BigMemory Max Administrator Guide

Version 4.3

April 2015

TERRACOTTA

This document applies to Terracotta Server Array Version 4.3 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: BMM-AG-43-20161110

Table of Contents

About the Terracotta Server Array....7
What is the Terracotta Server Array?....8
New for BigMemory Max 4.x....8
Definitions and Functional Characteristics....9

Terracotta Server Array Architecture....13
Terracotta Cluster in Development....14
Terracotta Cluster with Reliability....15
Terracotta Cluster with High Availability....17
Scaling the Terracotta Server Array....21

Configuring the Terracotta Server Array....25
About Terracotta Server Configuration....26
How Terracotta Servers Get Configured....26
How Terracotta Clients Get Configured....28
Configuration in a Development Environment....30
Configuration in a Production Environment....32
Binding Ports to Interfaces....34
Which Configuration?....35

Automatic Resource Management....37
What is Automatic Resource Management?....38
Eviction....38
Customizing the Eviction Strategy....42

Managing Near-Memory-Full Conditions....43
Behavior of the TSA under Near-Memory-Full Conditions....44
Restricted Mode Operations....45
Recovery....45

Monitoring Cluster Events....47
About Cluster Events....48
Event Types and Definitions....48

Backing Up Live In-Memory Data....55
About Live Backup....56
Creating a Backup....56
The Backup Directory....56
Restoring Data from a Backup....57

Clearing Data from a Terracotta Server....59
How to Clear Data from a Terracotta Server....60

Changing Topology of a Live Cluster....61
About Changing the Topology....62
Adding a New Server....62
Removing an Existing Server....63
Editing the Configuration of an Existing Server....63

Enabling Production Mode....65
Setting the Production Mode Property....66

Managing Distributed Garbage Collection....67
About Distributed Garbage Collection (DGC)....68
Running the Periodic Distributed Garbage Collection....68
Monitoring and Troubleshooting DGC....68

Starting the Terracotta Server as a Windows Service....69
Configuring the Terracotta Server to Run as a Service....70

Using BigMemory Hybrid....73
About BigMemory Hybrid....74
System Requirements....76
Hardware Capacity Guidelines....76
Configuring BigMemory Hybrid....76
Using the TMC with BigMemory Hybrid....77
Operator Events....78

Monitoring and Management Using JMX....79
About Using JMX....80
MBeans....81
JMX Remoting....81
ObjectName Naming Scheme....82
The Management Service....82
JConsole Example....83
JMX Tutorial....84
Performance Considerations....84
SSL-Secured Monitoring with JMX....84
Troubleshooting....86

Logging....91
SLFJ Logging....92
Recommended Logging Levels....92

Operational Scripts....93
Archive Utility (archive-tool)....94
Database Backup Utility (backup-data)....94
Backup Status (backup-status)....95
Cluster Thread and State Dumps (debug-tool, cluster-dump)....95

Distributed Garbage Collector (run-dgc)..... 96
Start and Stop Server Scripts (start-tc-server, stop-tc-server)..... 97
Server Status (server-stat)..... 98
Version Utility (version)..... 99

Terracotta Configuration Parameters..... 101
The Terracotta Configuration File..... 102
The Servers Parameters..... 105
/tc:tc-config/servers..... 105
/tc:tc-config/servers/server..... 105
/tc:tc-config/servers/server/data..... 106
/tc:tc-config/servers/server/logs..... 106
/tc:tc-config/servers/server/index..... 106
/tc:tc-config/servers/server/data-backup..... 106
/tc:tc-config/servers/server/tsa-port..... 107
/tc:tc-config/servers/server/jmx-port..... 107
/tc:tc-config/servers/server/tsa-group-port..... 107
/tc:tc-config/servers/server/management-port..... 107
/tc:tc-config/servers/server/security..... 108
/tc:tc-config/servers/server/security/ssl/certificate..... 108
/tc:tc-config/servers/server/security/keychain..... 108
/tc:tc-config/servers/server/security/auth..... 108
/tc:tc-config/servers/server/security/management..... 109
/tc:tc-config/servers/server/authentication..... 109
/tc:tc-config/servers/dataStorage..... 109
/tc:tc-config/servers/mirror-group..... 110
/tc:tc-config/servers/garbage-collection..... 111
/tc:tc-config/servers/restartable..... 112
/tc:tc-config/servers/client-reconnect-window..... 112
The Clients Parameters..... 112
/tc:tc-config/clients/logs..... 112

1 About the Terracotta Server Array

- What is the Terracotta Server Array? 8
- New for BigMemory Max 4.x 8
- Definitions and Functional Characteristics 9

What is the Terracotta Server Array?

