(Unknown Source)
at javax.management.remote.rmi.RMIClient.getRMIClient(RMIClient.java:127)
at javax.management.remote.rmi.RMIClient.connect(RMIClient.java:127)
at javax.management.remote.JMXConnectorFactory.connect(JMXConnectorFactory.java:248)
at Main.main(Main.java:31)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at com.intellij.rt.execution.application.AppMain.main(AppMain.java:120)
```

# Terracotta Tools Catalog

## Introduction

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.

## 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](#).

## Cluster Thread and State Dumps (debug-tool)

The Terracotta thread- and state-dump debug tool provides a way to easily generate debugging information that can be analyzed locally or forwarded to support personnel. The debug tool is useful in cases where firewalls block the use of standard JMX-based tools, or using the Developer Console cannot be convenient.

### Microsoft Windows

```
[PROMPT] %TERRACOTTA_HOME%\bin\debug-tool.bat <args>
```

### UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/debug-tool.sh <args>
```

where `<args>` are:

- `[-d]` – Takes a full state dump of all nodes in the cluster (servers and clients), logging output to each node's log. Note that the total amount of information generated by this option can be very large.
- `[-h]` – Prints brief help information to standard output.
- `[-n] <hostname>` – `(--hostname)` Allows you to specify a Terracotta server instance in the target cluster. If a host name is not specified with this argument, "localhost" is assumed.
- `[-p] <jmx-port>` – `(--jmlexport)` Specifies a JMX port on the target host. If a JMX port is not specified with this argument, "9520" is assumed.
- `[-u] <string>` – `(--username)` Specifies a JMX username (if JMX authorization is set up).
- `[-w] <string>` – `(--password)` Specifies a JMX password (if JMX authorization is set up).

## Cluster Thread and State Dumps (debug-tool)

Running the debug tool without the `-d` option produces a cluster thread dump (dumps taken from all nodes). The thread dumps are written to a zipped file called `cluster-dump.zip` which is saved in the working directory.

## 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

## Sessions Configurator (sessions-configuration)

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.

## 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.

## 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.

## 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>
```

## Archive Utility (archive-tool)

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>]

## 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>
```

## 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>
```

### 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](#).

Distributed Garbage Collector (run-dgc)

## 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 cycles 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>
```

## 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 [Terracotta Developer Console](#).

## 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 therefore useful only in development mode, where each DSO client provides its own configuration.

## Start and Stop Server Scripts (start-tc-server,stop-tc-server)

For production purposes, DSO clients should obtain their configuration from a Terracotta Server using the `tc.config` system property.

`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.

If the `stop-tc-server` script detects that the passive server in STANDBY state isn't reachable, it issues a warning and fails to shut down the active server. If failover is not a concern, you can override this behavior with the `--force` flag.

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>]
```

where `<args>` are:

- [f] <file-or-URL> – Specifies the configuration file to use, as a file path or URL.
- [--force] – Force shutdown of the active server.

In production mode, if the `stop-tc-server` script detects that the mirror server in STANDBY state isn't reachable, it issues a warning and fails to shut down the active server. If failover is not a concern, you can override this behavior with the `--force` flag.

- [n] <server-name> – The name of the server to shut down. Defaults to the local host.
- [s] – If the server is secured with a JMX password, then a username and password must be passed into the script.
- [u] – Specify the JMX username.
- [w] – Specify the JMX password.

## Stopping an SSL-Secured Server

Stop a server in a secure Terracotta cluster using the `stop-tc-server` script with the following arguments:

- `-f <tc-config-file>` — A valid path to the self-signed certificate must have been specified in the server's configuration file.
- `-u <username>` — The user specified must have the "admin" role.
- `-w <password>`

For more information, refer to [Setting Up Server Security](#).

## 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](#).

## 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&
```

## 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}/bin/server-stat.sh <args>
```

<args> are the same as shown for Microsoft Windows.

Example

## 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.health: OK
.0.0.role: ACTIVE
.0.0.state: ACTIVE-COORDINATOR
.0.0.jmxport: 9520
```

If no server is specified, by default the Server Status checks the status of localhost at JMX port 9520.

## 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](#).

## DSO Tools

NOTE:

The following subject matter covers aspects of core Terracotta DSO technology. DSO is recommended for \*advanced users only\*.

### 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&
```

### 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:

## Make Boot Jar Utility (make-boot-jar)

- ```
[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.

## 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.

## 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.

## 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.

Java Wrapper (dso-java)

## 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&
```

then 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.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.

# Introduction

The Terracotta Operations Runbook is a comprehensive guide to common failure scenarios, covering diagnosis, response, and recovery. Each failure scenario includes the following information:

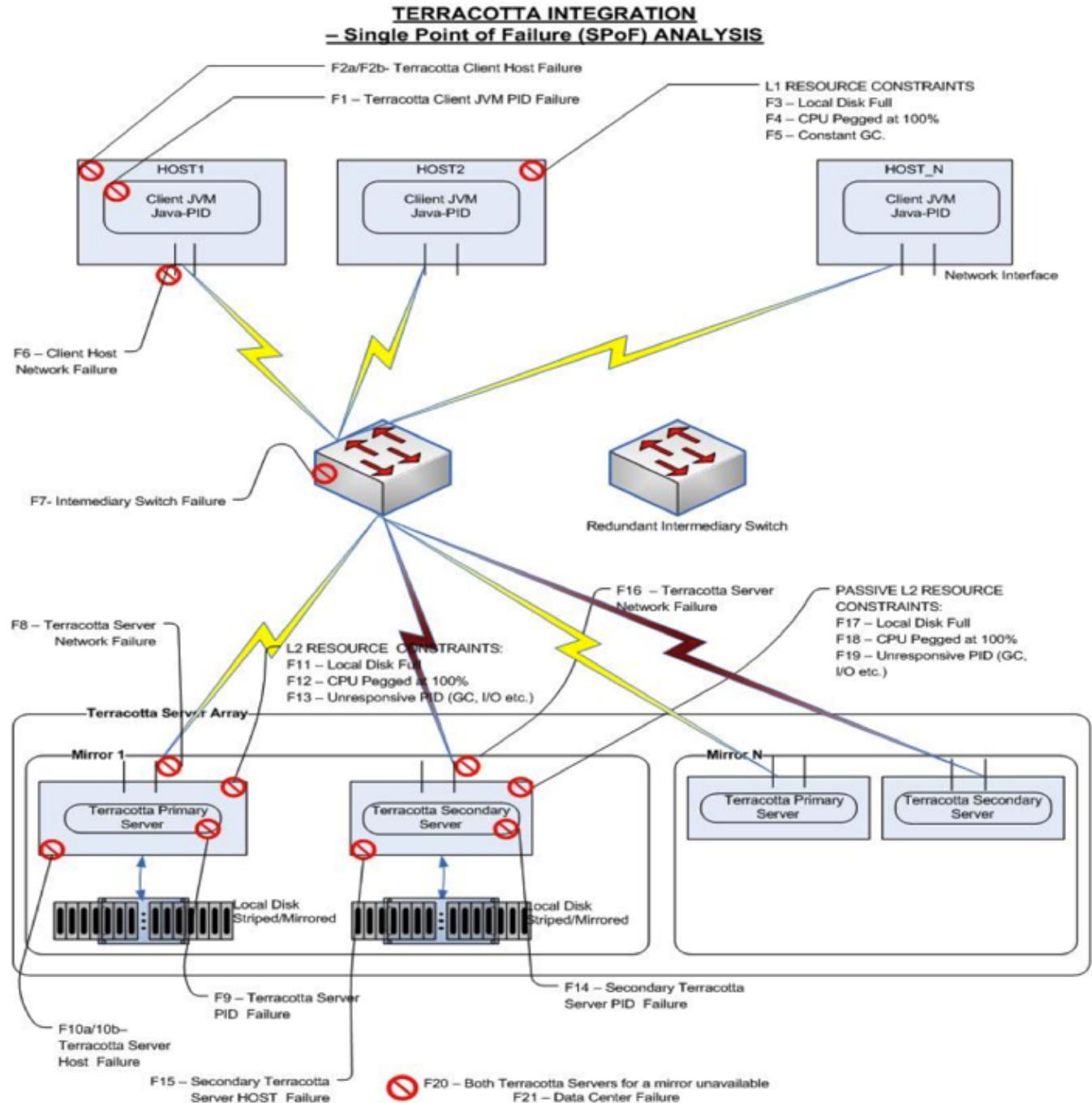
- Component identification (for example, Terracotta clients, Terracotta server instances, and networking components in a Terracotta deployment)
- Steps required for recovery or to restore HA
- Artifacts to be collected (logs, recovery time, etc.)

## Intended Audience

The Terracotta Operations Runbook is intended for operators engaged in monitoring and maintaining a Terracotta cluster in a production environment. Operators should use the Runbook both in training and during actual failures in a production environment.

# SPOF Analysis Diagram

The following diagram is a pictorial view of a typical Terracotta cluster and different failure points that are addressed in this document:



# Failure Scenarios

This page discusses "grey outages" (degraded characteristics) as well as "black and white" failures.

## Client (L1) Failures

### Loss of Terracotta-Client Java PID

#### Expected Behavior

- After PID loss, the log of Primary Terracotta Server (L2) for each mirror group prints 'DSO Server - Connection to [L1 IP:PORT] DISCONNECTED. Health Monitoring for this node is now disabled.'
- Slow down / Zero TPS at admin console for 15 seconds (L2-L1 reconnect) as the resources held by the L1 will not be released until then
- After 15 seconds terracotta server array ejects the L1 from cluster and primary L2 logs print 'shutdownClient() : Removing txns from DB :'
- Once L1 is ejected, Admin console does not show the failed L1 in client list and TPS recovers

#### Monitor

1. Observe latency in user request (some of the request might have to wait 15s).
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

#### Observation

L2 Active Log = WARN tc.operator.event -NODE : Server1 Subsystem: CLUSTER\_TOPOLOGY  
Message:Node ClientID[0] left the cluster

When= immediately from the loss of PID

Limit (with default values)= 0 seconds

For Reconnect properties enabled:

L2 Active Log = (same) When = after [l2.l1reconnect.timeout.millis] from the loss of PID

#### Expected Recovery Time

15 secs (L2-L1 reconnect)

#### Action to be taken

recycle client JVM

## Terracotta-Client Host Reboot

## Terracotta-Client Host Reboot

### Expected Behavior

- After PID loss, the Primary Terracotta Servers (L2) log prints 'DSO Server - Connection to [L1 IP:PORT] DISCONNECTED. Health Monitoring for this node is now disabled.'
- Slow down / Zero TPS at admin console for 15 seconds (L2-L1 reconnect) as the resources held by the L1 will not be released until then
- After 15 seconds terracotta server array ejects the L1 from cluster and prints 'shutdownClient() : Removing txns from DB : ' in L2 logs for each primary L2s.
- Once L1 is ejected, Admin console does not show the failed L1 in client list and TPS recovers

### Monitor

1. Observe latency in user request (some of the request might have to wait 15 secs).
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

### Observation

L2 Active Log = WARN tc.operator.event - NODE : Server1 Subsystem: HA Message: Node ClientID[0] left the cluster

When= after ping.idletime + (ping.interval \* ping.probes) + ping.interval

Limit (with default values)= 4 - 9 seconds

Limit is a measure of the time in which the process determines the case. Why it is a limit and not an absolute value?

This is because that there is a possibility that when the system encountered the problem it could be in one of the two states below.

State 1: All the components were in continuous conversation and thus the Health Monitoring has to factor in the first ping.idletime as a measure of detection of the problem.

State 2: All the components were connected to each other but the application load or the communication was such that there was a communication silence > ping.idletime. This means, that the system was doing Health Monitoring in the background already and the cluster was detected healthy at all times before this new problem arrived.

Therefore, it is possible that you may see the detection time as an interval within this limit.

All the expressions from here on show the maximum time it can take inclusive of the limiting ping.idletime. To get the limit interval just deduct the ping.idletime from the equations.

For Reconnect properties enabled:

L2 Active Log = (same)

When = after [l2.l1reconnect.timeout.millis] from the loss of PID

Terracotta-Client Host Reboot

### **Expected Recovery Time**

15 secs (L2-L1 reconnect)

### **Action to be taken**

Start Client-JVM after machine reboot (On Restart client rejoins the cluster.)

## **Terracotta-Client Host Extended Power Outage**

### **Expected Behavior**

text

### **Expected Recovery Time**

text

### **Action to be taken**

text

## **L1 Local Disk Full**

### **Expected Behavior**

Terracotta code should execute without any impact, except nothing will be logged in log files.

### **Monitor**

Whether application threads are able to proceed, as their ability to write to disk (e.g. logging) will be hampered.

### **Observation**

text

### **Expected Recovery Time**

As soon as disk usage falls back to normal.

### **Action to be taken**

Cleanup local disk to resume Terracotta Client Logging

## **L1 CPU Pegged**

## L1 CPU Pegged

### Expected Behavior

- Slow down in TPS at admin console because L1 will not be able to release resources (e.g. Locks) faster and the Terracotta Server Array (L2) will take more time to commit the transaction that are to be applied on this L1
- TPS recovers when CPU returns to normal. Run tests with difference intervals of high CPU usage (15s, 30s, 60s, 120s, 300s)

### Monitor

1. Observe latency in user request, some/all of them will be processed slower
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

### Observation

text

### Expected Recovery Time

As soon as CPU usage returns back to normal.

### Action to be taken

Analyze Root Cause and remedy.

## L1 Memory Pegged (Constant GC)

### Expected Behavior

Slow down/Zero in TPS at admin console because any resource (e.g. Locks) held by L1 will not be released until GC is over and terracotta server (L2) will not be able to commit transactions that are to be applied on this L1.

Case1: Full GC cycle less than 45 secs No message in L1/L2 logs. Admin console reflects normal TPS once L1 recovers from GC.

Case 2: Full GC cycle > 45 secs After 45 secs, L2 health monitoring declares L1 dead and prints this message in L2 logs 'INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl.TCGroupManager - L1:PORT is DEAD' After 45 secs, primary L2 ejects the L1 from cluster and prints 'shutdownClient() : Removing txns from DB :' in L2 logs. Once L1 is ejected Admin console does not show the failed L1 in client list. If the L1 recovers after 45 secs and tries to reconnect, the L2 refuses all connections and prints this message in L2 logs: INFO com.tc.net.protocol.transport.ServerStackProvider - Client Cannot Reconnect ConnectionID() not found. Connection attempts from the Terracotta client at [L1 IP:PORT] are being rejected by the Terracotta server array. Restart the client to allow it to rejoin the cluster. Many client reconnection failures can be avoided by configuring the Terracotta server array for "permanent-store" and tuning reconnection parameters. For more information, see <http://www.terracotta.org/ha>

## L1 Memory Pegged (Constant GC)

### Monitor

1. Observe latency in user request - some/all of them will be processed slower
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until the cluster recovers

### Observation

L2 Active Log = WARN com.tc.net.protocol.transport.ConnectionHealthCheckerImpl.TCGroupManager - 127.0.0.1:56735 might be in Long GC. Ping-probe cycles completed since last reply : 1 .... .... INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. TCGroupManager - localhost:56735 is DEAD

When= Detection in ping.idletime + (ping.interval \* ping.probes) + ping.interval Disconnection in (ping.idletime) + socketConnectCount \* [(ping.interval \* ping.probes) + ping.interval] Limit (with default values)= detection in 4 - 9 seconds, disconnection in 45 seconds

### Expected Recovery Time

Max allowed GC time at L1 = 'L2-L1 Health monitoring ' = 45 secs.

### Action to be taken

Analyze GC issues and remedy.

## L1-L2 Connectivity Failure

### L1 NIC Failure - Dual NIC Client Host

#### Expected Behavior

Slow down/Zero in TPS at admin console because any resource (e.g. Locks) held by L1 will not released and terracotta server(L2) will not able to commit transactions that are to be applied on this L1.

Case 1: Client host fails over to standby NIC within 14 seconds. No message in L1/L2 logs. TPS resumes to normal at admin console as soon L1 NIC is restored.

Case 2: Client host fails over to standby NIC after 14 seconds - After 14 secs, Terracotta Server Array health monitoring declares L1 dead and prints this message in L2 logs 'INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Server - L1 IP:PORT is DEAD' After 14 secs seconds primary L2 ejects the L1 from cluster and prints 'shutdownClient() : Removing txns from DB :' in L2 logs. Once L1 is ejected Admin console does not show the failed L1 in client list. If the L1 recovers after 14 secs and tries to reconnect, the L2 doesn't allow it to reconnect and prints this message in L2 logs: 'INFO com.tc.net.protocol.transport.ServerStackProvider - Client Cannot Reconnect ConnectionID() not found. Connection attempts from the Terracotta client at [L1 IP:PORT] are being rejected by the Terracotta server array. Restart the client to allow it to rejoin the cluster. Many client reconnection failures can be avoided by configuring the Terracotta server array for "permanent-store" and tuning reconnection parameters. For more information, see <http://www.terracotta.org/ha>

## L1 NIC Failure - Dual NIC Client Host

### Monitor

1. Observe latency in user request, some/all of them will be processed slower
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

### Observation

L1 Log = INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - Socket Connect to indev1.terracotta.lan:9510(callbackport:9510) taking long time. probably not reachable. [HealthChecker]  
INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - indev1.terracotta.lan:9510 is DEAD  
L2 Active Log = INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Server - Socket Connect to pbhardwa.terracotta.lan:52275(callbackport:52274) taking long time. probably not reachable. [HealthChecker]  
INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Server - pbhardwa.terracotta.lan:52275 is DEAD

When= Detection in ping.idletime + (ping.interval \* ping.probes) + ping.interval  
Disconnection in (ping.idletime) + (ping.interval \* ping.probes + socketConnectTimeout \* ping.interval) + ping.interval  
Limit (with default values)= detection in 4 - 9 seconds, disconnection in 14 seconds

### Expected Recovery Time

Max allowed NIC recovery at L1 = 'L2-L1 Health monitoring = 14 secs.

### Action to be taken

No action needed immediately. At some point fix failed NIC.

## Primary Switch Failure

### Expected Behavior

Terracotta code should execute without any impact, except nothing will be logged in log files.

### Monitor

If application threads are able to proceed as their ability to write to disk (e.g. logging) will be hampered

### Observation

text

### Expected Recovery Time

If switch fails such that primary L2 (of a mirror-group) is unreachable from hot-standby L2 of the same mirror group and all L1s.

- Zero TPS at admin console
- Max allowed Recovery time from switch failure = 'min ((L2-L1 health monitoring (14 secs)), (L1-L2 health monitoring (14 secs)), (L2-L2 health monitoring (14 secs)))' = 14 secs.

## **Primary Switch Failure**

- If failover to redundant switch occurs within 14 secs, cluster topology remains untouched and TPS resumes to normal at admin console after switch recovery
- If switch does not failover within 14 secs, hot-standby L2 starts election to become the primary and all L1 disconnect from primary L2.
- Hot-standby L2 will become primary after 19 secs (14 secs + 5 secs of election time) of switch failure.
- The complete recovery time will be more than 19 secs and exact time will depend on cluster runtime condition. Ideally cluster should recover completely within 25 secs.

If Switch fails such that primary and hot-standby L2 connectivity is intact, while L1s connectivity with primary L2 is broken - Zero TPS at admin console - Max allowed Recovery time from switch failure = L2-L2 Health Monitoring = 14 secs. - If failover to redundant switch occurs within 14 secs, cluster topology remains untouched and TPS resumes to normal at admin console after switch recovery - If switch does not failover within 14 secs, L2 quarantines all the L1s from cluster. After switch recovery all the L1s have to be restarted to make them rejoin the cluster.

### **Monitor**

1. Observe latency in user request as they are not processed until L1 recovers
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

### **Action to be taken**

No action needed immediately. Restore Switch at a later point.

## **Primary L2 NIC Failure - Dual NIC Terracotta Server Host**

### **Expected Behavior**

Expected behavior - Zero TPS at admin console

Case 1: TC server host fails over to standby NIC within 14 seconds - TPS resumes on admin console as soon as NIC recovery happens.

Case 2: TC server host does not fail over to standby NIC within 14 seconds, - After 14 secs all L1s disconnect from primary L2 and try connection with hot-standby L2. - After 14 secs, hot-standby L2 starts election to become primary and prints 'Starting Election to determine cluster wide ACTIVE L2' inside its logs. - After 19 secs, hot-standby L2 becomes primary L2 and prints 'Becoming State[ ACTIVE-COORDINATOR ]' inside its logs - Once hot-standby L2 becomes the primary, all L1s will reconnect to it. Cluster recovers when new primary log prints 'Switching GlobalTransactionID Low Water mark provider since all resent transactions are applied' and TPS resumes at admin console. - Once the old primary L2 recovers, it is zapped by the new primary L2.

### **Monitor**

1. Observe latency in user request as they are not processed until primary/hot-standby L2 recovers
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until TPS recovers

## Primary L2 NIC Failure - Dual NIC Terracotta Server Host

### Observation

L1 Log = INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - Socket Connect to indev1.terracotta.lan:9510(callbackport:9510) taking long time. probably not reachable. [HealthChecker]  
INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - indev1.terracotta.lan:9510 is DEAD  
L2 Passive Log = INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Server - Socket Connect to pbhardwa.terracotta.lan:52275(callbackport:52274) taking long time. probably not reachable. [HealthChecker]  
INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Server - pbhardwa.terracotta.lan:52275 is DEAD

When= Detection in ping.idletime + (ping.interval \* ping.probes) + ping.interval Disconnection in (ping.idletime) + (ping.interval \* ping.probes + socketConnectTimeout \* ping.interval) + ping.interval  
Passive become active in (ping.idletime) + (ping.interval \* ping.probes + socketConnectTimeout \* ping.interval) + ping.interval + Election Time Limit (with default values)= detection in 4 - 9 seconds, disconnection in 14 seconds, passive takes over in 19 seconds

### Expected Recovery Time

Max allowed recovery time = 'min ( (L2-L1 health monitoring (14 secs)), (L1-L2 health monitoring (14 secs)), (L2-L2 health monitoring (14 secs)) )' = 14 secs.

The complete recovery time will be more than 19 secs and exact time will depend on cluster runtime condition. Ideally cluster should recover completely within 25 seconds.

### Action to be taken

No action needed immediately. At some point FIX the failed NIC by after forcing a failover to the standby Terracotta Server.

## Terracotta Server (L2) Subsystem failure

### Hot-standby L2 Available - Primary L2 Java PID Exits

#### Expected Behavior

Zero TPS at admin console After 15 seconds, hot-standby L2 starts election to become the primary and print 'Starting Election to determine cluster wide ACTIVE L2' inside its logs. All L1 disconnects from primary L2 after 15 secs and connect to old hot-standby L2 when it becomes primary. After 20 secs hot-standby becomes primary and prints 'Becoming State[ ACTIVE-COORDINATOR ]' inside its logs Once hot-standby L2 becomes primary, all L1 will reconnect to hot-standby. Cluster recovers when new primary log prints 'Switching GlobalTransactionID Low Water mark provider since all resent transactions are applied' and TPS resumes at admin console.

#### Monitor

1. Observe latency in user request, none/all of them will not complete until L2-L2 reconnect interval
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until TPS recovers

## **Hot-standby L2 Available - Primary L2 Java PID Exits**

### **Observation**

L2 Passive Log = WARN tc.operator.event - NODE : Server1 Subsystem: CLUSTER\_TOPOLOGY  
Message: Node Server2 left the cluster When = Immediately when PID exits L1 Log = INFO  
com.tc.net.protocol.transport.ConnectionHealthCheckerImpl: DSO Client - Connection to [localhost:8510]  
DISCONNECTED. Health Monitoring for this node is now disabled. When= Immediately when PID exits  
Limit = Detection Immediate, L2 PassiVe takes over as Active after Election Time (default = 5 seconds) For  
Reconnect properties enabled: L2 Passive Log = (same) When = after  
[l2.nha.tcgroupcomm.reconnect.timeout] from the loss of PID

### **Expected Recovery Time**

The complete recovery time will be more than 20 secs (L2-L2 reconnect + Election time) and exact time will depend on cluster runtime condition. Ideally cluster should recover completely within 25 seconds.

### **Action to be taken**

No action needed immediately (given failover). Restart L2 (it will now become the hot-standby).

## **Hot-standby L2 Available - Primary L2 Host Reboot**

### **Expected Behavior**

Clients fail over to Hot-standby L2, which then becomes primary. Once Primary L2 comes back (i.e. is restarted after the machine reboot sequence), it will join the cluster as hot-standby.

### **Observation**

text

### **Expected Recovery Time**

Same as F9

### **Action to be taken**

No action needed immediately. Restart L2 after reboot (it will now become the hot-standby)

## **Hot-standby L2 Available - Primary Host Unavailable: Extended Power Outage**

### **Expected Behavior**

Zero TPS at admin console After 14 seconds, hot-standby starts election to become primary and print 'Starting Election to determine cluster wide ACTIVE L2' inside its logs. All L1 disconnects from primary L2 after 14 secs and connect to old hot-standby L2 when it becomes primary.

After 19 secs hot-standby becomes primary and prints 'Becoming State[ ACTIVE-COORDINATOR ]' inside its logs Once hot-standby L2 becomes primary, all L1s will reconnect to hot-standby. Cluster recovers when new primary log prints 'Switching GlobalTransactionID Low Water mark provider since all resent

**Hot-standby L2 Available - Primary Host Unavailable: Extended PowerOutage**

transactions are applied' and TPS resumes at admin console.

## **Monitor**

1. Observe latency in user request
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until TPS recovers

## **Observation**

text

## **Expected Recovery Time**

L1 will detect failure after (L1-L2 health monitoring (14 secs)) Hot-standby L2 will detect failure after (L2-L2 health monitoring (14 secs) The complete recovery time will be more than 19 secs (L2-L2 health monitoring (14 secs) + Election time (5 secs) and exact time will depend on cluster runtime condition. Ideally cluster should recover completely within 25 seconds.

## **Action to be taken**

Same as F10 (a)

# **Primary L2 Local Disk Full**

## **Expected Behavior**

Same as F9

## **Observation**

15 secs.

## **Expected Recovery Time**

text

## **Action to be taken**

No action needed immediately (given failover to Hot-standby L2) Clean up disk and restart services (this L2 will now be hot-standby)

# **Primary L2 - CPU Pegged at 100%**

## **Expected Behavior**

Slow down in TPS at admin console because L2 will take more time to process transactions TPS recovers when CPU returns to normal. Run tests with difference intervals of high CPU usage (15s, 30s, 60s, 120s, 300s)

## Primary L2 - CPU Pegged at 100%

### Monitor

1. Observe latency in user request as they are processes slower until CPU recovers
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

### Observation

text

### Expected Recovery Time

As soon as CPU usage returns back to normal

### Action to be taken

Root-cause analysis and fix (Thread dumps needed if escalated to TC)

## Primary L2 Memory Pegged - Excessive GC or I/O (Host Reachable but PID Unresponsive)

### Expected Behavior

Zero TPS at admin console as primary L2 cannot process any transaction.

Case 1: GC cycle < 45 secs - L1 and hot-standby L2 log will display 'WARN com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. TCGroupManager - L2 might be in Long GC. GC count since last ping reply :', if L2 is in GC for more than 9s. - TPS returns to normal at admin console as soon as primary L2 recovers from GC

Case 3: GC cycle > 45s - After 45s, hot-standby L2 declares primary L2 dead. - Hot-standby L2 prints '[HealthChecker] INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. TCGroupManager - L2 IP: PORT is DEAD' in its logs message. - After 45 secs, hot-standby starts election to become primary and print 'Starting Election to determine cluster wide ACTIVE L2' inside its logs. - After 50 secs hot-standby becomes primary and prints 'Becoming State[ ACTIVE-COORDINATOR ]' inside its logs.

- After 57 secs, all L1's declare the old primary L2 dead and print '[HealthChecker] INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - L2 IP:PORT is DEAD' message in their logs.
- After 57 secs, all L1 disconnect from old primary L2 and try connection with old hot-standby L2 (which should have become primary now).
- Once hot-standby L2 becomes primary, all L1s will reconnect to hot-standby. Cluster recovers when new primary log prints 'Switching GlobalTransactionID Low Water mark provider since all resent transactions are applied'.
- Once old primary L2 recovers from GC, it is zapped by the new primary L2.

### Monitor

1. Observe latency in user request as none of them is processed until primary L2 recovers from GC or hot-standby L2 takes over

## **Primary L2 Memory Pegged - Excessive GC or I/O (Host Reachable but PID Unresponsive)**

2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

### **Observation**

L1 Log = WARN com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 might be in Long GC. GC count since last ping reply : 1 ... .... But its too long. No more retries [HealthChecker] INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 is DEAD When = Detection in ping.IdleTime + l1.healthcheck.l2.ping.probes\* ping.interval + ping.interval Dead in ping.IdleTime + l1.healthcheck.l2.socketConnectCount \* (l1.healthcheck.l2.ping.probes \* ping.interval + ping.interval) Limit = Detect in 5-9 seconds, Dead in 57 L2 Passive Log = WARN com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 might be in Long GC. GC count since last ping reply : 1 ... .... But its too long. No more retries [HealthChecker] INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 is DEAD When = Detection in ping.IdleTime + l2.healthcheck.l2.ping.probes\* ping.interval + ping.interval Dead in ping.IdleTime + l2.healthcheck.l2.socketConnectCount \* (l2.healthcheck.l2.ping.probes \* ping.interval + ping.interval) Limit = Detect in 5 -9 seconds, Dead in 45 L2 passive takes over as Active after Dead Time + Election Time

### **Expected Recovery Time**

Max allowed GC time = 'min ((L1-L2 health monitoring (57 secs)), (L2-L2 health monitoring(45 secs)))' = 45 secs The max complete recovery time will be more than 57 secs and exact time will depend on cluster runtime condition. Ideally cluster should recover completely within 65 seconds.

### **Action to be taken**

Root cause analysis to avoid this situation (e.g. more Heap, GC Tuning, etc. based on what the root-cause analysis dictates).

## **Primary L2 Available - Hot-standby L2 PID Unresponsive**

### **Expected Behavior**

Slow/Zero TPS at admin console as primary L2 can not commit the transactions to hot-standby L2 Primary L2 prints 'Connection to [Passive L2 IP:PORT] DISCONNECTED. Health Monitoring for this node is now disabled.' In logs as soon as hot-standby L2 fails After 15s, primary L2 quarantines hot-standby L2 from cluster, prints 'NodeID[Passive L2 IP:PORT] left the cluster' in logs and TPS returns to normal at admin console

### **Monitor**

1. Once hot-standby is recycled it joins the cluster back. Monitor the time hot-standby L2 takes to move to PASSIVE-STANDBY standby state. Until hot-standby L2 moves to PASSIVE-STANDBY, cluster has single point of failure as hot-standby L2 cannot take over in case primary L2 fails.
2. Observe latency in user request (some/all of the requests might have to wait until L2-L2 reconnect interval).
3. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until L2-L2 reconnect interval.

## **Primary L2 Available - Hot-standby L2 PID Unresponsive**

### **Observation**

L2 Active Log = WARN com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 might be in Long GC. GC count since last ping reply : 1 ... .... But its too long. No more retries [HealthChecker] INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 is DEAD When = Detection in ping.IdleTime + l2.healthcheck.l2.ping.probes\* ping.interval + ping.interval Dead in ping.IdleTime + l2.healthcheck.l2.socketConnectCount \* (l2.healthcheck.l2.ping.probes \* ping.interval + ping.interval) Limit = Detect in 5 -9 seconds, Dead in 45

### **Expected Recovery Time**

Recovery Time = [L2-L2 Reconnect] = 15 secs

### **Action to be taken**

Restart hot-standby L2 (blow away dirty BDB database before restart) in case of PID failure /Host Failure

## **Primary L2 Available - Hot-standby L2 Host Failure**

### **Expected Behavior**

Slow/Zero TPS at admin console as primary L2 can not commit the transactions to hot-standby L2 After 14 secs, Primary L2 prints 'Connection to [Passive L2 IP:PORT] DISCONNECTED. Health Monitoring for this node is now disabled.' in logs. After 14 secs, Primary L2 quarantines hot-standby L2 from cluster, prints 'NodeID [Passive L2 IP:PORT] left the cluster' in logs and TPS returns to normal at admin console

### **Monitor**

1. Once hot-standby is recycled it joins the cluster back. Monitor the time hot-standby L2 takes to move to PASSIVE-STANDBY standby state. Until hot-standby L2 moves to PASSIVE-STANDBY, cluster has single point of failure as hot-standby L2 can not take over in case primary L2 fails.
2. Observe latency in user request (some/all of the requests might have to wait until L2-L2 reconnect interval).
3. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until L2-L2 reconnect interval.

### **Observation**

text

### **Expected Recovery Time**

Recovery Time = [L2-L2 health monitoring] = 14 secs

### **Action to be taken**

Restart hot-standby L2 (blow away dirty BDB database before restart - not needed in case of 2.7.x or above) in case of PID failure /Host Failure

Primary L2 Available - Hot-standby L2 NIC Failure (Dual NIC Host)

## **Primary L2 Available - Hot-standby L2 NIC Failure (Dual NIC Host)**

### **Expected Behavior**

Slow/Zero TPS at admin console as primary L2 can not commit the transactions to hot-standby L2 Case 1: Hot-standby-L2 host failover to secondary NIC within 14 secs - No impact on cluster topology. TPS at admin console resumes as soon as NIC is restored at hot-standby L2 Case 2: Secondary host does not failover to standby NIC in 14 secs - After 14 secs Primary L2 prints 'Connection to [indev2.terracotta.lan:46133] DISCONNECTED. Health Monitoring for this node is now disabled.' - After 14 secs, Primary L2 quarantines hot-standby L2 from cluster, prints 'NodeID[Passive L2 IP:PORT] left the cluster' in logs and TPS returns to normal at admin console

### **Monitor**

1. Once hot-standby is recycled, it rejoins the cluster. Monitor the time hot-standby L2 takes to move to PASSIVE-STANDBY standby state. Until hot-standby L2 moves to PASSIVE-STANDBY, cluster has single point of failure as hot-standby L2 cannot take over in case primary L2 fails.
2. Observe latency in user request (some/all of the requests might have to wait until L2-L2 reconnect interval).
3. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until L2-L2 reconnect interval.

### **Observation**

L2 Active Log = INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. TCGroupManager - Socket Connect to indev1.terracotta.lan:8530(callbackport:8530) taking long time. probably not reachable. INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. TCGroupManager - indev1.terracotta.lan:8530 is DEAD When = Detection in ping.idletime+ ping.probes\* ping.interval + ping.interval Dead in ping.idletime + ping.probes\* ping.interval + l2.healthcheck.l2.socketConnectTimeout\* ping.interval Limit = 9 -14 seconds (with default values)

### **Expected Recovery Time**

Recovery Time = [L2-L2 heath monitoring] = 14 secs

### **Action to be taken**

If quarantined from cluster, Restart hot-standby L2 (blow away dirty BDB database before restart - not needed for 2.7.x) in case of PID failure /Host Failure

## **Primary L2 Available - Hot-standby L2 "Gray" Issue (CPU High)**

### **Expected Behavior**

Slow TPS at Admin-Console as primary L2 takes more time to commit transaction at hot-standby L2 TPS recovers when CPU returns to normal. Run tests with difference intervals of high CPU usage (15s, 30s, 60s, 120s, 300s)

## **Primary L2 Available - Hot-standby L2 "Gray" Issue (CPU High)**

### **Monitor**

1. Observe latency in user request
2. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

### **Observation**

text

### **Expected Recovery Time**

Recovers as soon as CPU becomes normal

### **Action to be taken**

Analyze root-cause and resolve high-CPU issue at Hot-standby L2.

## **Primary L2 Available - Hot-standby L2 "Gray" Issue (Memory Pegged)**

### **Expected Behavior**

Slow/Zero TPS at admin console as primary L2 can commit transaction locally but cannot commit transaction at hot-standby L2

Case 1: GC cycle < 45 secs - Primary L2 log will display 'WARN com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. TCGroupManager - L2 might be in Long GC. GC count since last ping reply :', if L2 is in GC for more than 9 secs. - TPS returns to normal at admin console as soon as hot-standby L2 recovers from GC

Case 2: GC cycle > 45 secs - After 45 secs primary L2 health monitoring declares hot standby L2 dead. - Primary L2 prints 'HealthChecker] INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. TCGroupManager -[Passive L2 IP: PORT] is DEAD' in its logs message - After 45 seconds, primary L2 quarantines hot-standby L2 from cluster, prints 'NodeID[Passive L2 IP:PORT] left the cluster' in logs and TPS returns to normal at admin console.

### **Monitor**

1. Once hot-standby is recycled it rejoins the cluster. Monitor the time hot-standby L2 takes to move to PASSIVE-STANDBY standby state. Until hot-standby L2 moves to PASSIVE-STANDBY, cluster has single point of failure, as hot-standby L2 cannot take over in case primary L2 fails.
2. Backlog at application queue and observe backlog recovery time. Ensure backlog recovery time is within applications acceptable range
3. Observe latency in user request
4. GC and Heap usage at other L1s. Because of application level backlog, heap usage might increase until cluster recovers

## **Primary L2 Available - Hot-standby L2 "Gray" Issue (Memory Pegged)**

### **Observation**

L2 Active Log = WARN com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 might be in Long GC. GC count since last ping reply : 1 ... .... But its too long. No more retries [HealthChecker] INFO com.tc.net.protocol.transport.ConnectionHealthCheckerImpl. DSO Client - localhost:9510 is DEAD When = Detection in ping.IdleTime + l2.healthcheck.l2.ping.probes\* ping.interval + ping.interval Dead in ping.IdleTime + l2.healthcheck.l2.socketConnectCount \* (l2.healthcheck.l2.ping.probes \* ping.interval + ping.interval) Limit = Detect in 5 -9 seconds, Dead in 45

### **Expected Recovery Time**

Max Recovery Time = [L2-L2 health monitoring] = 45 secs

### **Action to be taken**

Root-cause analysis and fix needed for Memory on L2 getting pegged - (the actual action to be taken is fairly varied in this case, depending on the symptoms and analysis)

## **Primary L2 Available - Hot-standby L2 "Grey" Issue (Disk Full)**

### **Expected Behavior**

Same as F14.

### **Observation**

text

### **Expected Recovery Time**

Same as F14.

### **Action to be taken**

Hot-standby process dies with BDB errors. Restart hot-standby l2.

## **Primary and Hot-standby L2 Failure**

### **Expected Behavior**

All application threads that need DSO lock from the TC Server or those that are writing "Terracotta-transactions" with transaction buffer full will block. Once mirror-group(s) is restored, all the L1s connected to it before failure will reconnect and normal application activity resumes.

### **Observation**

text

Primary and Hot-standby L2 Failure

### **Expected Recovery Time**

Depends on when the Terracotta Server Array is recycled.

### **Action to be taken**

Not designed for N+1 failure. Restart mirror group(s) primary and hot-standby after collecting artifacts for root-cause analysis.

## **Other Failures**

### **Data Center Failure**

#### **Expected Behavior**

Not designed for N+1 failure. Restart mirror group(s) primary and hot-standby after collecting artifacts for root-cause analysis.

#### **Observation**

text

#### **Expected Recovery Time**

Minutes

#### **Action to be taken**

Once Data-Center is restored, restart Terracotta Server Array. Then restart L1 Nodes. Cluster state will be restored to point of outage.

# Recovery Scenarios

The recovery scenarios in the following sections are explained assuming we are using the default health check settings from tc.properties:

```
12.healthcheck.11.ping.idletime = 5000
12.healthcheck.11.ping.interval = 1000
12.healthcheck.11.ping.probes = 3
12.healthcheck.11.socketConnectTimeout = 5
12.healthcheck.11.socketConnectCount = 10

12.healthcheck.12.ping.idletime = 5000
12.healthcheck.12.ping.interval = 1000
12.healthcheck.12.ping.probes = 3
12.healthcheck.12.socketConnectTimeout = 5
12.healthcheck.12.socketConnectCount = 10

11.healthcheck.12.ping.idletime = 5000
11.healthcheck.12.ping.interval = 1000
11.healthcheck.12.ping.probes = 3
11.healthcheck.12.socketConnectTimeout = 5
11.healthcheck.12.socketConnectCount = 13
```

## 1. Default health monitoring parameter in tc.properties

- ◆ L1 - L2 detection of failure
  - ◊ 109 secs, if L2 is reachable and L1 can initiate a new socket connection. This basically allows a max of 109 secs GC on L2.
  - ◊ 13 secs, if connectivity to L2 is broken and L1 can not create a new socket connection to L2
- ◆ L2 - L2 detection of failure
  - ◊ If active (passive) L2 is reachable and passive(active) L2 can initiate a new socket connection = 85 secs
  - ◊ If connectivity to active (passive) L2 is broken and passive (active) L2 can not create a new socket connection to active (passive) L2 = 13 secs
- ◆ L2 - L1 detection of failure
  - ◊ 85 secs, if L1 is reachable and L2 can initiate a new socket connection. This allows a maximum of 85 secs GC on L1.
  - ◊ 13 secs, if the connectivity to L1 is broken and L2 cannot create a new socket connection.
  - ◊ In case L2 is not able to initiate a socket connection during the first connection cycle due to firewall settings etc., socket connection failure message would be printed in the server logs and all the L2 - L1 health check properties for this particular client will be multiplied by a factor of 10.

## 2. Reconnect properties

- ◆ L2 - L1 reconnect parameters
  - ◊ l2.l1reconnect.enabled = true (default is false)
  - ◊ l2.l1reconnect.timeout.millis = 15000 (default is 5000)
- ◆ L2 - L2 reconnect properties
  - ◊ l2.nha.tcgroupcomm.reconnect.enabled = true (default is false)
  - ◊ l2.nha.tcgroupcomm.reconnect.timeout = 15000 (default is 2000)

With the above parameters set:

## Recovery Scenarios

- Max GC allowed at L1 before it is quarantined from cluster = L2-L1 health monitoring = 85s
- Max allowed GC at passive L2 before it is quarantined by active L2 = L2-L2 health monitoring = 85 secs
- Max allowed GC at active L2 before
  - ◆ Passive takes over, is = (L2-L2 health monitoring(85 secs) + Election time(5 secs)) = 90 secs
  - ◆ L1 disconnects from active L2 and tries connection with another L2, is = L1-L2 health monitoring = 13 secs

# A Brief on Terracotta Health Monitoring

Terracotta Health Monitoring module takes care to figure the state of connectedness of all the Terracotta entities in a clustered deployment. Health Monitoring automatically detects various failure scenarios that may occur in a cluster and takes corrective actions to ensure that the cluster participants follow the necessary failover protocols to ensure the cluster is highly available at all points of time. This works independently of the application threads and therefore adds no performance overhead to the application.

## Variables and Usage

The Health Monitoring module exposes a few tunable parameters through the Terracotta configuration file. These parameters give the necessary flexibility to the deployment scheme for setting fine-grained tolerance values for various perceived failure scenarios.

The Health Monitoring goes on in the following directions:

1. Between Terracotta Clients and Terracotta Active Server (L1 -> L2)
2. Between Terracotta Active Server and Terracotta Passive Server (L2 -> L2)
3. Between Terracotta Active Server and Terracotta Clients (L2 -> L1)

Following are the parameters:

| Variable Name                                                                                                              | Description                                                                                                                                                                                                                                                                                                                       |
|----------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| l2.healthcheck.l1.ping.enabled<br>l2.healthcheck.l2.ping.enabled<br>l1.healthcheck.l2.ping.enabled                         | 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. |
| l2.healthcheck.l1.ping.interval<br>l2.healthcheck.l2.ping.interval<br>l1.healthcheck.l2.ping.interval                      | If no response is received to a ping probe, the time (in milliseconds) that HealthChecker waits between retries.                                                                                                                                                                                                                  |
| l2.healthcheck.l1.ping.probes<br>l2.healthcheck.l2.ping.probes<br>l1.healthcheck.l2.ping.probes                            | If no response is received to a ping probe, the maximum number (integer) of retries HealthChecker can attempt.                                                                                                                                                                                                                    |
| l2.healthcheck.l1.socketConnect<br>l2.healthcheck.l2.socketConnect<br>l1.healthcheck.l2.socketConnect                      | 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.                                                                                                          |
| l2.healthcheck.l1.socketConnectTimeout<br>l2.healthcheck.l2.socketConnectTimeout<br>l1.healthcheck.l2.socketConnectTimeout | 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 ping.interval by this value.                                     |
| l2.healthcheck.l1.socketConnectCount<br>l2.healthcheck.l2.socketConnectCount<br>l1.healthcheck.l2.socketConnectCount       | 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.                                                                                                                                  |

## Variables and Usage

|                                      |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|--------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| l2.nha.tcgroupcomm.reconnect.enabled | Enables a server instance to attempt reconnection with its peer server instance after a disconnection is detected. Default: false                                                                                                                                                                                                                                                                                                                                                    |
| l2.nha.tcgroupcomm.reconnect.timeout | Enabled if l2.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                                                                                                                                                                                                                                                                            |
| l2.ll1reconnect.enabled              | Enables a client to rejoin 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 rejoin the cluster, the client emits a mismatch error and exits. Default: true |
| l2.ll1reconnect.timeout.millis       | Enabled if l2.ll1reconnect.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.                                                                   |

## A Word about Reconnect Properties

The Reconnect properties lend another useful characteristics to the clustered deployment in case of small outages. If any Terracotta clustered entity happens to suffer a sudden network socket disruption, all the other entities that are monitoring its health get an EOF at their sockets instantly. In a normal case, when reconnect properties are not used, the entity which crashed is immediately quarantined from the cluster.

With reconnect properties enabled, all other cluster participants open a small window of opportunity for the crashed component to come back up alive and join the cluster. So this prohibits the Health Monitoring from severing the connection before the reconnection window closes on itself.

This works autonomously from the Health Monitoring process. These two processes work mutually exclusively. So, if Health Monitoring declares any cluster participant entity as dead, reconnect properties do not get applied to it (as in once declared dead, no reconnect time window will be opened). On the other hand, the Health Monitoring does not kick in once the reconnect window is over.

## Tolerance Formula and Flowchart

The final tolerance formula is denoted as:

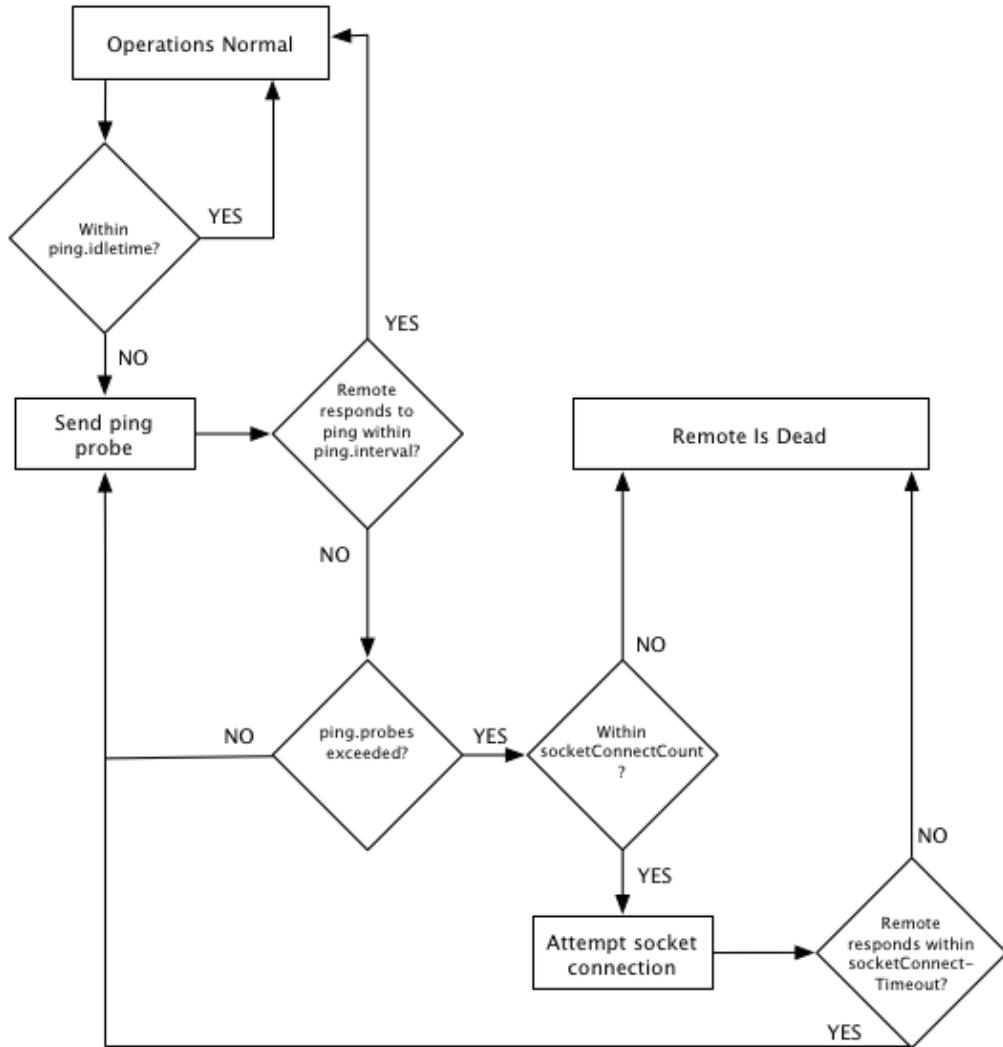
```
Max Time = (ping.idletime) + socketConnectCount * [(ping.interval * ping.probes)  
+ (socketConnectTimeout * ping.interval)]
```

OR