The Terracotta Server Array (TSA) 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:

- **Distributed In-memory Data Management** - Manages 10-100x more data in memory than data grids
- **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. For information, see "Configuring the HealthChecker Properties" in the *BigMemory Max High-Availability Guide*.
- **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

New for BigMemory Max 4.x

The 4.x TSA is an in-memory data platform, providing faster, more consistent, and more predictable access to data. With resource management, if you have more data than memory available, the TSA protects itself from going over its limit through data eviction and throttling. In most cases, it will recover and come back to its normal working state automatically. In addition, four systems are available to protect data: the Fast Restart feature, BigMemory Hybrid's use of SSD/Flash, active-mirror server groups, and backups.

Fast Restartability for Data Persistence

BigMemory's Fast Restart feature is now integrated into the TSA, providing crash resilience with quick recovery, plus a consistent record of the entire in-memory data set, no matter how large. For more information, see "Fast Restartability" on page 15.

Hybrid Data Storage

"BigMemory Hybrid" extends BigMemory distributed in a Terracotta Server Array so that data can be stored across a hybrid mixture of RAM and SSD/Flash. This additional storage is managed with the in-memory data as one TSA data set. For more information, see "Using BigMemory Hybrid" on page 73.

Resource Management

Resource management provides better control over the TSA's in-memory data through time, size, and count limitations. This enables automatic handling of, and recovery from, near-memory-full conditions. For more information, see "Automatic Resource Management" on page 37.

Predictable Eviction Strategy

Based upon user-configured time, size, and count limitations, the TSA's 3-pronged eviction strategy works automatically to ensure predictable behavior when memory becomes full. For more information, see "Eviction " on page 38.

Continuous Uptime

Improvements to provide continuous availability of data include flexibility in server startup sequencing, better utilization of extra mirrors in mirror groups, multi-stripe backup capability, optimizations to bulk load, and performance improvements for data access on rejoin. In addition, the TSA no longer uses Oracle Berkeley DB, enabling in-memory data to be ready for use much more quickly after any planned or unplanned restart.

Terracotta Management Console (TMC)

The expanded TMC replaces the Developer Console and Operations Center as the integrated platform for monitoring, managing, and administering all Terracotta deployments. There is also support for additional REST APIs for management and monitoring. For more information, start with the *Terracotta Management Console User's Guide*.

Additional Security Features

Active Directory (AD) and Lightweight Directory Access Protocol (LDAP) support on Terracotta servers, and custom SecretProvider on Terracotta clients. For more information, see the *BigMemory Max Security Guide*.

No More DSO, plus Simplified Configuration

DSO configuration has been deprecated, and the tc-config has a new format. Most of the elements are the same, but the structure is revised. For more information, see the *BigMemory Max Administrator Guide*.

Definitions and Functional Characteristics

The major components of a Terracotta installation are the following:

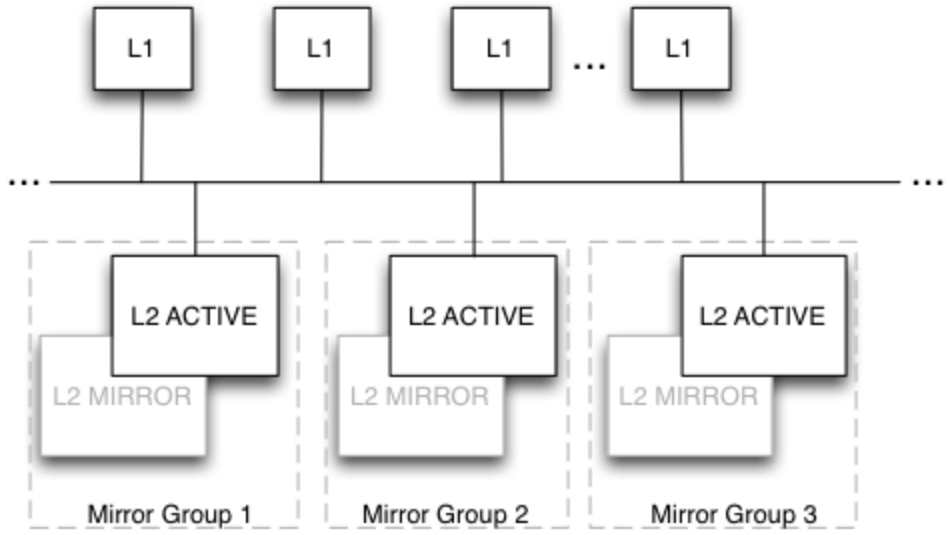
- **Cluster** - All of the Terracotta server instances and clients that work together to share application state or a data set.
- **Terracotta client** - Terracotta clients run on application servers along with the applications being clustered by Terracotta. Clients manage live shared-object graphs.

- **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 mirror (sometimes called "hot standby") 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. *Mirror servers add failover coverage within each mirror group*.
- **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 mirror Terracotta server instance. The active server instance manages and persists the fraction of shared data allotted to its mirror group, while each mirror server in the mirror group replicates (or mirrors) the shared data managed by the active server. *Mirror groups add capacity to the cluster*. The mirror servers are optional but highly recommended for providing failover.
- **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.

Tip: Nomenclature - This documentation 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.

Figure 1 illustrates a Terracotta cluster with three mirror groups. Each mirror group has an active server and a mirror, and manages one third of the shared data in the cluster.

Fig. 1: A Server Array With 3 Mirror Groups



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 mirrors. However, a performance overhead may become evident when adding more mirror servers due to the load placed on the active server by having to synchronize with each mirror.

- 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 cannot 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 mirror server instances can be hot-swapped in an array. In Fig. 1, the L2 MIRROR 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 mirror servers must have the same address (hostname or IP address). If you must swap in a mirror with a different configuration, see "Changing Topology of a Live Cluster" on page 61.

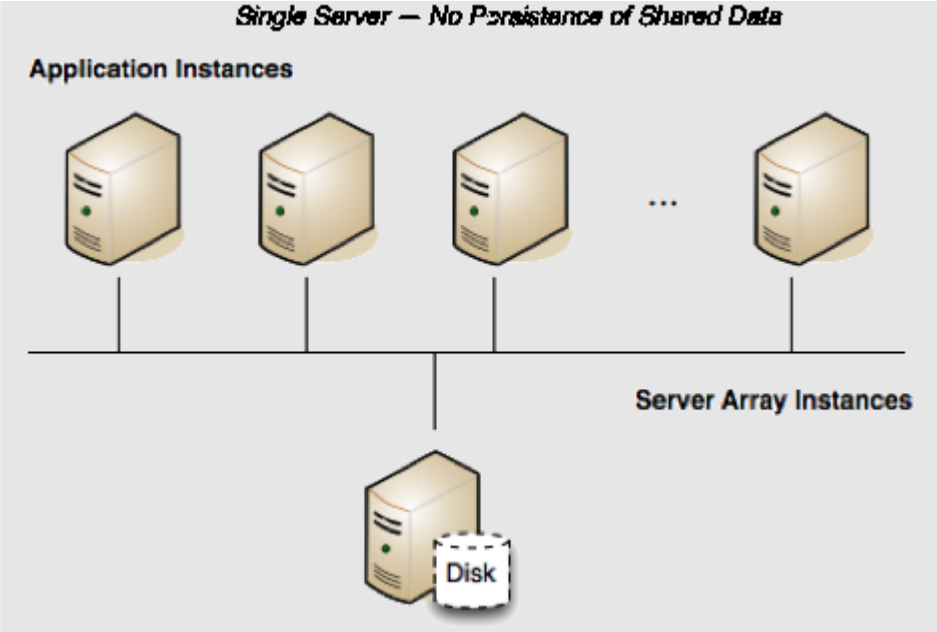
2 Terracotta Server Array Architecture

- Terracotta Cluster in Development 14
- Terracotta Cluster with Reliability 15
- Terracotta Cluster with High Availability 17
- Scaling the Terracotta Server Array 21

Terracotta Cluster in Development

Persistence: No | Failover: No | Scale: No

In a development environment, persisting shared data is often unnecessary and even inconvenient. Running a single-server Terracotta cluster without persistence is a good solution for creating an efficient development environment.