```
Max Time = reconnect.timeout.millis (in certain cases only)
```

The Health Monitoring module determines the other cluster entities' health based on the following flowchart process:

## Tolerance Formula and Flowchart



Depending on the deployment network's reliability and the degree of tolerance that is required, these parameters are tuned to give the desired result.

For more details, review the [HealthChecker](#) documentation.

## Assumptions

1. All results in this document are taken considering that if clustered peer is alive, then it is able to accept connection with in ping.interval timeframe
2. All numeric values conform to the default values inside the tc-config.xml

# How to Report an Issue to Terracotta

Note: the following process is for paying customers of Terracotta. If you are using the open source versions of the projects, please visit the [Terracotta forums](#) and post your questions there.

Your contract with Terracotta includes definitions that stipulate how issues are to be handled. These stipulations are based on issue level:

- For P1 and P2 issues: Call the Support Hotline (1-415-738-4000, Option 2/1). Terracotta will contact you within the stipulated SLA of your Enterprise Subscription.
- For P3 and P4 issues: Email to [customersupport@terracottatech.com](mailto:customersupport@terracottatech.com). Terracotta will contact you within the stipulated SLA of your Enterprise Subscription.

When you report an issue to Terracotta, be sure to provide the information shown in the Artifacts List. You can use FTP to deliver the artifacts (see below, Delivering Artifacts to Terracotta).

## Artifacts List

- Problem Summary
- Problem Description
- List of Terracotta Clients
  - ◆ Include hardware and software platform specs.
- Terracotta Client Logs
- Std-out Logs of client JVM
- JStat Monitoring on Terracotta Clients Output Logs
  - ◆ This output is needed to characterize GC on Terracotta clients.
- List of Terracotta Server Instances
  - ◆ Include hardware and software platform specs.
- JStat Monitoring on Terracotta Server Logs
  - ◆ This output is needed to characterize GC on Terracotta server instances.
- Terracotta Server Instances Logs
- Output of ls -al 'terracotta objectdb directory'
- Screen shot of Terracotta Administrator Console Tabs
- Classes, DGC, Roots, and Runtime statistics.
- CSV file generated by monitoring tool
- Thread Dumps on Terracotta Server Instances and the Offending Terracotta Client
  - ◆ One set every 15 seconds, at least 4 times.
- (Optional) Thread Dumps on Other Terracotta Clients - 1 set every 15s, at least 4 times
  - ◆ One set every 15 seconds, at least 4 times.
- (Optional) Output of Lock Profiling Session
- (Optional) Objectdb Data File

## Delivering Artifacts to Terracotta

You can deliver the artifacts in the Artifacts List using FTP. Use the Terracotta FTP site [ftp.terracottatech.com](ftp://ftp.terracottatech.com) and the name and password supplied to you by Terracotta. From a command line, enter:

```
ftp ftp.terracottatech.com
```

## Delivering Artifacts to Terracotta

You will be prompted for a name and then a password.

# Developing Applications With the Terracotta Toolkit

## Introduction

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 1.6 JVM or higher and requires no boot-jars, agents, or container-specific code.

The Terracotta Toolkit is available with Terracotta kits version 3.3.0 and higher.

## 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</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>
```

## 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.

# 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.

## 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;
...
// Start the client.
TerracottaClient client = new TerracottaClient("localhost:9510");

// Make the Toolkit available to your application.
ClusteringToolkit toolkit = client.getToolkit();
...
```

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.

## 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.

### Toolkit Data Structures and Serialization

Only the following types can be put into Toolkit data structures:

- 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

## Toolkit Data Structures and Serialization

- Arrays of any of the types listed above

To add other types to a Toolkit data structure, serialize it and then put the resulting byte array into the data structure.

## 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.

## 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.

## Cluster Information

The Terracotta Toolkit allows you to access cluster information for monitoring the nodes in the cluster, as

## Cluster Information

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);
```

## 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.

## Locks

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.

## 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.
```

## Clustered Barriers

```
Barrier clusteredBarrier = toolkit.getBarrier(String barrierName, int numberOfParties);
```

## 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)`.

## Destroying Clustered Terracotta Toolkit Objects

You can use the `unregister` methods available in the interface `org.terracotta.api.ClusteringToolkitExtension` to destroy clustered Toolkit objects. The following example shows how to unregister a clustered Map:

```
TerracottaClient client = new TerracottaClient("localhost:9510");
ClusteringToolkit toolkit = client.getToolkit();

Map map = toolkit.getMap("myMap");

// The instance must be cast to ClusteringToolkitExtension.
ClusteringToolkitExtension toolkitExtension = (ClusteringToolkitExtension) toolkit;

// The Terracotta cluster no longer has knowledge of myMap.
toolkitExtension.unregisterMap("myMap");
```

Do not attempt to continue using clustered Toolkit objects that have been unregistered. Doing so can lead to unpredictable results.

# Terracotta Toolkit Reference

This section describes functional aspects of the Terracotta Toolkit.

## 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.

## 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 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 `11.socket.connect.timeout` (see [First-Time Client Connection](#)).

## 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.

## 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 . In this case, two Terracotta clients will be running with every Tomcat instance.

## 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.

## Clients Sharing a Node ID

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.

# Working with 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 [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.

## 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 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>
    <dependency>
        <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.

## 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:

## Using the tc-maven Plugin

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

The following is an abbreviated list of goals available with the tc-maven plugin:

| Goal           | Function                                               |
|----------------|--------------------------------------------------------|
| tc:help        | Print help.                                            |
| tc:start       | Start the Terracotta server.                           |
| tc:stop        | Stop the Terracotta server.                            |
| tc:restart     | Restart the Terracotta server.                         |
| tc:dev-console | Start the Terracotta Developer Console.                |
| tc:run         | Start multiple Terracotta server and client processes. |
| tc:clean       | Clean Terracotta data and logs directories.            |
| tc:terminate   | Stop web servers that started with tc:run.             |

Execute tc:help to print more detailed information on these goals.

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 3.5.2 of this JAR file:

```
mvn install:install-file -Dfile=terracotta-ee-3.5.2.jar \
-DpomFile=terracotta-ee-3.5.2.pom -Dpackaging=jar -Dversion=3.5.2
```

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.

## 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>
```

## Working With Terracotta SNAPSHOT Projects

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

## 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>
```

# Terracotta Clustering Best Practices

The following sections contain advice on optimizing the operations of a Terracotta cluster.

## Analyze Java Garbage Collection (GC)

Long GC cycles are one of the most common causes of issues in a Terracotta cluster because a full GC event pauses all threads in the JVM. Servers disconnecting clients, clients dropping servers, OutOfMemoryErrors, and timed-out processes are just some of the problems long GC cycles can cause.

Having a clear understanding of how your application behaves with respect to creating garbage, and how that garbage is being collected, is necessary for avoiding or solving these issues.

### Printing and Analyzing GC Logs

The most effective way to gain that understanding is to create a profile of GC in your application by using tools made for that purpose. Consider using JVM options to generate logs of GC activity:

- `-verbose:gc`
- `-Xloggc:<filename>`
- `-XX:+PrintGCDetails`
- `-XX:+PrintGCTimeStamps`

Apply an appropriate parsing and visualization tool to GC log files to help analyze their contents.

### Observing GC Statistics With jstat

One way to observe GC statistics is by using the Java utility `jstat`. The following command will produce a log of GC statistics, updated every ten seconds:

```
jstat -gcutil <pid> 10 1000000
```

An important statistic is the Full Garbage Collection Time. The difference between the total time for each reading is the amount of time the system was paused. A jump of more than a few seconds will not be acceptable in most application contexts.

## Solutions to Problematic GC

Once your application's typical GC cycles are understood, consider one or more of the following solutions:

- Using [BigMemory Max](#) to eliminate the drag GC imposes on performance in large heaps.

BigMemory opens up off-heap memory for use by Java applications, and off-heap memory is not subject to GC.

- Configuring the [HealthChecker parameters](#) in the Terracotta cluster to account for the observed GC cycles.

Increase nodes' tolerance of inactivity in other nodes due to GC cycles.

- Tuning the GC parameters to change the way GC runs in the heap.

## Solutions to Problematic GC

If running multi-core machines and no collector is specifically configured, consider `-XX:+UseParallelGC` and `-XX:+UseParallelOldGC`.

If running multiple JVMs or application processes on the same machine, tune the number of concurrent threads in the parallel collector with `-XX:ParallelGCThreads=<number>`.

Another collector is called Concurrent Mark Sweep (CMS). This collector is normally not recommended (especially for Terracotta servers) due to certain performance and operational issues it raises. However, under certain circumstances related to the type of hosting platform and application data usage characteristics, it may boost performance and may be worth testing with.

- If running on a 64-bit JVM, and if your JDK supports it, use `-XX:+UseCompressedOops`.

This setting can reduce substantially the memory footprint of object pointer used by the JVM.

## Detect Memory Pressure Using the Terracotta Logs

Terracotta server and client logs contain messages that help you track memory usage. Locations of server and client logs are configured in the Terracotta configuration file, `tc-config.xml`.

You can view the state of memory usage in a node by finding messages similar to the following:

```
2011-12-04 14:47:43,341 [Statistics Logger] ... memory free : 39.992699 MB
2011-12-04 14:47:43,341 [Statistics Logger] ... memory used : 1560.007301 MB
2011-12-04 14:47:43,341 [Statistics Logger] ... memory max : 1600.000000 MB
```

These messages can indicate that the node is running low on memory and could soon experience an `OutOfMemoryError`. You could take one or more of the following actions:

- Increase the heap memory available to Terracotta.

Heap memory available to Terracotta is indicated by the message `2011-12-04 14:47:43,341 [Statistics Logger] ... memory max : 1600.000000 MB`.

- If increasing heap memory is problematic due to long GC cycles, consider the remedies suggested in [this section](#).

## Reduce Swapping

An operating system (OS) that is swapping to disk can substantially slow down or even stop your application. If the OS is under pressure because Terracotta servers—along with other processes running on a host—are squeezing the available memory, then memory will start to be paged in and out. This type of operation, when too frequent, requires either tuning of the swap parameters or a permanent solution to a chronic lack of RAM.

Many tools are available to help you diagnose swapping. Popular options include using a built-in command-line utility. On Linux, for example:

- See available RAM with `free -m` (display memory statistics in megabytes). Pay attention to swap utilization.
- `vmstat` displays swap-in ("si") and swap-out ("so") numbers. Non-zero values indicate swapping activity. Set `vmstat` to refresh on a short interval to detect trends.

## Reduce Swapping

- Process status can be used to get detailed information on all processes running on a node. For example, `ps -eo pid,ppid,rss,vsizer,pcpu,pmem,cmd -ww --sort=pmem` displays processes ordered by memory use. You can also sort by virtual memory size ("vsizer") and real memory size ("rss") to focus on both the most memory-consuming processes and their in-memory footprint.

## Keep Disks Local

To provide scalability and persistence, and (when necessary) ease pressure on memory to use it more efficiently, Terracotta servers write and read data from a disk-based database. This database should always be on local disks to avoid potential issues from delays or disconnections.

Avoid using SAN, NFS/NAS, and other networked disk stores that could cause lock timeouts and trigger a `TCDatabaseException`. If you must use a storage system that is not local, avoid using Terracotta persistent mode with the Terracotta Server Array to reduce or eliminate write to disk.

Hate to see your Terracotta servers rely on disk to ease pressure on memory? Consider adding [BigMemory Max](#).

## Do Not Interrupt!

*Ensure that your application does not interrupt clustered threads.* This is a common error that can cause the Terracotta client to shut down or go into an error state, after which it will have to be restarted.

The Terracotta client library runs with your application and is often involved in operations which your application is not necessarily aware of. These operations can get interrupted, something the Terracotta client cannot anticipate. Interrupting clustered threads, in effect, puts the client into a state which it cannot handle.

## Diagnose Client Disconnections

If clients disconnect on a regular basis, try the following to diagnose the cause:

- Analyze the Terracotta client logs for potential issues, such as long GC cycles.
- Analyze the Terracotta server logs for disconnection information and any rejections of reconnection attempts by the client.
- See the operator events panel in the Terracotta Developer Console for disconnection events, and note the reason.

If the disconnections are due to long GC cycles or inconsistent network connections in the client, consider the remedies suggested in [this section](#). If disconnections continue to happen, and you are using Ehcache, consider configuring caches with [nonstop behavior](#) and enabling [rejoin](#).

## Bring a Cluster Back Up in Order

Terracotta servers that are configured to persist data across restarts are operating in "permanent-store" mode. This type of cluster attempts to restore all server data (all "shared" data) when the servers return, and to remember previous clients. This persistence mode is configured in the Terracotta configuration file, `tc-config.xml`.

## Bring a Cluster Back Up in Order

In an orderly shutdown, any passive (backup) servers should be taken down first and brought up last. (Note that clients should be brought down before any servers are brought down.) This ensures a database that remains in sync between active and passive servers. Once all passive servers are down, active servers can be shut down. When restoring the cluster, the last server to go down should be brought up first.

If a cluster crashes, and no passive server takes over (even temporarily), then the last active server to go down should be brought up first.

## Manage Sessions in a Cluster

- Make sure the configured time zone and system time is consistent between all application servers. If they are different a session may appear expired when accessed on different nodes.
- Set `-Dcom.tc.session.debug.sessions=true` and `-Dcom.tc.session.debug.invalidate=true` to generate more debugging information in the client logs.
- All clustered session implementations (including terracotta Sessions) require a mutated session object be put back into the session after it's mutated. If the call is missing, then the change isn't known to the cluster, only to the local node. For example:

```
Session session = request.getSession();
Map m = session.getAttribute("foo");
m.clear();

// Without the following call, the clear() is not effective across the cluster.
session.setAttribute("foo", m);
```

Without a `setAttribute()` call, the session becomes inconsistent across the cluster. Sticky sessions can mask this issue, but as soon as the session is accessed on another node, its state does not match the expected one. To view the inconsistency on a single client node, add the Terracotta property `-Dcom.tc.session.clear.on.access=true` to force locally cached sessions to be cleared with every access.

If third-party code cannot be refactored to fix this problem, and you are running Terracotta 3.6.0 or higher, you can write a servlet filter that calls `setAttribute()` at the end of every request. Note that this solution may substantially degrade performance.

```
package controller.filter;

import java.io.IOException;
import java.util.Enumeration;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;

public class IterateFilter implements Filter {

    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
```

## Manage Sessions in a Cluster

```
throws IOException, ServletException {
    HttpSession session = ((HttpServletRequest) request).getSession();
    if (session != null) {
        @SuppressWarnings("rawtypes")
        Enumeration e = session.getAttributeNames();
        while (e.hasMoreElements()) {
            String name = (String)e.nextElement();
            Object value = session.getAttribute(name);
            session.setAttribute(name, value);
        }
    }
}

public void init(FilterConfig filterConfig) throws ServletException {
    // TODO Auto-generated method stub
}

public void destroy() {
    // TODO Auto-generated method stub
}
}
```

# BigMemory Max Explained

Nodes running JVMs 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 software, to use that memory effectively. This drawback has limited Terracotta servers, for example, to using a small Java object heap as an in-memory store, backed by a limitless but slower disk store.

## How BigMemory Improves Performance

The performance of Terracotta clients and server instances is affected by the amount of faulting required to make data available. In-memory data elements are fetched speedily 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 has the advantage of being limitless in size. In-memory storage is limited in size by system and hardware constraints, yet even this limit is difficult for Java heaps to reach due to the heavy cost imposed by 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.

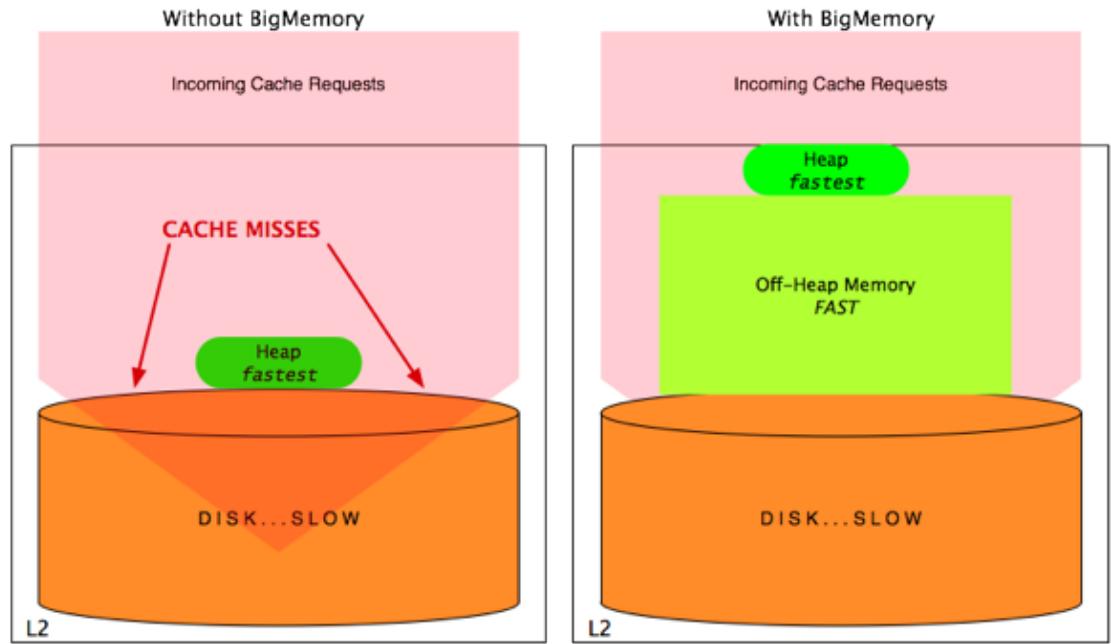
BigMemory allows Terracotta servers to expand memory storage in a way that bypasses the limitations resulting from Java GC. Using this *off-heap* memory gives the Terracotta cluster a number of important advantages:

- Larger in-memory stores without the pauses of GC.
- More locality of reference as more data is stored at the client.
- 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.

## How BigMemory Improves Performance



## Using BigMemory With Terracotta Servers

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.

BigMemory can use a 32-bit or 64-bit JVM (see [Release and Platform Compatibility Information](#) to locate supported platforms). 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.

See the [configuration page](#) for information on how to configure BigMemory on the Terracotta Server Array.

# Technical FAQ

The Technical FAQ answers questions on how to use Terracotta products, integration with other products, and solving issues. If your question doesn't appear here, consider posting it on the [Terracotta forums](#). Other resources for resolving issues include:

- [Release Notes](#) – Lists features and issues for specific versions of Terracotta products.
- [Compatibility Information](#) – Includes tables on compatible versions of Terracotta products, JVMs, and application servers.
- [General FAQ](#) – For non-technical questions.

The FAQ is divided into the following sections:

## CONFIGURATION

{toc-zone|3:3}

### How do I enable persistent mode?

In the servers section of your config.xml, add the following lines:

```
<server host="host1" name="server1">
...
<dso>
...
<persistence>
    <mode>permanent-store</mode>
</persistence>
...
</dso>
...
</server>
```

Note that the temporary-swap mode performs better than permanent-store, and should be used where persistency of shared data is not required across restarts.

### How do I configure failover to work properly with two Terracotta servers?

Configure both servers in the <servers> section of the Terracotta configuration file. Start the two Terracotta server instances that use that configuration file, one server assumes control of the cluster (the ACTIVE) and the second becomes the backup (the PASSIVE). See the [high-availability chapter](#) in the product documentation for more information.

{/toc-zone}

## DEVELOPMENT

{toc-zone|3:3}

How do I know that my application has started up with a Terracotta client and are sharing data?

## **How do I know that my application has started up with a Terracotta client and are sharing data?**

The [Terracotta Developer Console](#) displays cluster topology by listing Terracotta server groups and connected client nodes in a navigation tree. If you're using Ehcache, Ehcache with Hibernate, Quartz Scheduler, or Terracotta Web Sessions, special panels in the Developer Console become active to give greater visibility into application data. These panels display in-memory values, provide live statistics, and offer a number of controls for configuring and manipulating in-memory data.

In addition, check standard output for messages that the Terracotta client has started up without errors. Terracotta clients also log messages to a file specified in the <clients> section of the Terracotta configuration file.

## **Is there a maximum number of objects that can be held by one Terracotta server instance?**

The number of objects that can be held by a Terracotta server instance is two billion, a limit imposed by the design of Java collections. It is unlikely that a Terracotta cluster will need to approach even 50 percent of that maximum. However, if it does, other issues may arise that require the rearchitecting of how application data is handled in the cluster.

## **Why is it a bad idea to change shared data in a shutdown hook?**

If a node attempts to change shared data while exiting, and the shutdown thread blocks, the node may hang and be dropped from the cluster, failing to exit as planned. The thread may block for any number of reasons, such as the failure to obtain a lock. A better alternative is to use the cluster events API to have a second node (one that is not exiting) execute certain code when it detects that the first node is exiting. If you are using Ehcache, use the cluster-events Ehcache API. In general, you can use the Terracotta Toolkit API to set up cluster-events listeners.

If you're using DSO and the Terracotta Toolkit, you can call `org.terracotta.api.Terracotta.registerBeforeShutdownHook(Runnable beforeShutDownHook)` to perform various cleanup tasks before the Terracotta client disconnects and shuts down.

Note that a Terracotta client is not required to release locks before shutting down. The Terracotta server will reclaim those locks, although any outstanding transactions are not committed.

## **What's the best way for my application to listen to Terracotta cluster events such as lost application nodes?**

If you are using Ehcache, use the cluster-events Ehcache API. In general, you can use the Terracotta Toolkit API to set up cluster-events listeners.

## **How can my application check that the Terracotta process is alive at runtime?**

How can my application check that the Terracotta process is alive at runtime?

Your application can check to see if the system property `tc.active` is true. For example, the following line of code would return true if Terracotta is active at the time it is run:

```
Boolean.getBoolean("tc.active");
```

## What are some ways to externally monitor a cluster?

See [this question](#).

{/toc-zone}

## ENVIRONMENT

{toc-zone|3:3}

## Where is there information on platform compatibility for my version of Terracotta software?

Information on the latest releases of Terracotta products, including a link to the latest platform support, is found on the [Product Information](#). This page also contains a table with links to information on previous releases.

## Can I run the Terracotta process as a Microsoft Windows service?

Yes. See [Starting up TSA or CLC as Windows Service using the Service Wrapper](#).

## How do I use my JRE instead of the one shipped with Terracotta?

Set the `TC_JAVA_HOME` environment variable to point to a [supported JDK or JRE](#). The target should be the top level installation directory of the JDK, which is the directory containing the `bin` directory.

## Do you have any advice for running Terracotta software on Ubuntu?

The known issues when trying to run Terracotta software on Ubuntu are:

- Default shell is `dash` not `bash`. Terracotta scripts don't behave under dash. You might solve this issue by setting your default shell to bash or changing `/bin/sh` in our scripts to `/bin/bash`.
- The Ubuntu default JDK is from GNU. Terracotta software compatibility information is on the [Product Information](#) page.
- See the [UnknownHostException](#) topic.

## Which Garbage Collector should I use with the Terracotta Server (L2) process?

The Terracotta Server performs best with the default garbage collector. This is pre-configured in the startup scripts. If you believe that Java GC is causing performance degradation in the Terracotta Server, [BigMemory](#) is recommended as the simplest and best way to reduce latencies by reducing collection times.

Which Garbage Collector should I use with the Terracotta Server (L2)process?

Generally, the use of the Concurrent Mark Sweep collector (CMS) is discouraged as it is known to cause heap fragmentation for certain application-data usage patterns. Expert developers considering use of CMS should consult the Oracle tuning and best-practice documentation.

{/toc-zone}

## INTEROPERABILITY

{toc-zone|3:3}

### Can I substitute Terracotta for JMS? How do you do messaging in Terracotta clusters?

Using Terracotta with a simple data structure (such as `java.util.concurrent.LinkedBlockingQueue`), you can easily create message queues that can replace JMS. Your particular use case should dictate whether to replace JMS or continue using it alongside Terracotta. See the [Terracotta Toolkit API](#) for more information on using a clustered queue.

### Does Terracotta clustering work with Hibernate?

Through Ehcache, you can enable and cluster [Hibernate second-level cache](#).

### What other technologies does Terracotta software work with?

Terracotta software integrates with most popular Java technologies being used today. For a full list, contact us at [{\\$contact\\_email}](#).

{/toc-zone}

## OPERATIONS

{toc-zone|3:3}

### How do I confirm that my Terracotta servers are up and running correctly?

Here are some ways to confirm that your Terracotta servers are running:

- Connect to the servers using the [Terracotta Developer Console](#).
- Check the standard output messages to see that each server started without errors.
- Check each server's logs to see that each server started without errors. The location of server logs is specified in `tc-config.xml`.
- Use the Terracotta script `server-stat.sh` or `server-stat.bat` to generate a short status report on one or more Terracotta servers.
- Use a tool such as `wget` to access the `/config` or `/version` servlet. For example, for a server running on localhost and using DSO port 9510, use the following `wget` command to connect to the version servlet:  
`wget -q -O - http://localhost:9510/config`

How do I confirm that my Terracotta servers are up and running correctly?

```
[PROMPT] wget http://localhost:9510/version
```

## How can I control the logging level for Terracotta servers and clients?

Create a file called `.tc.custom.log4j.properties` and edit it as a standard `log4j.properties` file to configure logging, including level, for the Terracotta node that loads it. This file is searched for in the path specified by the environment variable `TC_INSTALL_DIR` (if defined), `user.home`, and `user.dir`.

## Are there ways I can monitor the cluster that don't involve using the Terracotta Developer Console or Operations Center?

You can monitor the cluster using JMX. A good starting point is the [Terracotta JMX guide](#). This document does not have a complete list of MBeans, but you can use a tool such as JConsole to view the MBeans needed for monitoring.

Statistics are available over JMX via the object name "org.terracotta:type=Terracotta Server,subsystem=Statistics,name=Terracotta Statistics Gatherer". The [Terracotta Cluster Statistics Recorder](#) has both command-line and RESTful interfaces. However, statistics recording can substantially degrade performance due its high resource cost.

Certain cluster parameters, such as heap size and cached-object count, are available via "org.terracotta:type=Terracotta Server,name=DSO".

Cluster events are available over JMX via the object name "org.terracotta:type=TC Operator Events,name=Terracotta Operator Events Bean".

## How many Terracotta clients (L1s) can connect to the Terracotta Server Array (L2s) in a cluster?

While the number of L1s that can exist in a Terracotta cluster is theoretically unbounded (and cannot be configured), effectively planning for resource limitations and the size of the shared data set should yield an optimum number. Typically, the most important factors that will impact that number are the requirements for performance and availability. Typical questions when sizing a cluster:

- What is the desired transactions-per-second?
- What are the failover scenarios?
- How fast and reliable is the network and hardware? How much memory and disk space will each machine have?
- How much shared data is going to be stored in the cluster? How much of that data should be on the L1s? Will BigMemory be used?
- How many stripes (active Terracotta servers) does the cluster have?
- How much load will there be on the L1s? On the L2s?

The most important method for determining the optimum size of a cluster is to test various cluster configurations under load and observe how well each setup meets overall requirements.

```
{/toc-zone}
```

# TROUBLESHOOTING

{toc-zone|3:3}

## **After my application interrupted a thread (or threw InterruptedException), why did the Terracotta client die?**

The Terracotta client library runs with your application and is often involved in operations which your application is not necessarily aware of. These operations may get interrupted, too, which is not something the Terracotta client can anticipate. Ensure that your application does not interrupt clustered threads. This is a common error that can cause the Terracotta client to shut down or go into an error state, after which it will have to be restarted.

## **Why does the cluster seem to be running more slowly?**

There can be many reasons for a cluster that was performing well to slow down over time. The most common reason for slowdowns is Java Garbage Collection (GC) cycles.

Another possible cause is when an active server is syncing with a mirror server. If the active is under substantial load, it may be slowed by syncing process. In addition, the syncing process itself may appear to slow down. This can happen when the mirror is waiting for specific sequenced data before it can proceed. This is indicated by log messages similar to the following:

```
WARN com.tc.12.ha.L2HACoordinator - 10 messages in pending queue.
Message with ID 2273677 is missing still
```

If the message ID in the log entries changes over time, no problems are indicated by these warnings.

One indication that slowdowns are occurring on the server and that clients are throttling their transaction commits is the appearance of the following entry in client logs:

```
INFO com.tc.object.tx.RemoteTransactionManagerImpl - ClientID[2]
(: TransactionID=[65037] : Took more than 1000ms to add to sequencer : 1497 ms
```

## **Why do all of my objects disappear when I restart the server?**

If you are not running the server in persistent mode, the server will remove the object data when it restarts. If you want object data to persist across server restarts, run the server in [persistent mode](#).

## **Why are old objects still there when I restart the server?**

If you are running the server in persistent mode, the server keeps the object data across restarts. If you want objects to disappear when you restart the server you can either run in non-persistent mode or remove the data files from disk before you restart the server. See [this question](#).

## **Why is the Terracotta Developer Console or Terracotta Operations Center timing out when it tries to connect to a Terracotta server?**

**Why is the Terracotta Developer Console or Terracotta OperationsCenter timing out when it tries to connect?**

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.

## **Why does the Developer Console runs very slowly when I'm monitoring the cluster?**

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.pmoffscreen` to "false" to allow Swing buffers to reside in memory:

```
-Dsun.java2d.pmoffscreen=false
```

For information about this Java system property, see  
<http://download.oracle.com/javase/1.5.0/docs/guide/2d/flags.html#pmoffscreen>.

You can add these properties to the Developer Console start-up script (`dev-console.sh` or `dev-console.bat`).

## **Why can't certain nodes on my Terracotta cluster see each other on the network?**

A firewall may be preventing different nodes on a cluster from seeing each other. If Terracotta clients attempt to connect to a Terracotta server, for example, but the server seems to not have any knowledge of these attempts, the clients may be blocked by a firewall. Another example is a backup Terracotta server that comes up as the active server because it is separated from the active server by a firewall.

## **Client and/or server nodes are exiting regularly without reason.**

Client or server processes that quit ("L1 Exiting" or "L2 Exiting" in logs) for seemingly no visible reason may have been running in a terminal session that has been terminated. The parent process must be maintained for the life of the node process, or use another workaround such as the `nohup` option.

## **Why is more than one active server coming up when I configured only one?**

If you have a setup with one active Terracotta server instance and a number of standbys, but are seeing errors because more than one active server is in the cluster, a "split brain" scenario has occurred.

**Why is more than one active server coming up when I configured only one?**

Due to network latency or load, the Terracotta server instances may not have enough time to hold an election. Increase the property in the Terracotta configuration file to the lowest value that solves this issue.

If you are running on Ubuntu, see the note at the end of the [UnknownHostException](#) topic.

## **I have a cluster with more than one stripe (more than one active Terracotta server) but data is distributed very unevenly between the two stripes.**

The Terracotta Server Array distributes data based on the hashCode of keys. To enhance performance, each server stripe should contain approximately the same amount of data. A grossly uneven distribution of data on Terracotta servers in a cluster with more than one active server can be an indication that keys are not being hashed well. If your application is creating keys of a type that does not hash well, this may be the cause of the uneven distribution.

## **Why is a crashed Terracotta server instance failing to come up when I restart it?**

If it's running in persistent mode, the ACTIVE Terracotta server instance should come up with all shared data intact. However, if the server's database has somehow become corrupt, you must clear the crashed server's data directory before restarting.

## **I lost some data after my entire cluster lost power and went down. How can I ensure that all data persists through a failure?**

If only some data was lost, then Terracotta servers were configured to persist data. The cause for losing a small amount of data could be disk "write" caching on the machines running the Terracotta server instances. If every Terracotta server instance lost power when the cluster went down, data remaining in the disk cache of each machine is lost.

Turning off disk caching is not an optimal solution because the machines running Terracotta server instances will suffer a substantial performance degradation. A better solution is to ensure that power is never interrupted at any one time to every Terracotta server instance in the cluster. This can be achieved through techniques such as using uninterruptible power supplies and geographically subdividing cluster members.

## **Why does the JVM on my SPARC machines crash regularly?**

You may be encountering a known issue with the Hotspot JVM for SPARC. The problem is expected to occur with Hotspot 1.6.0\_08 and higher, but may have been fixed in a later version. For more information, see this [bug report](#).

{/toc-zone}

# **SPECIFIC ERRORS AND WARNINGS**

{ toc-zone|3:3 }

Why, after restarting an application server, does the error Client Cannot Reconnect... repeat endlessly until

## **Why, after restarting an application server, does the error Client Cannot Reconnect... repeat endlessly until the Terracotta server's database is wiped?**

The default value of the client reconnection setting `l1.max.connect.retries` is set to "-1" (infinite). If you frequently encounter the situation described in this question and do not want to wipe the database and restart the cluster, change the retry setting to finite value. See the [high-availability page](#) for more information.

## **What does the warning "WARN com.tc.bytes.TCByteBufferFactory - Asking for a large amount of memory..." mean?**

If you see this warning repeatedly, objects larger than the recommended maximum are being shared in the Terracotta cluster. These objects must be sent between clients and servers. In this case, related warnings containing text similar to `Attempt to read a byte array of len: 12251178; threshold=8000000` and `Attempting to send a message (com.tc.net.protocol.delivery.OOOProtocolMessageImpl) of size` may also appear in the logs.

If there are a large number of over-sized objects being shared, low-memory issues, overall degradation of performance, and OutOfMemory errors may result.

## **What is causing regular segmentation faults (segfaults) with clients and/or servers failing?**

Sigsegfaults may be caused by Hyperic (Sigar), the library used by Terracotta servers and clients to report on certain system resources, mainly activity data for CPU, combined CPU, disk, and network. You may need to turn off Sigar, thus losing the ability to monitor and record these network resources directly through Terracotta software. To turn off Sigar, set the Terracotta property `sigar.enabled` to false on the nodes that exhibit the error:

```
-Dsigar.enabled=false
```

For more information on setting Terracotta properties, see the [Terracotta Configuration Guide and Reference](#).

## **When starting a Terracotta server, why does it throw a DBVersionMismatchException?**

The server is expecting a Terracotta database with a compatible version, but is finding one with non-compatible version. This usually occurs when starting a Terracotta server with an older version of the database. Note that this can only occur with servers in the permanent-store persistence mode.

## **Why am I getting MethodNotFound and ClassNotFound exceptions?**

If you've integrated a Terracotta product with a framework such as Spring or Hibernate and are getting one of these exceptions, make sure that an older version of that Terracotta product isn't on the classpath. With Maven involved, sometimes an older version of a Terracotta product is specified in a framework's POM and ends up ahead of the current version you've specified. You can use tools such as jconsole or jvisualvm to debug, or specify `-XX:+TraceClassLoading` on the command line.

I'm getting a Hyperic (Sigar) exception, and the Terracotta Developer Console or Terracotta Operations Center

## **I'm getting a *Hyperic (Sigar)* exception, and the Terracotta Developer Console or Terracotta Operations Center is not showing certain metrics?**

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.

## **When I start a Terracotta server, why does it fail with a schema error?**

You may get an error similar to the following when a Terracotta server fails to start:

```
Error Message:  
Starting BootJarTool...  
2008-10-08 10:29:29,278 INFO - Terracotta 2.7.0, as of 20081001-101049  
(Revision 10251 by cruise@rh4mo0 from 2.7)  
2008-10-08 10:29:30,459 FATAL -  
*****  
The configuration data in the file at '/opt/terracotta/conf/tc-config.xml'  
does not obey the Terracotta schema:  
[0]: Line 8, column 3: Element not allowed: server in element servers  
*****
```

This error occurs when there's a schema violation in the Terracotta configuration file, at the line indicated by the error text. To confirm that your configuration file follows the required schema, see the schema file included with the Terracotta kit. The kit includes schema files (\*.xsd) for Terracotta, Ehcache, and Quartz configurations.

## **Why is a newly started passive (backup) Terracotta server failing to join the cluster as it tries to synchronize with the active server?**

If a newly started passive (backup) Terracotta server fails with an error similar to

*com.tc.objectserver.persistence.db.DBException*:

*com.sleepycat.je.LockTimeoutException*: (JE 4.1.10) Lock expired, and then continues to fail with that error upon restart, then the synchronization phase between the passive and active Terracotta servers must be tuned. Specifically, try raising the value of the following Terracotta property:

```
<!-- Value is in microseconds. -->  
<property name="12.berkeleydb.je.lock.timeout" value="180000000" />
```

Note that you must restart the active server for this property to take effect.

## **The Terracotta servers crash regularly and I see a ChecksumException.**

If the logs reveal an error similar to *com.sleepycat.je.log.ChecksumException*: Read invalid log entry type: 0 LOG\_CHECKSUM, there is likely a corrupted disk on at least one of the servers.

Why is java.net.UnknownHostException thrown when I try to run Terracotta sample applications?

## Why is java.net.UnknownHostException thrown when I try to run Terracotta sample applications?

If an UnknownHostException occurs, and you experience trouble running the Terracotta Welcome application and the included sample applications on Linux (especially Ubuntu), you may need to edit the etc/hosts file.

The UnknownHostException may be followed by "unknown-ip-address".

For example, your etc/hosts file may contain settings similar to the following:

```
127.0.0.1      localhost  
127.0.1.1      myUbuntu.usa myUbuntu
```

If myUbuntu is the host, you must change 127.0.1.1 to the host's true IP address.

**NOTE:** You may be able to successfully start Terracotta server instances even with the "invalid" etc/hosts file, and receive no exceptions or errors, but other connectivity problems can occur. For example, when starting two Terracotta servers that should form a mirror group (one active and one standby), you may see behavior that indicates that the servers cannot communicate with each other.

## On a node with plenty of RAM and disk space, why is there a failure with errors stating that a "native thread" cannot be created?

You may be exceeding a limit at the system level. In \*NIX, run the following command to see what the limits are:

```
ulimit -a
```

For example, a limit on the number of processes that can run in the shell may be responsible for the errors.

## Why does the Terracotta server crash regularly with *java.io.IOException: File exists?*

Early versions of JDK 1.6 had a [JVM bug](#) that caused this failure. Update JDK 1.6 to avoid this issue.

```
{/toc-zone}
```

# Distributed Cache Tutorial

Ehcace version 2.6

Follow these steps to get a sample distributed Ehcache application running in a Terracotta cluster on your machine.

## 1 Download and Install the Terracotta Enterprise Suite

Use the links below to download the Terracotta Enterprise Suite and a 30-day trial license. Run the installer and tell it where to unpack the distribution. We'll refer to this location as ``. In the following instructions, where forward slashes ("/") are given in directory paths, substitute back slashes ("\") for Microsoft Windows installations.

### Download

Enterprise Ehcache (including BigMemory, Ehcache Search and the Hibernate distributed cache plugin), Web Sessions, Quartz Scheduler and the Terracotta Server Array all come bundled with the Terracotta Enterprise Suite.

**Important:** Make sure to download the trial license key in addition to the software bundle.

[Download the Terracotta Enterprise Suite >](#)

### Install

Run the installer (your installer version may vary):

```
%> java -jar terracotta-ee-3.7.x-installer.jar
```

Copy the license key to the terracotta distribution directory:

```
%> cp terracotta-license.key /
```

## 2 Start the Distributed Cache Sample

We'll use the "colorcache" sample in the ehcache samples directory of the Terracotta distribution to demo the distributed cache.

### Start the Terracotta Server

```
%> cd <terracotta>/ehcache/samples/colorcache  
%> bin/start-sample-server.sh
```

### Start the Sample

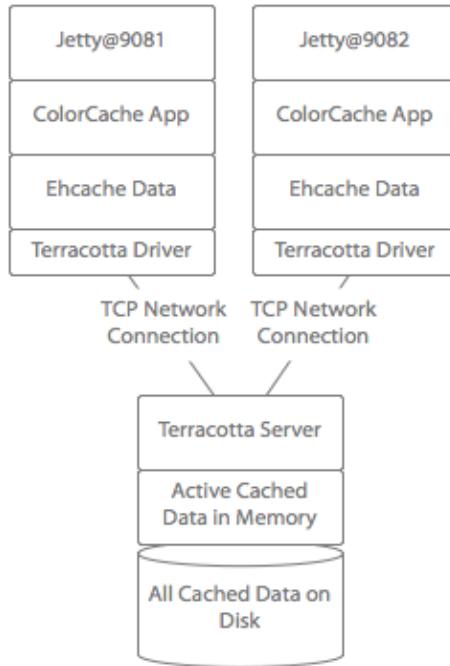
```
%> bin/start-sample.sh
```

Download

## 3 View the Sample

The distributed cache sample starts up two instances of Jetty, each connected to the Terracotta server for access to the distributed cache.

The sample application uses Ehcache to cache color data that it retrieves from a (simulated) database call. It uses the Terracotta distributed Ehcache plugin to make the cache distributed so the cached data is shared across all instances of the application:



Once the Jetty instances have started, you can view the sample application by visiting the following links:

<http://localhost:9081/colorcache> >  
<http://localhost:9082/colorcache> >

*Note: these links will not work unless you have the sample running.*

After Jetty loads the sample application, it should look something like this in your browser:

### ColorCache

Enter Color Name:

No Color Selected

Download

## 4 Load a Color

Now, load a color by entering a color name into the text field. Start by entering "red." The application will then load the color data for "red" (255, 0, 0).

### ColorCache

The screenshot shows a web page titled "ColorCache". At the top, there is an input field labeled "Enter Color Name:" containing the text "red" and a button labeled "Retrieve Color". Below the input field is a large red square. To the right of the square, the text "Color 'red' [255,0,0] retrieved in 3673 milliseconds." is displayed, with "3673 milliseconds" circled in blue. Below this text is the message "Not cached = slow".

When a color is first loaded, the application simulates the execution of a slow operation such as loading data from a database or executing a computationally intensive operation. It will then place the color object into Ehcache. Subsequent calls to load that color will be read from the cache, thereby executing much more quickly.

Try clicking the "Retrieve Color" button again to see the application load from cache.

### ColorCache

The screenshot shows a web page titled "ColorCache". At the top, there is an input field labeled "Enter Color Name:" containing the text "red" and a button labeled "Retrieve Color". Below the input field is a large red square. To the right of the square, the text "Color 'red' [255,0,0] retrieved in 6 milliseconds." is displayed, with "6 milliseconds" circled in blue. Below this text is the message "Cached = fast".

## 5 View the Distributed Cached Data in Other JVMs

To see the distributed cache work in the other application server, click on the link in the hint box.

The screenshot shows a yellow rectangular window with the following information:

|                                    |                                                                 |
|------------------------------------|-----------------------------------------------------------------|
| Current server: 9081               | ColorCache demonstrates sharing a distributed cache between     |
| Cache size: 1                      | name of a color e.g. red to request its RGB code from the back  |
| Time to live: 0                    | requesting the same color from the other node. You will see tha |
| Time to idle: 120                  | the cache. The cache is also configured to evict unused color e |
| Go to: <a href="#">Server 9082</a> | With the Terracotta Developer Console, you can monitor the ca   |

Below the window, a blue arrow points upwards, and the text "Click here to see the distributed cache at work in the other app server." is displayed.

When the page on the other application server loads, the color data for "red" loads from cache, even though the cache was loaded on the first application server:

Download

## ColorCache

Enter Color Name:



Color 'red' [255,0,0] retrieved in 646 milliseconds.

Cached = fast,  
even on the other  
app server

Try loading a few more colors into the cache. You can see the cached colors in the box on the right. Clicking on a color swatch in that box will load the color data from the cache into the main display.



Click on these swatches  
to load them into the main  
display.

You'll see those colors loaded quickly as they are loaded from the cache:

## ColorCache

Enter Color Name:



Color 'green' [0,255,0] retrieved in 7 milliseconds.

Cached = fast

Click on the link in the hint box to switch back and forth between the two application servers:

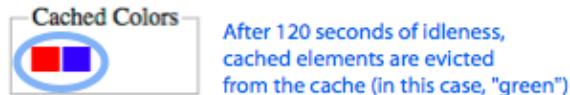
Current server: 9082      ColorCache demonstrates sharing a distributed cache between  
Cache size: 3      name of a color e.g. red to request its RGB code from the back  
Time to live: 0      requesting the same color from the other node. You will see that  
Time to idle: 120      the cache. The cache is also configured to evict unused color entries.  
Go to: [Server 9081](#)

Click here to see the  
distributed cache at work  
in the other app server.

You'll notice that the state of the cache is always in sync in both application servers, no matter which one loads the cache.

## Download

Also, notice that, if you don't load a cached color within 120 seconds of placing it in the cache, it will expire from the cache.



The cache eviction policy is configurable on a per-cache basis and evicted elements are automatically evicted across the entire cluster coherently so you will never get cache drift.

## 6 View Runtime Cache Statistics and Manage Configuration

Terracotta comes with a console that lets you inspect and dynamically change the runtime contents, statistics, and configuration of the distributed cache. To see it in action, start the console:

```
%> <terracotta>/bin/dev-console.sh
```

When the console starts, you will see a splash screen prompting you to connect to the Terracotta server. The Terracotta server coordinates all of the statistics gathering and configuration management of the cache cluster.

Click the "Connect..." button to connect the console to the cluster.



Once connected, open the Ehcache statistics and configuration panel by clicking on Ehcache in the tree-control menu:

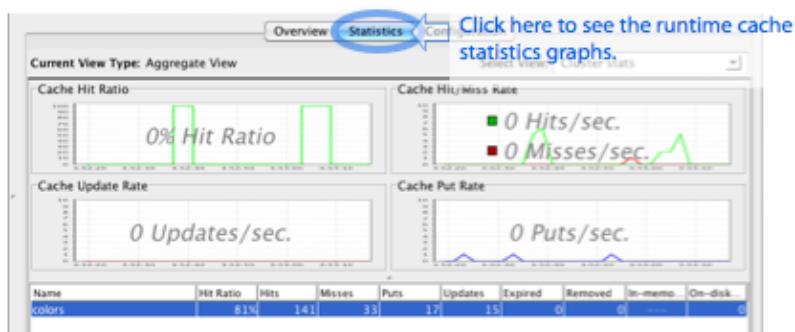
The screenshot shows the Ehcache statistics and configuration panel. On the left, a tree-control menu is expanded to show "Terracotta cluster", "My application", and "Ehcache" (which is circled in blue). Other items include "Hibernate", "Clustered heap", "Object browser (2 roots)", "Instance counts", "Garbage collection", "Diagnostics", "Runtime statistics", and "Lock profiler". The main panel shows "Cache Manager: ColorCache" with tabs for "Overview" and "Statistic". It displays a message: "Clustering 1 of 1 enabled caches from a total of 1. Statistics gathering is On." Below are buttons for "Enable All Caches", "Disable All Caches", and "Clear All Cache". At the bottom, there are sections for "Current Value" (with a histogram showing green for Hits and red for Misses) and "Global Cache Performance".

## Download

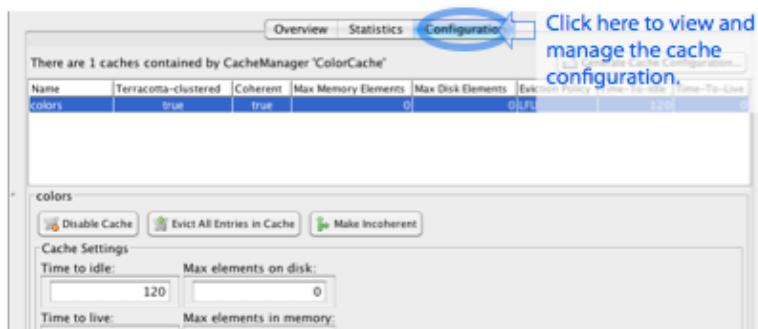
Click reload on your browser a few times and you will see the meter in the console register a green bar indicating cache hits. The red bar indicates cache misses and the blue bar indicates cache puts:



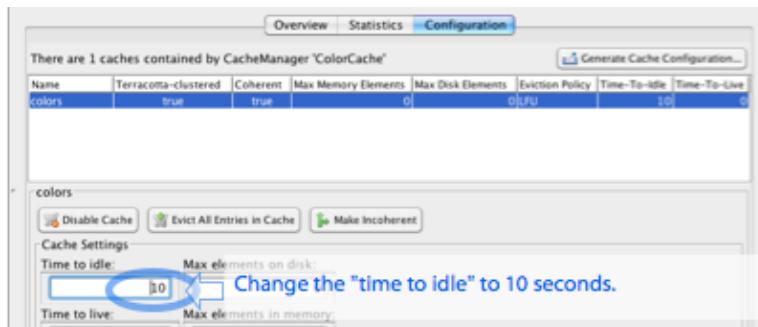
Click on the "Statistics" tab in the console to see the runtime statistics graphs:



Click on the "Configuration" tab in the console to view and manage the cache configuration:



Try changing the "time to idle" parameter on the cache to 10 seconds to see the configuration change in real time:



## Download

You can see the effect of the configuration change by loading colors into the cache. If you view them again before the new 10 second idle threshold, they will be loaded from cache. After 10 seconds of idleness, they will be evicted from the cache. If you view the other application server, you'll see the cache entries have been evicted uniformly from all instances of the cache.

# Hibernate Distributed Cache Tutorial

Follow these steps to get a sample Hibernate distributed cache application running in a Terracotta cluster on your machine.

## 1 Download and Install the Terracotta Enterprise Suite

Use the links below to download the Terracotta Enterprise Suite and a 30-day trial license. Run the installer and tell it where to unpack the distribution. We'll refer to this location as ``. In the following instructions, where forward slashes ("/") are given in directory paths, substitute back slashes ("\") for Microsoft Windows installations.

### Download

Enterprise Ehcache (including BigMemory, Ehcache Search and the Hibernate distributed cache plugin), Web Sessions, Quartz Scheduler and the Terracotta Server Array all come bundled with the Terracotta Enterprise Suite.

**Important:** Make sure to download the trial license key in addition to the software bundle.

[Download the Terracotta Enterprise Suite >](#)

### Install

Run the installer (your installer version may vary):

```
%> java -jar terracotta-ee-3.7.x-installer.jar
```

Copy the license key to the terracotta distribution directory:

```
%> cp terracotta-license.key /
```

## 2 Start the Hibernate Distributed Cache Sample

We'll use the "hibernate" sample in the ehcache directory of the Terracotta distribution to demo the distributed cache for Hibernate.

### Start the Terracotta Server

```
%> cd <terracotta>/ehcache/samples/events  
%> bin/start-sample-server.sh
```

### Start the Sample Database

```
%> bin/start-db.sh
```

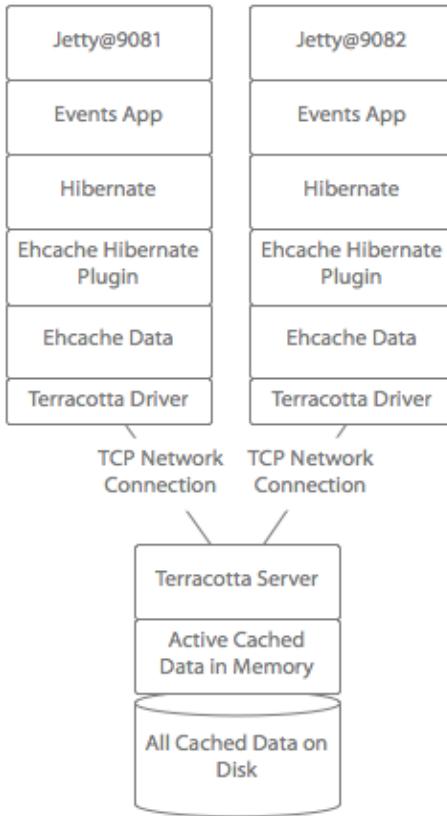
### Start the Sample

```
%> bin/start-sample.sh
```

## 3 View the Sample

The hibernate distributed cache sample starts up two instances of Jetty, each connected to the Terracotta server for access to the distributed cache.

The sample application uses Hibernate to load event data from the database. It uses the Ehcache Hibernate plugin to cache that event data. The Ehcache Hibernate plugin uses Terracotta to make the cache distributed so the cached data is shared across all instances of the application:



Once the Jetty instances have started, you can view the sample application by visiting the following links:

[>](http://localhost:9081/events)  
[>](http://localhost:9082/events)

*Note: these links will not work unless you have the sample running.*

After Jetty loads the sample application, it should look something like this in your browser:

Download

### Add new event:

Title:   
Date (e.g. 24.12.2009):

## 4 Create an Event

Enter data for a new event and click on the "store" button:

### Add new event:

Title: Rock Out  
Date (e.g. 24.12.2009): 12.12.2010  
  Click here to add the event to the database (and to the distributed cache)

 Add event data to these fields

The data will be added to the database and loaded into the cache.

*Added event.*

### Add new event:

Title:   
Date (e.g. 24.12.2009):

### Events in database:

| Event title | Event date |
|-------------|------------|
| Rock Out    | 12.12.2010 |

 The event is created in the database and displayed here.

## 5 View the Data in Other JVMs

To see the event data in the other application server, click on the link in the hint box.

Current server: 9081      The Events sample demonstrates a standard Hibernate demo c  
Go to: [Server 9082](#)      With the Terracotta Developer Console, you can monitor the ca  
Hibernate tab under *My application*.

 Click here to see the application in the other app server

## Download

When the page on the other application server loads, the event data loads from cache, even though the cache was loaded on the first application server.

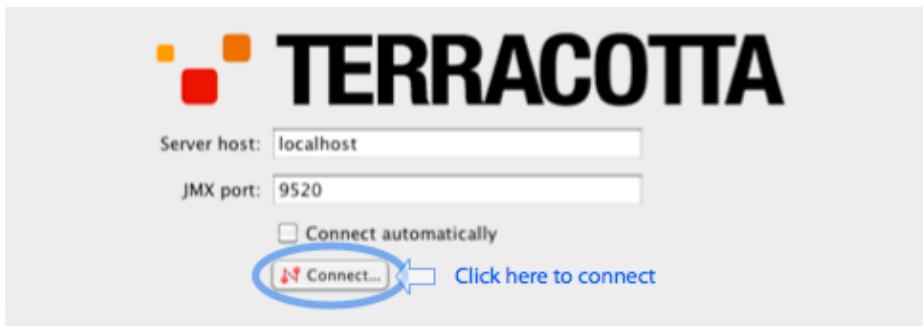
## 6 View Cache Effectiveness and Manage Configuration

Terracotta comes with a console that lets you inspect and dynamically change the runtime contents, statistics, and configuration of the distributed cache. To see it in action, start the console:

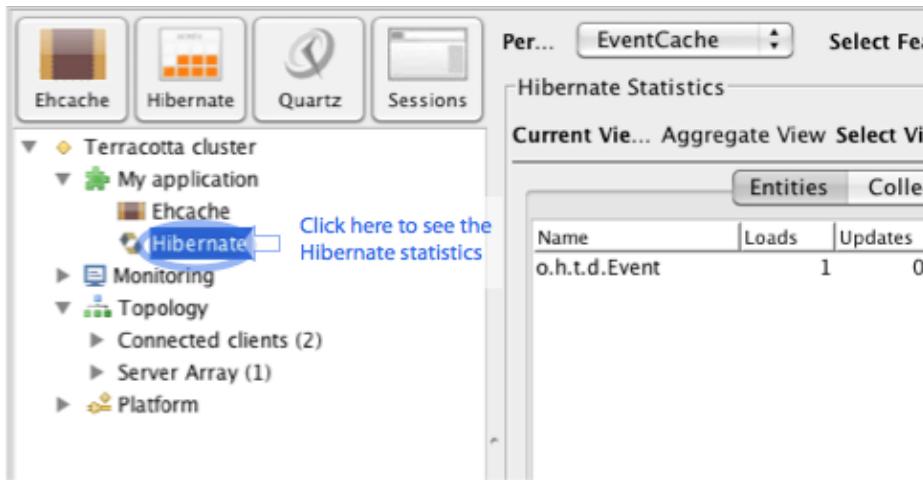
```
%> <terracotta>/bin/dev-console.sh
```

When the console starts, you will see a splash screen prompting you to connect to the Terracotta server. The Terracotta server coordinates all of the statistics gathering and configuration management of the cache cluster.

Click the "Connect..." button to connect the console to the cluster.



Once connected, open the Hibernate cache statistics and configuration panel by clicking on Hibernate in the tree-control menu:



| Name          | Loads | Updates |
|---------------|-------|---------|
| o.h.t.d.Event | 1     | 0       |

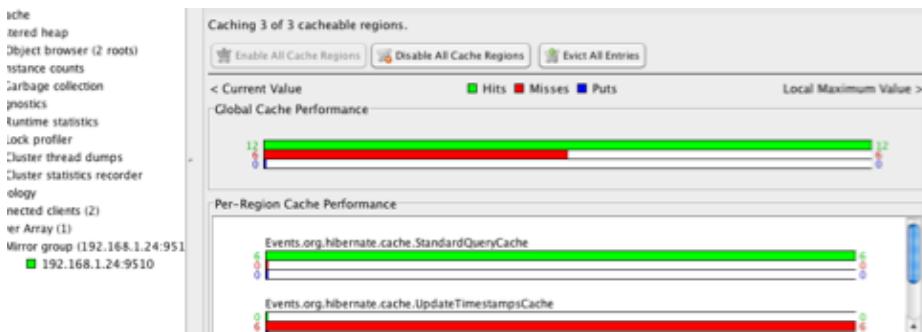
The "Entities," "Collections," and "Queries" panels in the Hibernate pane display basic Hibernate statistics for the whole cluster as well as discrete statistics for each individual application server.

To see the clustered cache statistics for Hibernate, click on the "Second-Level Cache" button:

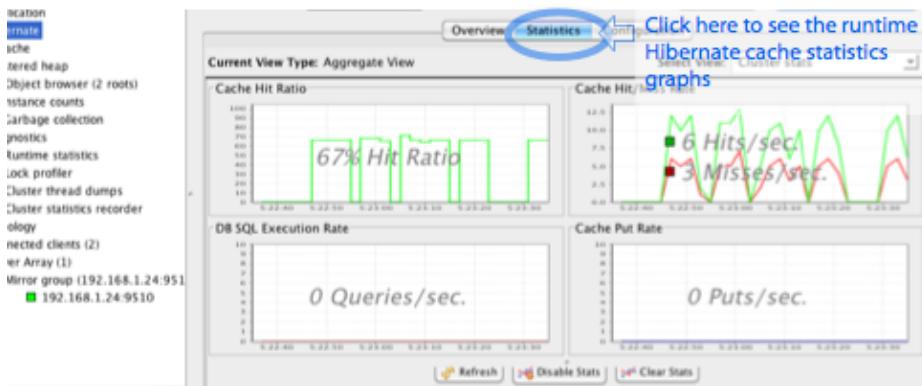
## Download

The screenshot shows the EventCache interface with the "Second-Level Cache" tab selected. At the top, there are tabs for Overview, Statistics, and Configuration. Below the tabs, there are buttons for "Enable All Cache Regions", "Disable All Cache Regions", and "Evict All Entries". A legend indicates "Hits" (green), "Misses" (red), and "Puts" (blue). A progress bar at the bottom shows values for Hits (0), Misses (0), and Puts (0).

Click reload on your browser a few times and you will see the meter in the console register a green bar indicating cache hits. The red bar indicates cache misses and the blue bar indicates cache puts:



Click on the "Statistics" tab in the console to see the runtime statistics graphs:



Click on the "Configuration" tab in the console to view and manage the cache configuration:

## Download

The screenshot shows the Persistence Unit configuration interface for an EventCache. The 'Select Feature' dropdown is set to 'Hibernate'. The 'Configuration' tab is selected. A message at the top right says 'Click here to view and manage the cache configuration...' with a link. Below it, a table lists two cacheable regions:

| Region                                       | Cached | TTI | TTL | Target Max Total Count |
|----------------------------------------------|--------|-----|-----|------------------------|
| Events.org.hibernate.cache.StandardQueryC... | On     | 120 | 120 | 10                     |
| Events.org.hibernate.tutorial.domain.Event   | On     | 120 | 120 | 10                     |

Try changing the "time to idle" parameter on the cache to 10 seconds to see the configuration change in real time:

The screenshot shows the Persistence Unit configuration interface for an EventCache. The 'Select Feature' dropdown is set to 'Hibernate'. The 'Configuration' tab is selected. A message at the top right says 'Generate Cache Configuration...' with a link. Below it, a table lists two cacheable regions:

| Region                                       | Cached | TTI | TTL | Target Max Total Count |
|----------------------------------------------|--------|-----|-----|------------------------|
| Events.org.hibernate.cache.StandardQueryC... | On     | 120 | 120 | 10                     |
| Events.org.hibernate.tutorial.domain.Event   | On     | 10  | 120 | 10                     |

A callout points to the 'Time to idle' field in the 'Events.org.hibernate.tutorial.domain.Event' region settings. The value '10' is highlighted with a blue circle.

You can see the effect of the configuration change by looking at the cache hit rate in the console. If you load an event into cache and then read it from the cache before the 10 second idle threshold, it will be loaded from cache. After 10 seconds of idleness, the cached event will be evicted from the cache. If you view the other application server, you'll see the cache entries have been evicted uniformly from all instances of the cache.

# Enterprise Ehcache Installation

## Introduction

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

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:

## Step 1: Requirements

- JDK 1.6 or higher.
- [Terracotta 3.7.4](#) 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.

## 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 (2.4.3 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`.

- \${TERRACOTTA\_HOME}/common/terracotta-toolkit-<API>-

`runtime-ee-<version>.jar`

## Step 2: Install the Distributed Cache

The Terracotta Toolkit JAR contains the Terracotta client libraries. <API> 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`:  
`-Dcom.sun.enterprise.server.ss.ASQuickStartup=false`
- WebLogic application server – You must use the [supported version](#) of WebLogic.

## 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. For more information on configuring <terracottaConfig>, see the [configuration reference](#).

### TIP: Terracotta Clients and Servers

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

## 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"
```

## Add Terracotta to Specific Caches

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

## 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`.

## Step 4: Start the Cluster

1. Start the Terracotta server:

### UNIX/Linux

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

### Microsoft Windows

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

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

### UNIX/Linux

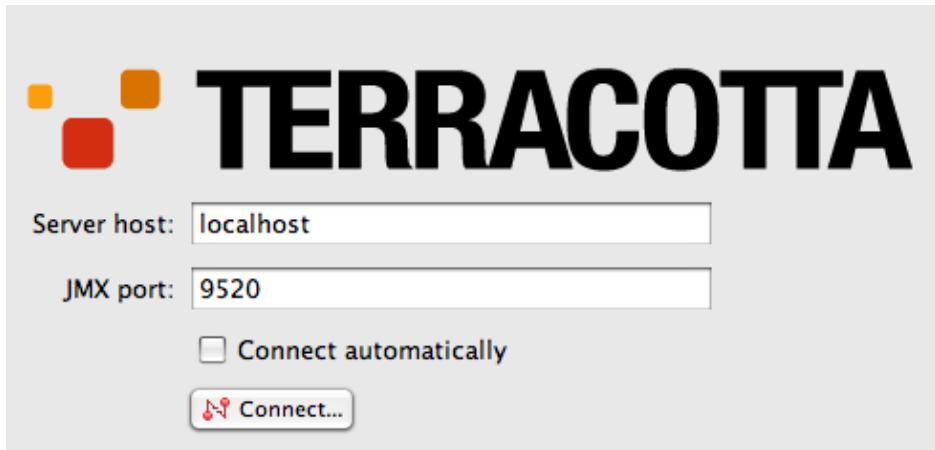
```
 ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

### Microsoft Windows

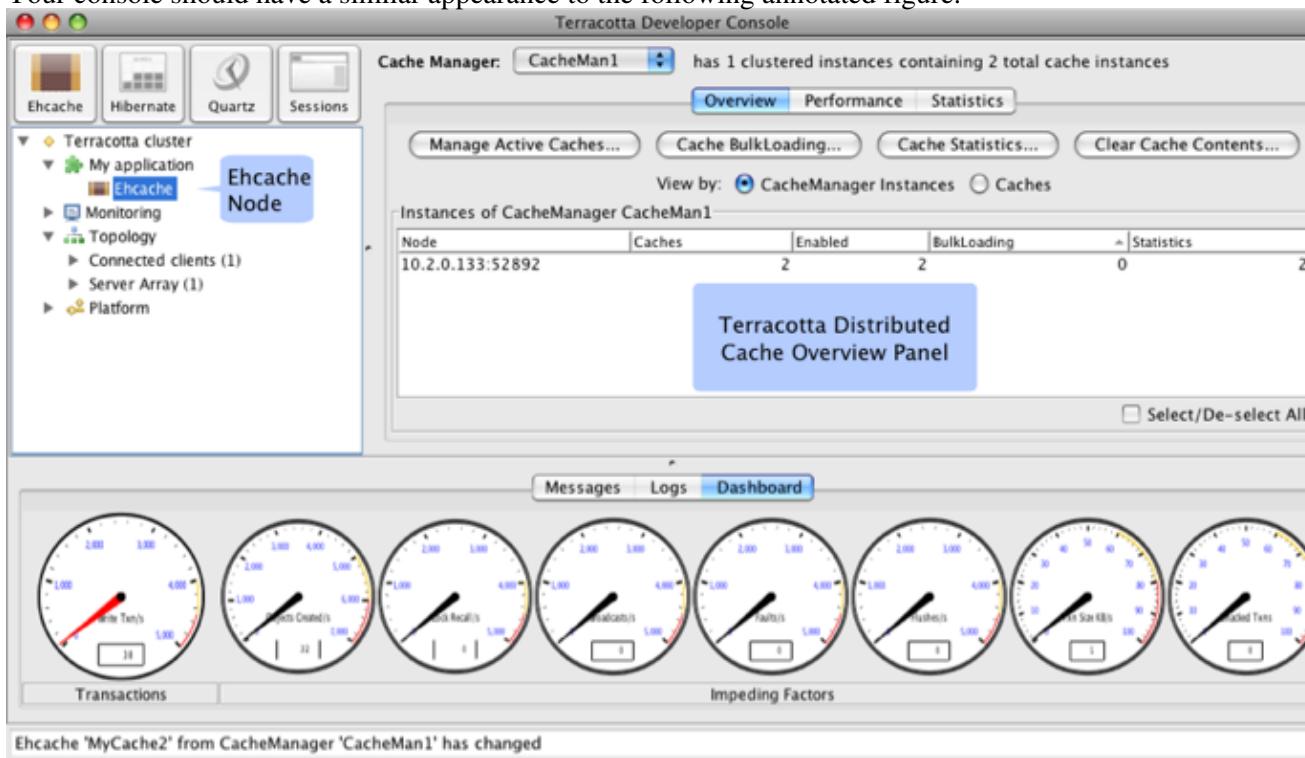
```
 ${TERRACOTTA_HOME}\bin\dev-console.bat
```

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

## Step 4: Start the Cluster



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.



## 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.

Procedure:

## 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 mirror 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 3.7.4 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`:
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

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

## Step 6: Learn More

### Microsoft Windows

```
 ${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.

## 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:

- [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 Developer Console](#) – Provides visibility into and control of caches.

# Fast Restartability

## Introduction

This page covers cache persistence, fast restartability, and using the local disk as a cache storage tier. While Ehcache offers various disk usage choices (summarized [here](#)), as of version 2.6, the recommended option for persistence is the Fast Restart store.

Fast Restart provides enterprise-ready crash resilience with an option to store a fully consistent copy of the cache on the local disk at all times. The persistent storage of the cache on disk means that after any kind of shutdown — planned or unplanned — the next time that the application starts up, all of the previously cached data is still available and very quickly accessible. The advantages of the Fast Restart store include:

1. A persistent store of the cache on disk survives crashes, providing the fastest restart. Because cached data does not need to be reloaded from the data source after a crash, but is instead loaded from the local disk, applications can resume at full speed after restart. Recovery of even terabytes of data after a failure will be very fast, minimizing downtime.
2. A persistent store on disk always contains a real-time copy of the cache, providing true fault tolerance. Even with BigMemory, where terabytes of data can be held in memory, the synchronous backup of data to disk provides the equivalent of a hot mirror right at the application and server nodes.
3. A consistent copy of the cache on local disk provides many possibilities for business requirements, such as working with different datasets according to time-based needs or moving datasets around to different locations. It can range from a simple key-value persistence mechanism with fast read performance, to an operational store with in-memory speeds during operation for both reads and writes.

## Cache Persistence Implementation

Ehcache has a RestartStore which provides fast restartability and options for cache persistence. The RestartStore implements an on-disk mirror of the in-memory cache. After any restart, data that was last in the cache will automatically load from disk into the RestartStore, and from there the data will be available to the cache.

Cache persistence on disk is configured by adding the `<persistence>` sub-element to a cache configuration. The `<persistence>` sub-element includes two attributes: `strategy` and `synchronousWrites`.

```
<cache>
  <persistence strategy="localRestartable|localTempSwap|none|distributed"
    synchronousWrites="false|true"/>
</cache>
```

## Strategy Options

The options for the `strategy` attribute are:

- **"localRestartable"** — Enables the RestartStore and copies all cache entries (on-heap and/or off-heap) to disk. This option provides fast restartability with fault tolerant cache persistence on disk.

## Strategy Options

This option is available for standalone Enterprise Ehcache only.

- "**distributed**" — Defers to the <terracotta> configuration for persistence settings. This option is for clustered Ehcache. For more information, refer to [Terracotta Clustering Configuration Elements](#).
- "**localTempSwap**" — Enables temporary local disk usage. This option provides an extra tier for storage during cache operation, but this disk storage is not persisted. After a restart, the disk tier is cleared of any cache data.
- "**none**" — Does not offload cache entries to disk. With this option, all of the cache is kept in memory. This is the default mode.

## Synchronous Writes Options

If the `strategy` attribute is set to "localRestartable", then the `synchronousWrites` attribute may be configured. The options for `synchronousWrites` are:

- **synchronousWrites="false"** — This option provides an eventually consistent copy of the cache on disk at all times. Writes to disk happen when efficient, and cache operations proceed without waiting for acknowledgement of writing to disk. After a restart, the cache is recovered as it was when last synced. This option is faster than `synchronousWrites="true"`, but after a crash, the last 2-3 seconds of written data may be lost.

If not specified, the default for `synchronousWrites` is "false".

- **synchronousWrites="true"** — This option provides a fully consistent copy of the cache on disk at all times. As elements are put into the cache, they are synchronously written to disk. The write to disk happens before a put returns to the caller. After a restart, the cache is recovered exactly as it was before shutdown. This option is slower than `synchronousWrites="false"`, but after a crash, it provides full cache consistency.

For transaction caching with `synchronousWrites`, soft locks are used to protect access. If there is a crash in the middle of a transaction, then upon recovery the soft locks are cleared on next access.

**Note:** The `synchronousWrites` attribute is also available in the <terracotta> sub-element. If configured in both places, it must have the same value.

## DiskStore Path

The path to the directory where any required disk files will be created is configured with the <diskStore> sub-element of the Ehcache configuration.

- For "localRestartable", a unique and explicitly specified path is required.
- For "localTempSwap", if the DiskStore path is not specified, a default path is used for the disk storage tier, and the default path will be auto-resolved in the case of a conflict with another CacheManager.

## Configuration Examples

This section presents possible disk usage configurations for standalone Ehcache 2.6.

Options for Crash Resilience

## Options for Crash Resilience

**Note:** Crash resilience options are for Enterprise Ehcache only.

The following configuration provides fast restartability with fully consistent cache persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="true"/>
  </cache>
</ehcache>
```

The following configuration provides fast restartability with eventually consistent cache persistence:

```
<ehcache>
  <diskStore path="/path/to/store/data"/>
  <cache>
    <persistence strategy="localRestartable" synchronousWrites="false"/>
  </cache>
</ehcache>
```

## Clustered Caches

If a cache is clustered, then persistence strategy should be set to "distributed". Persistence settings for clustered Ehcache are configured in the `<terracotta>` sub-element.

```
<cache>
  <persistence strategy="distributed"/>
  <terracotta clustered="true" consistency="eventual" synchronousWrites="false"/>
</cache>
```

## Temporary Disk Storage

The "localTempSwap" persistence strategy allows the cache to use the local disk during cache operation. The disk storage is temporary and is cleared after a restart.

```
<ehcache>
  <diskStore path="/auto/default/path"/>
  <cache>
    <persistence strategy="localTempSwap"/>
  </cache>
</ehcache>
```

**Note:** With the "localTempSwap" strategy, you can use `maxEntriesLocalDisk` or `maxBytesLocalDisk` at either the Cache or CacheManager level to control the size of the disk tier.

## In-memory Only Cache

When the persistence strategy is "none", all cache stays in memory (with no overflow to disk nor persistence on disk).

```
<cache>
  <persistence strategy="none"/>
```

## In-memory Only Cache

```
</cache>
```

## Programmatic Configuration Example

The following is an example of how to programmatically configure cache persistence on disk:

```
Configuration cacheManagerConfig = new Configuration()
    .diskStore(new DiskStoreConfiguration()
        .path("/path/to/store/data"));
CacheConfiguration cacheConfig = new CacheConfiguration()
    .name("my-cache")
    .maxBytesLocalHeap(16, MemoryUnit.MEGABYTES)
    .maxBytesLocalOffHeap(256, MemoryUnit.MEGABYTES)
    .persistence(new PersistenceConfiguration().strategy(Strategy.LOCALRESTARTABLE));
cacheManagerConfig.addCache(cacheConfig);

CacheManager cacheManager = new CacheManager(cacheManagerConfig);
Ehcache myCache = cacheManager.getEhcache("my-cache");
```

## Fast Restart Performance

When configured for fast restartability ("localRestartable" persistence strategy), the cache becomes active on restart after all the data is loaded. The amount of time until a cache is restarted is proportionate to the amount of data in the cache and the speed of the underlying infrastructure. Generally, recovery can occur as fast as the disk speed. With an SSD, for example, if you have a read throughput of 1 GB per second, you will see a similar loading speed during cache recovery.

## Fast Restart Limitations

The following recommendations should be observed when configuring caches for fast restartability:

1. The size of on-heap or off-heap caches should not be changed during a shutdown. If the amount of memory allocated to a cache is reduced, elements will be evicted upon restart.
2. Restartable caches should not be removed from the CacheManager during a shutdown.
3. If a restartable cache is disposed, the reference to the cache is deleted, but the cache contents remain in memory and on disk. After a restart, the cache contents are once again recovered into memory and on disk. The way to safely dispose of an unused restartable cache is to call clear on the cache and then dispose, so it doesn't take any space in disk or memory.

## Compatibility with Previous Versions

### Comparison of Disk Usage Options

The following table summarizes the configuration options for disk usage in Ehcache 2.6 and previous versions.

| Disk Usage                                  | Ehcache 2.6                                                                         | Ehcache 2.5 and Earlier |
|---------------------------------------------|-------------------------------------------------------------------------------------|-------------------------|
| Fast Restartability with Strong Consistency | persistence<br>strategy="localRestartable"<br>synchronousWrites="true" (Enterprise) | Not available           |

## Comparison of Disk Usage Options

|                                               |                                                 |                                                                |
|-----------------------------------------------|-------------------------------------------------|----------------------------------------------------------------|
| Fast Restartability with Eventual Consistency | only)                                           |                                                                |
| Persistence for Clustered Caches              | <pre>persistence strategy="distributed"</pre>   | Remove or edit out any disk persistence configuration elements |
| Non-Fault-Tolerant Persistence                | Use one of the fault-tolerant options above*    | <pre>overflowToDisk="true" diskPersistent="true"**</pre>       |
| Temporary Storage Tier                        | <pre>persistence strategy="localTempSwap"</pre> | <pre>overflowToDisk="true" diskPersistent="false"</pre>        |
| In-memory Only (no disk usage)                | <pre>persistence strategy="none"</pre>          | <pre>overflowToDisk="false"</pre>                              |

\*It is recommended to use one of the fault-tolerant options, however non-fault-tolerant persistence is still available. If `<persistence>` has not been specified, you can still use `overflowToDisk="true"` `diskPersistent="true"`.

\*\*In Ehcache 2.5 and earlier, cache persistence on disk for standalone Ehcache is configured with the `overflowToDisk` and `diskPersistent` attributes. If both are set to "true", cached data is saved to disk asynchronously and can be recovered after a clean shutdown or planned flush. To prevent corrupt or inconsistent data from being returned, checking measures are performed upon a restart, and if any discrepancy is found, the cache that was stored on disk is emptied and must be reloaded from the data source.

## Upgrading to Ehcache 2.6

After upgrading to Ehcache 2.6, it is strongly recommended to add the `<persistence>` sub-element to your cache configuration, and to delete, disable, or edit out disk persistence configuration elements from previous versions. The previous elements include:

- `overflowToDisk`
- `diskPersistence`
- `DiskStoreBootstrapCacheLoaderFactory`

**Note:** If any of the elements above are specified in the same configuration with either the `<persistence>` sub-element or the `<terracotta>` sub-element, it will cause an Invalid Configuration Exception.

After upgrading, however, it is not mandatory to add the `<persistence>` sub-element. In Ehcache 2.6, disk persistence configuration elements from previous Ehcache versions will continue to be available with the same functionality, as long as the `<persistence>` sub-element has not been specified.

## Upgrading for Open Source Ehcache

All persistence strategy options are available in Open Source Ehcache 2.6, except "localRestartable". In addition, all disk persistence configuration elements from previous Ehcache versions continue to be available with the same functionality (as long as the `<persistence>` sub-element has not been added).

For cache persistence on disk, you should continue to use the `overflowToDisk` and `diskPersistent` attributes. For more information, refer to [Persistence](#) in the Ehcache 2.5 documentation.

# Enterprise Ehcache API Guide

Enterprise Ehcache includes APIs for extending your application's capabilities.

## 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">
  <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>
  <terracotta />
</cache>
```

To learn more about the Ehcache Search API, refer to the following:

- The Ehcache Search API page at <http://www.ehcache.org/documentation/apis/search>.
- The `net.sf.ehcache.search*` packages in this [Javadoc](#).

## 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)));
```

## Sample Code

```
// 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.
...
```

## Dynamic Search

Dynamic Search provides flexibility by allowing the search configuration to be changed after a cache is initialized. This is done with one method call, at the point of element insertion into the cache. The `DynamicAttributesExtractor` method returns a map of attribute names to index and their respective

## Dynamic Search

values. This method is called for every Ehcache.put() and replace() invocation.

If you think that you will want to add search attributes after a cache is initialized, you can explicitly indicate the dynamic search configuration. Set the allowDynamicIndexing attribute to "true" to enable use of the dynamic attributes extractor:

```
<cache name="cacheName" ...>
    <searchable allowDynamicIndexing="true">
        ...
    </searchable>
</cache>
```

Assuming that we have previously created Person objects containing attributes such as name, age, and gender, the following example shows how to create a dynamically searchable cache and register the DynamicAttributesExtractor:

```
Configuration config = new Configuration();
config.setName("default");
CacheConfiguration cacheCfg = new CacheConfiguration("PersonCache");
cacheCfg.setEternal(true);
cacheCfg.terracotta(new TerracottaConfiguration().clustered(true));
Searchable searchable = new Searchable().allowDynamicIndexing(true);

cacheCfg.addSearchable(searchable);
config.addCache(cacheCfg);

CacheManager cm = new CacheManager(config);
Ehcache cache = cm.getCache("PersonCache");
final String attrNames[] = {"first_name", "age"};
// Now you can register a dynamic attribute extractor that would index the cache elements, using
cache.registerDynamicAttributesExtractor(new DynamicAttributesExtractor() {
    Map<String, Object> attributesFor(Element element) {
        Map<String, Object> attrs = new HashMap<String, Object>();
        Person value = (Person)element.getObjectValue();
        // For example, extract first name only
        String fName = value.getName() == null ? null : value.getName().split("\s+")[0];
        attrs.put(attrNames[0], fName);
        attrs.put(attrNames[1], value.getAge());
        return attrs;
    }
});
// Now add some data to the cache
cache.put(new Element(10, new Person("John Doe", 34, Person.Gender.MALE)));
```

Given the code above, the newly put element would be indexed on values of name and age fields, but not gender. If, at a later time, you would like to start indexing the element data on gender, you would need to create a new DynamicAttributesExtractor instance that extracts that field for indexing.

## Dynamic Search Rules

- A dynamically searchable cache must have a dynamic extractor registered before data is added to it. (This is to prevent potential races between extractor registration and cache loading which might result in an incomplete set of indexed data, leading to erroneous search results.)
- Each call on the DynamicAttributesExtractor method replaces the previously registered extractor, as there can be at most one extractor instance configured for each such cache.
- If a dynamically searchable cache is initially configured with a predefined set of search attributes, then this set of attributes will always be queried for extracted values, regardless of whether or not

## Stored Search Indexes

- there is a dynamic search attribute extractor configured.
- The initial search configuration takes precedence over dynamic attributes, so if the dynamic attribute extractor returns an attribute name already used in the initial searchable configuration, an exception will be thrown.
- Clustered Ehcache clients do not share dynamic extractor instances or implementations. In a clustered searchable deployment, the initially configured attribute extractors cannot vary from one client to another, and this is enforced by propagating them across the cluster. However, for dynamic attribute extractors, each clustered client maintains its own dynamic extractor instance, not shared with the others. Each distributed application using dynamic search must therefore maintain its own attribute extraction consistency.

**Note:** Dynamic search is available for offheap-backed caches only.

## 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.

## Best Practices for Optimizing Searches

- Construct searches wisely by including only the data that is actually required.
  - Only use `includeKeys()` and/or `includeAttribute()` if those values are actually required for your application logic.
  - If you don't need values or attributes, be careful not to burden your queries with unnecessary work. For example, if `result.getValue()` is not called in the search results, then don't use `includeValues()` in the original query.
  - Consider if it would be sufficient to get attributes or keys on demand. For example, instead of running a search query with `includeValues()` and then `result.getValue()`, run the query for keys and include `cache.get()` for each individual key.

**Note:** `includeKeys()` and `includeValues()` have lazy deserialization, which means that keys and values are de-serialized only when `result.getKey()` or `result.getValue()` is called. However, there is still some time cost with `includeKeys()` and `includeValues()`, so consider carefully when constructing your queries.

- Searchable keys and values are automatically indexed by default. If you do not need to search against keys or values, turn off automatic indexing with the following:

```
<cache name="cacheName" ...>
  <searchable keys="false" values="false"/>
  ...
</searchable>
```

## Best Practices for Optimizing Searches

- ```
</cache>
```
3. Limit the size of the results set with `query.setMaxResults(int number_of_results)`. Another recommendation for managing the size of the result set is to use a built-in Aggregator function to return a summary statistic (see the `net.sf.ehcache.search.aggregator` package in this [Javadoc](#)).
  4. Make your search as specific as possible. Queries with "ILike" criteria and fuzzy (wildcard) searches may take longer than more specific queries. Also, if you are using a wildcard, try making it the trailing part of the string instead of the leading part ("321\*" instead of "\*123"). If you want leading wildcard searches, then you should create a `<searchAttribute>` with the string value reversed in it, so that your query can use the trailing wildcard instead.
  5. When possible, use the query criteria "Between" instead of "LessThan" and "GreaterThan", or "LessThanOrEqual" and "GreaterThanOrEqual". For example, instead of using `le(startDate)` and `ge(endDate)`, try not `(between(startDate, endDate))`.
  6. Index dates as integers. This can save time and may even be faster if you have to do a conversion later on.
  7. Searches of eventually consistent caches are faster because queries are executed immediately, without waiting for pending transactions at the local node to commit. **Note:** This means that if a thread adds an element into an eventually consistent cache and immediately runs a query to fetch the element, it will not be visible in the search results until the update is published to the server.
  8. Because changes to the cache may not be available to queries until the changes are applied cluster wide (or, for transactional caches, `commit()` is called), it is possible to get results that include removed elements, inconsistent data from the same query executed at different times, or 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 values using keys returned by a query, use null guards.

## Enterprise Ehcache Cluster Events

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

### 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 <a href="#">Events API Example Code</a> ). |
| <code>Collection&lt;Node&gt; getNodes()</code> | Returns information on all the nodes in the cluster, including ID, hostname, and IP address.  |

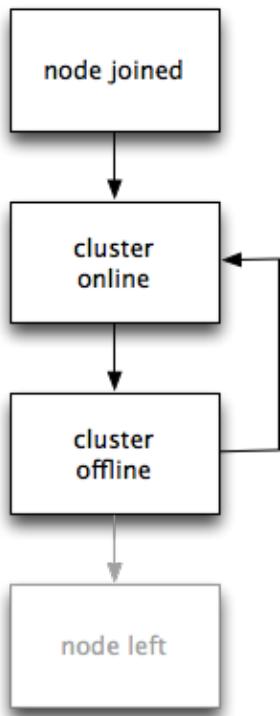
## Cluster Topology

|   |   |
|---|---|
| boolean addTopologyListener(ClusterTopologyListener listener) | Adds a cluster-events listener.<br>Returns true if the listener is already active.      |
| boolean removeTopologyListener(ClusterTopologyListener)       | Removes a cluster-events listener.<br>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. |

## Cluster Events



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

- `nodeJoined(ClusterNode)` – Indicates that a node has joined the cluster. All nodes except the current node detect this event once per the current node's lifetime. For the current node, see [nodeJoined for the Current Node](#).
- `nodeLeft(ClusterNode)` – Indicates that a node has left the cluster. All remote nodes can detect this event once per the current node's lifetime. The current node may fail to detect this event, but if it is still functioning it may automatically rejoin the cluster (as a new node) once the connection to the cluster is reestablished.
- `clusterOnline(ClusterNode)` – The current node is able to perform operations within the cluster. This event can be received by the current node only after its `nodeJoined` event is emitted.

## Cluster Events

- `clusterOffline(ClusterNode)` – The current node is unable to perform operations within the cluster. This event can be received by the current node only after the `clusterOnline` event.

## Events API Example Code

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

### 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");`

### **nodeJoined for the Current Node**

Since the current node joins the cluster before code adding the topology listener runs, the current node may never receive the `nodeJoined` event. You can detect if the current node is in the cluster by checking if the cluster is online:

```
cluster.addTopologyListener(cacheListener);  
if(cluster.isClusterOnline()) {  
    cacheListener.clusterOnline(cluster.getCurrentNode());  
}
```

## 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

## Bulk-Load API

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  |
|--|---|
| <pre>public boolean<br/>isClusterBulkLoadEnabled()</pre>   | Returns true if a cache is in bulk-load mode (is *not* consistent) throughout the cluster. Returns false if the cache is not in bulk-load mode (*is* consistent) anywhere in the cluster.   |
| <pre>public boolean<br/>isNodeBulkLoadEnabled()</pre>      | Returns true if a cache is in bulk-load mode (is *not* consistent) on the current node. Returns false if the cache is not in bulk-load mode (*is* consistent) on the current node.  |
| <pre>public void<br/>setNodeBulkLoadEnabled(boolean)</pre> | 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 <a href="#">nonstop cache</a> , 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 <a href="#">Tuning Nonstop Timeouts and Behaviors</a> . |
| <pre>public void<br/>waitUntilBulkLoadComplete()</pre>     | 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.   |

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 `ObjectNotFoundException`s that may occur are logged.
- `get()` methods that fail with `ObjectNotFoundException` return null.
- Eviction is independent of consistency mode. Any configured or manually executed eviction proceeds unaffected by a cache's consistency mode.

## 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.
```

## Bulk-Load API Example Code

```
// Load data into myCache...

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

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.
}

...
...
```

## 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 of 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">
    <persistence strategy="distributed"/>
    <terracotta clustered="true" consistency="strong" />
</cache>
```

## Unlocked Reads for Consistent Caches(UnlockedReadsView)

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.unlockedreadsvew.UnlockedReadsView.

| Method   | Definition  |
|--|---|
| public String getName()  | Returns the name of the unlocked cache view.  |
| public Element get(final Object key)<br>`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. |

## UnlockedReadsView and Data Freshness

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

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

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

## 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).                                 |
| public boolean isReadLockedByCurrentThread (Object key)  | Returns true if the current thread holds a read lock on the element specified by the argument (key).  |
| public boolean isWriteLockedByCurrentThread (Object key) | Returns true if the current thread holds a write lock on the element specified by the argument (key). |

## Explicit Locking

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(key1);
}
```

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

## 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);
...
```

## 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.

## Write-Behind Queue in Enterprise Ehcache

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

```
<cache name="myCache" eternal="false" maxElementsInMemory="1000">
    <persistence strategy="none"/>
    <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.

# Enterprise Ehcache Configuration Reference

Enterprise Ehcache uses the standard Ehcache configuration file to set clustering and consistency behavior, optimize cached data, support for Java Transaction API (JTA) and OSGi, and more.

## 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 [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 [How Configuration Affects Element Eviction](#) for more information). You can bypass this issue by using "eventual" consistency mode (see [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 [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.

## 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. Under most circumstances, the concurrency value is optimized by the Terracotta Server Array and does not require tuning.

If an explicit and fixed segmentation value must be set, use the `<terracotta>` element's concurrency attribute, making sure to set an appropriate concurrency value. A too-low concurrency value could cause unexpected eviction of elements. A too-high concurrency value may create many empty segments on the Terracotta Server Array (or many segments holding a few or just one element). In this case, `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 empty.

The following information provides additional guidance for choosing a concurrency value:

- With extremely large data sets, a high concurrency value can improve performance by hashing the data into more segments, which reduces lock contention.
- If `maxElementsOnDisk` is not set, set to 0, or set to a value equal to or greater than 256, set concurrency equal to 256 (except for extremely large data sets). This is the default value.
- 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.
- In environments with very few cache elements or a very low `maxElementsOnDisk` value, be sure to set concurrency to a value close to the number of expected elements.
- In general, the concurrency value should be no less than than the number of active servers in the Terracotta Server Array, and optimally at least twice the number of active Terracotta servers.

To learn how to set concurrency for a cache, see the section on the `<terracotta>` element.

## Non-Blocking Disconnected (Nonstop) Cache

A nonstop cache allows certain cache operations to proceed on clients that have become disconnected from the cluster or if a cache operation cannot complete by the nonstop timeout value. One way clients go into nonstop mode is when they receive a "cluster offline" event. Note that a nonstop cache can go into nonstop mode even if the node is not disconnected, such as when a cache operation is unable to complete within the timeout allotted by the nonstop configuration.

## 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">
    <persistence strategy="distributed"/>
    <terracotta>
        <nonstop immediateTimeout="false" timeoutMillis="30000">
            <timeoutBehavior type="noop" />
        </nonstop>
    </terracotta>
</cache>
```

## Configuring Nonstop

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

## 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 actions 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, the first request made by a client can take up to the time specified by `timeoutMillis` and subsequent requests timeout 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 |
|---|----------|
| (DEFAULT) Throw NonStopCacheException. See <a href="#">When is NonStopCacheException Thrown?</a> for more information on this exception.  |          |
| 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.    |          |
| 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.

### Network Interruptions

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`.

### Slow Cache Operations

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.

For example, a `cache.acquireWriteLockOnKey(key)` operation may exceed the nonstop timeout while waiting for a lock. This would trigger nonstop mode only because the lock couldn't be acquired in time. Using `cache.tryWriteLockOnKey(key, timeout)`, with the method's timeout set to less than the

## Nonstop Timeouts and Behaviors

nonstop timeout, avoids this problem.

### Bulk Loading

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:

```
-Dnet.sf.ehcache.nonstop.bulkOpsTimeoutMultiplyFactor=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 [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(); }`

## 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

## How Configuration Affects Element Eviction

on that element (sometimes called *inline* eviction).

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. This storage approach is referred to as "DCV2" (Distributed Cache v2).

### TIP: Eviction With UnlockedReadsView and Bulk Loading

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

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

Flushing 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: 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 cache entries if servers are configured to have infinite store (`maxElementsOnDisk` is not set or is set to 0). A server may also not evict cache entries if they remain resident in any client cache. Under these conditions, **the expected data set must fit in the server array or the cluster may suffer from performance degradation and errors.**

To learn about eviction and controlling the size of the cache, see the Ehcache documentation on [data life](#) and [sizing caches](#).

## Strict Consistency, UnlockedReadsView, and Bulk Loading

When a cache that strict consistency is decorated with `UnlockedReadsView` (see [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

## Strict Consistency, UnlockedReadsView, and Bulk Loading

`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 [Bulk-Load API](#)).

## 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:

- **Eventual** – This mode guarantees that data in the cache will eventually be consistent. Read/write performance is substantially boosted at the cost of potentially having an inconsistent cache for brief periods of time. This mode is set using the Ehcache configuration file and cannot be changed programmatically (see the attribute "consistency" in `<terracotta>`).
- **Strong** – This mode ensures that data in the cache remains consistent across the cluster at all times. It guarantees that a read gets an updated value only after all write operations to that value are completed, and that each put operation is in a separate transaction. The use of locking and transaction acknowledgments maximizes consistency at **a potentially substantial cost in performance**. This mode is set using the Ehcache configuration file and cannot be changed programmatically (see the attribute "consistent" 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 [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 cluster-wide (application-level) locking on specific elements in a cache. There is guaranteed consistency across the cluster at all times for operations on elements covered by a lock. When used with the strong consistency mode in a cache, *each cache operation* is committed in a single transaction. When used with the eventual consistency mode in a cache, *all cache operations covered by an explicit lock* are committed in a single transaction. 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 [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

## Understanding Performance and Cache Consistency

this mode by bypassing the requirement for a read lock. See [Unlocked Reads for Consistent Caches \(UnlockedReadsView\)](#) for more information.

- **Atomic methods** – To guarantee write consistency at all times and avoid potential race conditions for put operations, use the 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 [Explicit Locking](#)). Note that using locks may impact performance.
- **Bulk-loading methods** – Bulk-loading Cache methods `putAll()`, `getAll()`, and `removeAll()` provide high-performance and eventual consistency. These can also be used with strong consistency. If you can use them, it's unnecessary to use bulk-load mode. See the [API documentation](#) for details.

To optimize consistency and performance, consider using eventually consistent caches while selectively using explicit locking in your application where cluster-wide consistency is critical.

## 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** – 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 [Cache Events Configuration](#) for more information on configuring the scope of cache events.

## Handling Cache Update Events

Caches 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 [Explicit Locking](#)). You can also use the atomic cache methods `putIfAbsent()` or to check for the existence of an element:

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

## Handling Cache Update Events

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.

## 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 [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 [Using Rejoin to Automatically Reconnect Terracotta Clients](#) for more information.

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

## 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`.
- If Enterprise Ehcache client with rejoin enabled is running in a JVM with Terracotta clients that do not have rejoin, then only that client will rejoin after a disconnection. The remaining clients cannot rejoin and may cause the application to behave unpredictably.
- Once a client rejoins, the `clusterRejoined` event is fired on that client only.

## Using Rejoin to Automatically Reconnect Terracotta Clients

### 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" />
```

For more options on configuring `<terracottaConfig>`, see the [configuration reference](#).

### 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. Allocating a larger PermGen space can make an OOME less likely under these conditions.

The default value of PermGen on Oracle 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 OOMEs 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 OOMEs.

**TIP:** Use the Most Current Supported Version of the JDK

Rejoin operations are known to be more stable on JDK versions greater than 1.5.

### 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.

## 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 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:

## Working With Transactional Caches

- Data can be accessed only transactionally, even for read-only purposes. You must encapsulate data access with `begin()` and `commit()` statements. This may not be necessary under certain circumstances (see, for example, the discussion on [Spring in Transactions in Ehcache](#)).
- `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 except in strict mode (`xa_strict`). Transactional caches in other modes 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 [Implementing an Element Comparator](#).

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

- Strict XA – Has full 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.

## Strict XA (Support for All JTA Components)

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.

## Strict XA (Support for All JTA Components)

- 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"
      copyOnRead="true"
      copyOnWrite="true"
      consistency="strong"
      transactionalMode="xa_strict">
  <persistence strategy="distributed"/>
  <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 [Avoiding XA Commit Failures With Atomic Methods](#) for more information.

XA (Basic JTA Support)

## XA (Basic JTA Support)

Transactional caches set to "xa" provide support for basic JTA operations. Configuring and using XA does not differ from using local transactions (see [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"
      copyOnRead="true"
      copyOnWrite="true"
      consistency="strong"
      transactionalMode="xa">
  <persistence strategy="distributed"/>
  <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.

## 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

## Local Transactions

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.
```

## 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);  
  
// True only if the element was successfully replaced.  
if (cache.replace(olde, new Element(key, val + 1)) {  
    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.

## 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()`.

## Implementing an Element Comparator

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"
      copyOnRead="true"
      copyOnWrite="true"
      consistency="strong"
      transactionalMode="xa">
  <elementValueComparator class="com.company.xyz.MyElementComparator" />
  <persistence strategy="distributed"/>
  <terracotta clustered="true" valueMode="serialization" />
</cache>
```

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

A comparator can also be specified programmatically.

## 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>
```

Your OSGi bundle will require the following JAR files (showing version from a Terracotta 3.6.2 kit):

- `ehcache-core-2.5.2.jar
- `ehcache-terracotta-2.5.2.jar
- `slf4j-api-1.6.1.jar
- `slf4j-nop-1.6.1.jar

Or use another appropriate logger binding.

Use the following directory structure:

```
-- net.sf.ehcache
  |
  | - ehcache.xml
  |- ehcache-core-2.5.2.jar
  |
  |- ehcache-terracotta-2.5.2.jar
  |
  | - slf4j-api-1.6.1.jar
  |
  | - slf4j-nop-1.6.1.jar
  |
  | - META-INF/
    | - MANIFEST.MF
```

## Working With OSGi

The following is an example manifest file:

```
Manifest-Version: 1.0
Export-Package: net.sf.ehcache;version="2.5.2"
Bundle-Vendor: Terracotta
Bundle-ClassPath: .,ehcache-core-2.5.2.jar,
    ehcache-terracotta-2.5.2.jar,slf4j-api-1.6.1.jar,slf4j-nop-1.6.1.jar
Bundle-Version: 2.5.2
Bundle-Name: EHCache bundle
Created-By: 1.6.0_15 (Apple Inc.)
Bundle-ManifestVersion: 2
Import-Package: org.osgi.framework;version="1.3.0"
Bundle-SymbolicName: net.sf.ehcache
Bundle-RequiredExecutionEnvironment: J2SE-1.5
```

Use versions appropriate to your setup.

To create the bundle, execute the following command in the `net.sf.ehcache` directory:

```
jar cvfm net.sf.ehcache.jar MANIFEST.MF *
```

# Enterprise Ehcache for Hibernate

## Introduction

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 and quick installation procedure.
- [Testing and Tuning Enterprise Ehcache for Hibernate](#) Information to help locate trouble spots and find ideas for improving performance.
- [Enterprise Ehcache for Hibernate Reference](#) Covers configuration and other topics.

## Installing Enterprise Ehcache for Hibernate

### Step 1: Requirements

- JDK 1.6 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 3.7.4 package

### 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 (2.4.3 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`.

- \${TERRACOTTA\_HOME}/common/terracotta-toolkit-<API>-

## Step 2: Install and Update the JAR files

```
runtime-ee-<version>.jar
```

The Terracotta Toolkit JAR contains the Terracotta client libraries. <API> 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`:  
`<Dcom.sun.enterprise.server.ss.ASQuickStartup=false`
- 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`.

## 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 Echache 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.

## Step 3: Prepare Your Application for Caching

### 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.

### 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.

### 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.

## Step 4: Edit Configuration Files

# 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.

## Hibernate Configuration File

For Hibernate 4.x, add the following to your `hibernate.cfg.xml`:

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

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: `hibernate.cache.use_second_level_cache=true hibernate.cache.region.factory_class=net.sf.ehcache.hibernate.EhCacheRegionFactory`

For Hibernate 4, use `org.hibernate.cache.ehcache.EhCacheRegionFactory` instead of `net.sf.ehcache.hibernate.EhCacheRegionFactory`.

## 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
```

## Step 4: Edit Configuration Files

```
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>
```

### Expiration Settings

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

## Step 5: Start Your Application with the Cache

You must start both your application and a Terracotta server.

1. Start the Terracotta server with the following command:

### UNIX/Linux

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

### Microsoft Windows

## Step 5: Start Your Application with the Cache

```
[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.

## 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.

#### 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">
    </servers>

```

## Step 6: Edit the Terracotta Configuration

```
<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. --&gt;
&lt;clients&gt;
  &lt;logs&gt;%{user.home}/terracotta/client-logs&lt;/logs&gt;
&lt;/clients&gt;
&lt;/tc:tc-config&gt;</pre>
```

3. Install Terracotta 3.7.4 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.

## Step 7: Learn More

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

## Step 7: Learn More

- [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](#)

# 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 [Expiration Parameters](#) - Turn Off [Query Cache](#) - Reduce [Unnecessary Database Connections](#) - Configure Database [Connection Pools](#) - Turn off Unnecessary [Statistics Gathering](#)

## 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.

## Testing the Cache

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

## 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.

Explicit sizing of caches is discussed in the [Ehcache documentation](#).

## Expiration Parameters

Expiration is important because it forces unneeded data to be automatically evicted when accessed or when constraints require resources to be freed. The most important parameters for tuning cache size and cache performance in general are the following:

- `timeToIdle` (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.
- `timeToLive` (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. Other options for limiting the life of data and the size of caches are discussed in the [Ehcache documentation](#).

### How to Set Expiration Parameters

You can set expiration parameters in these ways:

- In `ehcache.xml` – Configuration file for Enterprise Ehcache for Hibernate with properties for controlling expiration on a per-cache basis. See the [Ehcache documentation](#) for more information.
- In the Terracotta Developer Console – The GUI for Hibernate second-level cache allows you to apply real-time values to expiration parameters and export a configuration file. For more information, see [Enterprise Ehcache for Hibernate Applications](#).
- Programmatically – When creating caches programmatically.

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

## 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 expiration parameter

## Optimizing the Cache Size

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.

### Expiration Example

Applications that use Enterprise Ehcache for Hibernate to implement the second-level should have TTI and TTL properly tuned to prevent unnecessarily large data caches and stale values.

The following sections detail how certain cached data is configured for second-level caching in a test-taking web-based application that can handle thousands of concurrent users. 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"
    maxEntriesLocalHeap="1000"
    eternal="false"
    timeToIdleSeconds="0"
    timeToLiveSeconds="0">
    <persistence strategy="distributed"/>
    <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 evicted 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"
    maxEntriesLocalHeap="1000"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="00">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.User.roles"
    maxEntriesLocalHeap="1000"
    eternal="false"
```

## Optimizing the Cache Size

```
timeToIdleSeconds="120"
timeToLiveSeconds="0">
<persistence strategy="distributed"/>
<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"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToIdleSeconds="3600"
      timeToLiveSeconds="86400">
  <persistence strategy="distributed"/>
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Section"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToIdleSeconds="3600"
      timeToLiveSeconds="86400">
  <persistence strategy="distributed"/>
  <terracotta/>
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Section.questions"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToIdleSeconds="3600"
      timeToLiveSeconds="86400">
  <persistence strategy="distributed"/>
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Section.sections"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToIdleSeconds="3600"
      timeToLiveSeconds="86400">
  <persistence strategy="distributed"/>
  <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Question"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToIdleSeconds="3600"
      timeToLiveSeconds="86400">
  <persistence strategy="distributed"/>
  <terracotta/>
```

## Optimizing for Read-Only Data

```
</cache>
<cache name="org.terracotta.reference.exam.domain.Question.choices"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToIdleSeconds="3600"
      timeToLiveSeconds="86400">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
<cache name="org.terracotta.reference.exam.domain.Choice"
      maxEntriesLocalHeap="1000"
      eternal="false"
      timeToIdleSeconds="3600"
      timeToLiveSeconds="86400">
    <persistence strategy="distributed"/>
    <terracotta/>
</cache>
```

## Optimizing for Read-Only Data

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

- Turn off expiration for often-used, long-lived data.
  - See the [Ehcache documentation](#) for more information on configuring data lifetimes.
- Turn on key caching (see [Local Key Cache](#)).
- Eliminate "empty" database connections (see [Reducing Unnecessary Database Connections](#)).

## 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](#)

## Reducing Unnecessary Database Connections

### 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.

### 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.
```

## Reducing Memory Usage with Batch Processing

```
        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>
```

## 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` (OOME) during large operations of this type.

You can prevent OOMEs 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
        tx.commit();
        tx = session.beginTransaction();
    }
}
```

## Other Important Tuning Factors

```
//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](#).

## Other Important Tuning Factors

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

### 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").

### 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.

### 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.

## Other Important Tuning Factors

### 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.

### 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.

### 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:

- [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" />
```

## Other Important Tuning Factors

```
</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 `12.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.12.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="12.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.

## Other Important Tuning Factors

If you lower the value of `12.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 `12.berkeleydb.je.maxMemoryPercent` incrementally until an optimal level is observed.

## 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**.

## 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.

## 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.

## 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.

## 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.

## Other Important Tuning Factors

### 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 issues](#) for more information.

### 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.

## Enterprise Ehcache for Hibernate Reference

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

### 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*.

### 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 [Setting Cache Eviction](#).

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

### Cache-Configuration File Properties

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

## Cache Configuration File

### Exporting Configuration from the Developer Console

To create or edit a cache configuration in a live cluster, see [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.

### 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](#).

### 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.

#### **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.

**WARNING: Using this strategy with transactional caches can cause unpredictable results.**

#### **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.

**WARNING: To avoid errors and unpredictable behavior, use this strategy only with caches that have strong consistency and do not have nonstop mode enabled.**

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.

## Cache Concurrency Strategies

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.

### **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.

This strategy works best with caches that have strong consistency. It will show some degradation with caches that have eventual consistency instead. It may be able to continue functioning with nonstop mode, though will do better without it.

**WARNING: Using this strategy with transactional caches can cause unpredictable results.**

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.

### **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.

**WARNING: Use this strategy with transactional caches only. Using with other types of caches will cause errors.**

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](#).

### **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.

## Setting Up Transactional Caches

With NONSTRICT\_READ\_WRITE, it is possible that an entitymanager will query the database more often if frequent updates are invalidating target cache entries.

With READ\_WRITE, updates do not invalidate the cache, and so an entitymanager may read from the cache (same as TRANSACTIONAL). However, under READ\_WRITE, an entitymanager will have to read from the database if the target cache entry is under a lock at the time the read attempt is made.

NONSTRICT\_READ\_WRITE, on the other hand, may read a stale value from the cache if the read attempt is made before the transaction completes.

READ\_WRITE also forces use of the entitymanager's timestamp for comparison purposes when evaluating the freshness of data, which again can lead to more database access operations.

## Setting Up Transactional Caches

If your application is using JTA, you can set up transactional caches in a second-level cache with Enterprise Ehcache for Hibernate. To do so, ensure the following:

### Ehcache

- You are using Ehcache 2.1.0 or higher.
- The attribute transactionalMode is set to "xa" or "xa-strict".
- 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"
      ...
      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"/>
```

## Configuring Multiple Hibernate Applications

Or using annotations:

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

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

## 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.

## 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.

## 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
```

## Cache Regions in the Object Browser

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

## 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).

## Is a Cache Appropriate for Your Use Case?

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

### 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](#)).

### 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.

### 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. An [asynchronous approach](#) to writing the data may be a good solution for this issue.

### 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

## Is a Cache Appropriate for Your Use Case?

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.

## 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.

## 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.

# Clustering Quartz Scheduler

Quartz version 2.1

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:

## Step 1: Requirements

- JDK 1.6 or higher.
- [Terracotta 3.7.4 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.

## 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>-  
runtime-ee-<version>.jar

The Terracotta Toolkit JAR contains the Terracotta client libraries. <API> 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.

## Step 2: Install Quartz Scheduler

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`:  
~~~ -Dcom.sun.enterprise.server.ss.ASQuickStartup=false ~~~ - WebLogic application server – You must use the [supported version](#) of WebLogic.

## 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`.

### 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 a 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.

### Scheduler Instance Name

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

## Scheduler Instance Name

```
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.

## 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:

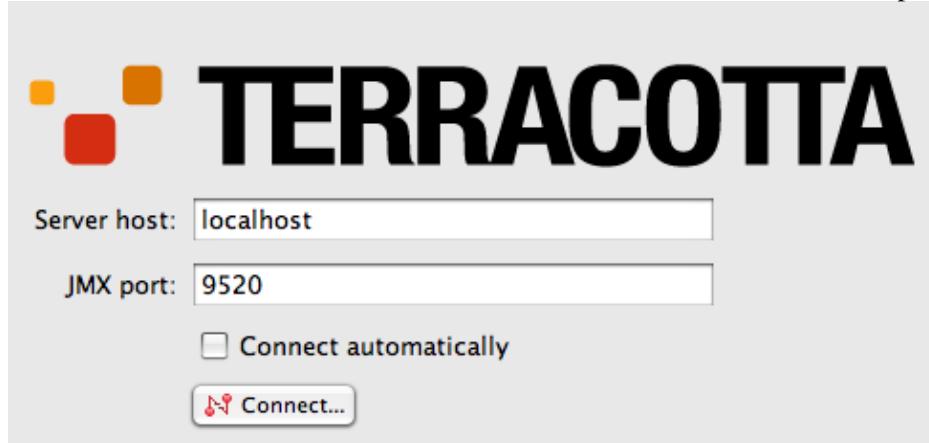
### UNIX/Linux

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

### Microsoft Windows

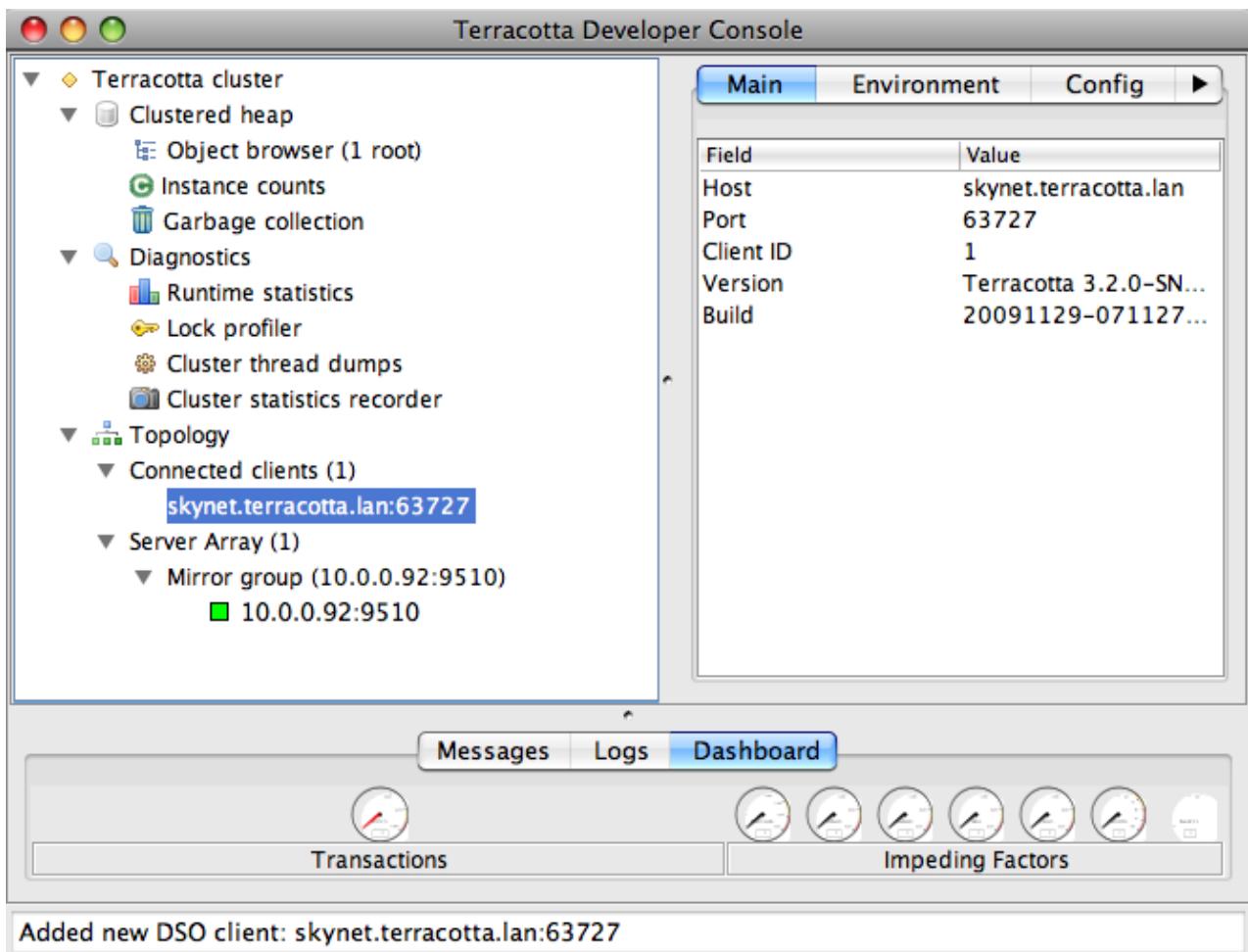
```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

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.

#### Step 4: Start the Cluster



## 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.

### 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">
```

## Procedure:

```
<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 3.7.4 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.

## Step 6: 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 Developer Console](#) – Provides visibility into and control of caches.

# Quartz Scheduler Where (Locality API)

## Introduction

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. You should already be familiar with using Quartz Scheduler (see the [installation guide](#) and the [Quartz Scheduler documentation](#)).

## 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 [Terracotta DSO Installation](#).

## Configuring Quartz Scheduler Where

To configure Quartz Scheduler Where, follow these steps:

1. Edit `quartz.properties` to cluster with Terracotta. See [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 [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`.

## Configuring Quartz Scheduler Where

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

# 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

## Using SystemPropertyInstanceIdGenerator

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
```

## Available Constraints

Quartz Scheduler Where offers the following constraints:

- CPU – Provides methods for constraints based on minimum number of cores, available threads, and maximum amount of CPU 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.

See the code samples provided below for how to use these constraints.

## 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;
```

## Quartz Scheduler Where Code Sample

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

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

***** Node Group + OS Constraint
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(OsConstraint.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();

***** Memory Constraint
Create a locality-aware trigger for myJob2 that will fire on any
node that has a certain amount of free memory available:
****/

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

***** A Locality-Aware Trigger For an Existing Job
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"))
```

## CPU-Based Constraints

```
.where(
    node()
        .is(partOfNodeGroup("allNodes")))
.build();

***** Locality Constraint Based on Cache Keys

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.

*****/
```

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

// 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. If any of the constraints in effect — trigger or job — are not met, the trigger will go into an error state and the job will not be fired.

## CPU-Based Constraints

The CPU constraint allows you to run jobs on machines with adequate processing power:

```
...

import static org.quartz.locality.constraint.CpuConstraint.loadAtMost;

...

// Create a locality-aware trigger for someJob.
LocalityTrigger trigger =
    localTrigger(newTrigger()
        .forJob("someJob")
```

## Failure Scenarios

```
.withIdentity("someTrigger"))
.where(
    node()
        // fire on any node in allNodes
        // with at most the specified load:
        .is(partOfNodeGroup("allNodes"))
        .has(loadAtMost(.80)))
.build();
```

The load constraint refers to the CPU load (a standard \*NIX load measurement) averaged over the last minute. A load average below 1.00 indicates that the CPU is likely to execute the job immediately. The smaller the load, the freer the CPU, though setting a threshold that is too low could make it difficult for a match to be found.

Other CPU constraints include `CpuConstraint.coresAtLeast(int amount)`, which specifies a node with a minimum number of CPU cores, and

`CpuConstraint.threadsAvailableAtLeast(int amount)`, which specifies a node with a minimum number of available threads.

### 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.

## Failure Scenarios

If a trigger cannot fire on the specified node or targeted node group, the associated job will not execute. Once the `misfireThreshold` timeout value is reached, the trigger misfires and any misfire instructions are executed.

## 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`.

# Quartz Scheduler Reference

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

## 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.

## 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.

### 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.

### 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 objects into a clustered JobDataMap could also introduce other errors that are avoided with Strings and primitives.

## 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:

## Cluster Data Safety

```
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.

## 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.

# 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](#).

## Step 1: Requirements

- JDK 1.6 or higher.
- [Terracotta 3.7.4 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](#).

## 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 files 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>-runtime-ee-<version>.jar

The Terracotta Toolkit JAR contains the Terracotta client libraries. <API> 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 sessions and toolkit JAR files 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 (REQUIRED)           |
| JBoss AS 7.0 or 7.1         | <jboss install dir>/WEB-INF/lib                     |
| Jetty                       | WEB-INF/lib                                         |
| Tomcat 5.0 and 5.5          | \$CATALINA_HOME/server/lib                          |
| Tomcat 6.0 and 7.0          | \$CATALINA_HOME/lib                                 |

## Step 2: Install the Terracotta Sessions JAR

WebLogic                   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.

## 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.

### 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 <code>&lt;filter-class&gt;</code>                                           |
|-------------|--------------------------------------------------------------------------------------|
| Jetty 6.1   | <code>org.terracotta.session</code><br><code>.TerracottaJetty61xSessionFilter</code> |
| Jetty 7.4.1 |                                                                                      |

## Jetty, WebLogic, and WebSphere

|               |                                                                |
|---------------|----------------------------------------------------------------|
|               | org.terracotta.session<br>.TerracottaJetty74xSessionFilter     |
| WebLogic 9    | org.terracotta.session<br>.TerracottaWeblogic9xSessionFilter   |
| WebLogic 10   | org.terracotta.session<br>.TerracottaWeblogic10xSessionFilter  |
| WebSphere 6.1 | org.terracotta.session<br>.TerracottaWebsphere61xSessionFilter |
| WebSphere 7.0 | org.terracotta.session<br>.TerracottaWebsphere70xSessionFilter |

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.

## Tomcat and JBoss AS 6.0 or Earlier

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<br>.TerracottaJboss40xSessionValve |
| JBoss Application Server 4.2 |                                                           |

## Tomcat and JBoss AS 6.0 or Earlier

|                              |                                                            |
|------------------------------|------------------------------------------------------------|
|                              | org.terracotta.session<br>.TerracottaJboss42xSessionValve  |
| JBoss Application Server 5.1 | org.terracotta.session<br>.TerracottaJboss51xSessionValve  |
| JBoss Application Server 6.0 | org.terracotta.session<br>.TerracottaJboss60xSessionValve  |
| Tomcat 5.0                   | org.terracotta.session<br>.TerracottaTomcat50xSessionValve |
| Tomcat 5.5                   | org.terracotta.session<br>.TerracottaTomcat55xSessionValve |
| Tomcat 6.0                   | org.terracotta.session<br>.TerracottaTomcat60xSessionValve |
| Tomcat 7.0                   | org.terracotta.session<br>.TerracottaTomcat70xSessionValve |

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: If you are using a WAR file, `context.xml` should be in /META-INF for Tomcat and in /WEB-INF for JBoss Application Server.

## JBoss AS 7.0 or 7.1

Add the following to WEB-INF/jboss-web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>

<jboss-web>
...
<valve>
  <class-name>org.terracotta.session.TerracottaJboss7xSessionValve</class-name>
  <param>
    <param-name>tcConfigUrl</param-name>
    <param-value>localhost:9510</param-value>
  </param>
</valve>
...
</jboss-web>
```

where the value for `tcConfigUrl` contains an URL or file path (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

JBoss AS 7.0 or 7.1

configuration.

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

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

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

## 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:

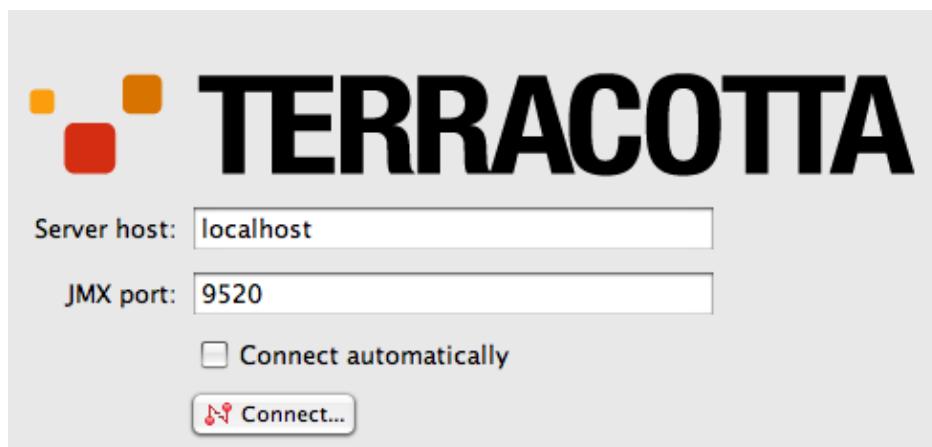
### UNIX/Linux

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

### Microsoft Windows

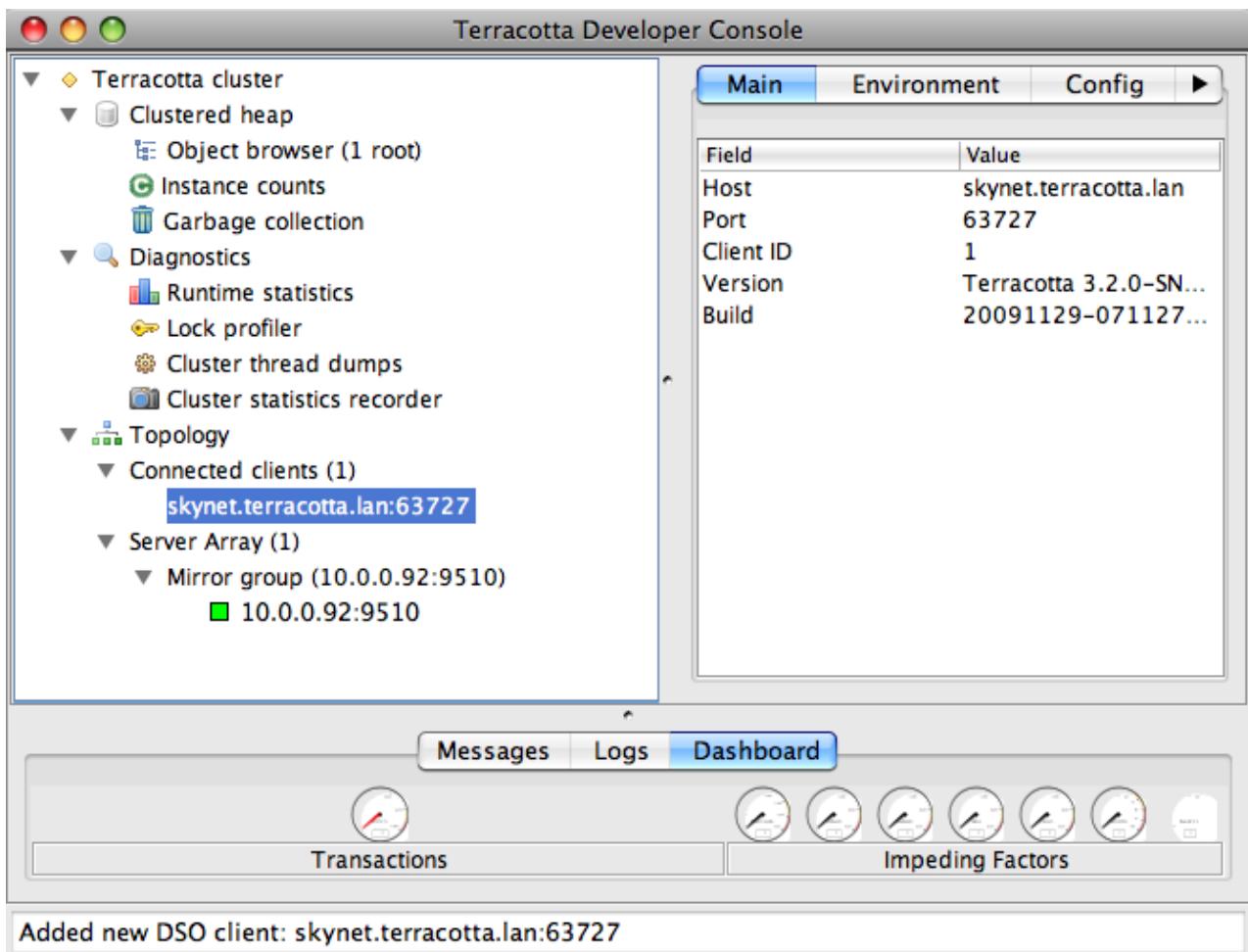
```
[PROMPT] ${TERRACOTTA_HOME}\bin\dev-console.bat
```

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.

#### Step 4: Start the Cluster



## 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.

### 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">
```

## Procedure:

```
<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 3.7.4 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.

## Step 6: Learn More

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

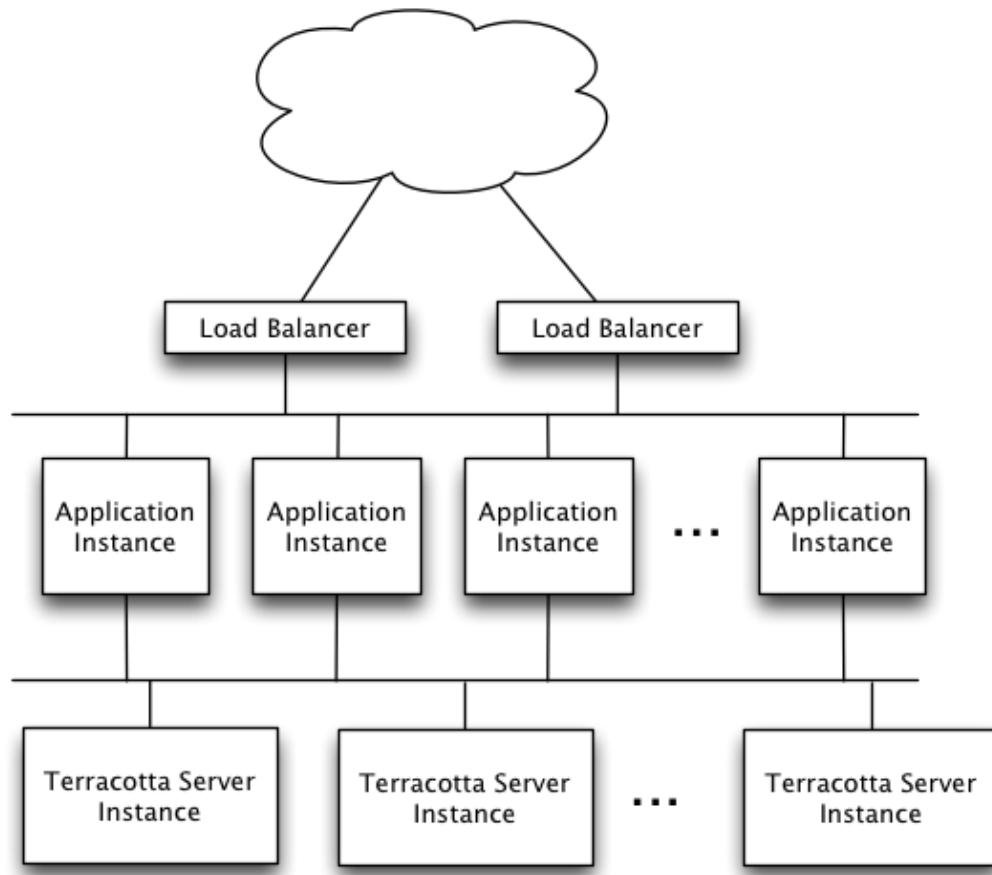
- [Working with Terracotta Configuration Files](#) – Explains how `tc-config.xml` is propagated and loaded in a Terracotta cluster in different environments.
- [Terracotta Server Arrays](#) – Shows how to design Terracotta clusters that are fault-tolerant, maintain data safety, and provide uninterrupted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

# Web Sessions Reference

This section contains further information on configuring and troubleshooting Terracotta Web Sessions.

## 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](#).

# 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.

## 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 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 [Deadlocks When Session Locking Is Enabled](#).

## 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 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>
```

## Synchronous Writes

To enable synchronous writes in Valve-based configuration, add a `synchronousWrite` attribute as follows:

```
<Valve className="com.terracotta.TerracottaContainerSpecificSessionValve"  
tcConfigUrl="localhost:9510" synchronousWrite="true"/>
```

# Troubleshooting

The following sections summarize common issues than can be encountered when clustering web sessions.

## 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.

## Changes Not Replicated

Terracotta Web Sessions must run in serialization mode. Serialization mode is not an option as it is in the DSO 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.

## 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.

## 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.

## 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 one node and destroyed on a second node, the event is received on the second node, not the first node.

# Security Overview

## Introduction

Security can be applied to both authentication (such as login credentials) and authorization (the privileges of specific roles).

We recommend that you plan and implement a security strategy that encompasses all of the points of potential vulnerability in your environment, including, but not necessarily limited to, your application servers (Terracotta clients), Terracotta servers in the TSA, the Terracotta Management Console (TMC), and any BigMemory .NET or C++ clients.

**Note:** Terracotta does not encrypt the data on its servers, but applying your own data encryption is another possible security measure.

## For Details

See [Security Overview for 4.1](#).

# About Terracotta DSO Installations

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 [resources page](#) before proceeding with the DSO installation. For more information on comparing standard and DSO installation methods, see [Standard Versus DSO Installations](#).

## 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**.

## 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 [Performing a DSO Installation](#).

# Performing a DSO Installation

## Introduction

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

## Prerequisites

- JDK 1.5 or higher. See the [certified-platforms page](#) for certified JVMs.
- [Terracotta 3.7.4 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.

## 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 is the version of Ehcache (2.4.3 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.
- 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.

## 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.

## 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.

## 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:

## TIMs for Clustering Enterprise Ehcache

```
<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.

## 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.

## 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.

## TIMs for Integrating an Application Server

### **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:

## Clustering a Web Application with Terracotta Web Sessions

### 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/lib/dso-boot/dso-boot-hotspot_linux_150_16.jar'
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-
```

## 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.

## 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](#)

### 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;
```

## Enterprise Ehcache Configuration

```
// 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 LFU or LRU. Setting MemoryStoreEvictionPolicy to FIFO causes the error `IllegalArgumentException`.

## 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.

## Enterprise Ehcache for Hibernate Configuration

### 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="LFU">
  <!-- 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](#).

## 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>.

## 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](#).

## 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

### 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. -->
```

## Location of TIMs

```
</modules>
```

The Terracotta Toolkit API version available for your Terracotta kit may be different than the one shown in this example.

## Location of TIMs

Generally, TIMs are found on the following path:

```
 ${TERRACOTTA_HOME}/platform/modules/org/terracotta/modules/tim-<name>-<version>/<TIM-version>/tim
```

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-
```

The Terracotta Toolkit (terracotta-toolkit) is found in:

```
 ${TERRACOTTA_HOME}/platform/modules/org/terracotta/toolkit/terracotta-toolkit-1.0/1.0.0/terracott
```

## 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

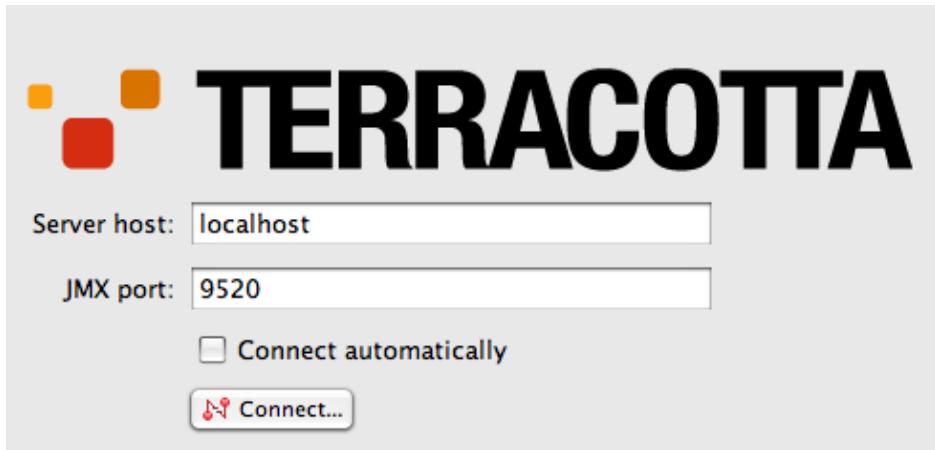
```
[PROMPT] ${TERRACOTTA_HOME}/bin/dev-console.sh &
```

### Microsoft Windows

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

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

#### Step 4: Start the Cluster



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.

A screenshot of the Terracotta Developer Console. The interface has a title bar "Terracotta Developer Console" with three icons (red, yellow, green). On the left is a sidebar with icons for Ehcache, Hibernate, Quartz, and Sessions. The "Ehcache" icon is selected. The main area shows a tree view under "My application": "Terracotta cluster" is expanded, showing "Ehcache" which is also expanded, showing "Monitoring", "Topology", "Connected clients (1)", "Server Array (1)", and "Platform". A blue callout box points to the "Ehcache Node" entry. The right side of the interface is the "Terracotta Distributed Cache Overview Panel". It shows a table for "Instances of CacheManager CacheMan1":

| Node             | Caches | Enabled | BulkLoading | Statistics |
|------------------|--------|---------|-------------|------------|
| 10.2.0.133:52892 | 2      | 2       | 0           | 2          |

Below the table is a section titled "Terracotta Distributed Cache Overview Panel" with a "Transactions" section containing several circular performance indicators. The bottom of the screen shows a message: "Ehcache 'MyCache2' from CacheManager 'CacheMan1' has changed".

## Quartz Scheduler Where 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 [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>
```

## Quartz Scheduler Where DSO Installation

```
...
<modules>
  <module name="tim-quartz-2.0-ee" />
  ...
</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.