By default, a Terracotta server has Fast Restartability disabled, which means it will not persist data after a restart. 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-9.xsd">
 <servers>
   <server name="Server1">
     <data>/opt/terracotta/server1-data</data>
     <tsa-port>9510</tsa-port>
     <jmx-port>9520</jmx-port>
     <tsa-group-port>9530</tsa-group-port>
     <management-port>9540</management-port>
     <dataStorage size="4g">
        <offheap size="4g"/>
     </dataStorage>
   </server>
  </servers>
 ...
</tc:tc-config>
```

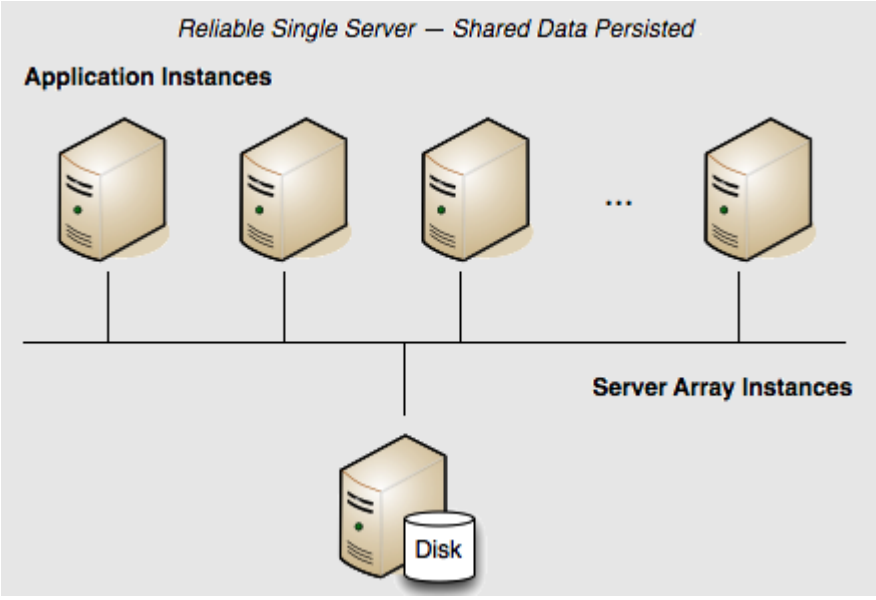
If this server goes down, the application state (all clustered data) in the shared memory is lost. In addition, when the server is up again, all clients must be restarted to rejoin the

cluster. Note that servers are required to run with off-heap, and that shared data is also lost.

Terracotta Cluster with Reliability

Persistence: Yes | Failover: No | Scale: No

The configuration above may be advantageous in development, but if shared in-memory data must be persisted, the server should be configured to use its local disk. Terracotta servers achieve data persistence with the Fast Restart feature.



Fast Restartability

The 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 your BigMemory Max data is still available and very quickly accessible.

The Fast Restart feature persists the real-time record of the in-memory data in a Fast Restart store on the server's 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. In addition, previously connected clients are allowed to rejoin the cluster within a window set by the `<client-reconnect-window>` element.

To configure the Terracotta server for Fast Restartability, add and enable the `<restartable>` element in the tc-config.xml.

```
<?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-9.xsd">
 <servers>
   <server name="Server1">
     <data>/opt/terracotta/server1-data</data>
```

```
    <tsa-port>9510</tsa-port>
    <jmx-port>9520</jmx-port>
    <tsa-group-port>9530</tsa-group-port>
    <management-port>9540</management-port>
    <dataStorage size="4g">
       <offheap size="4g"/>
    </dataStorage>
  </server>
  <!-- Fast Restartability must be added explicitly. -->
  <restartable enabled="true"/>
  <!-- By default the window is 120 seconds. -->
  <client-reconnect-window>120</client-reconnect-window>
 </servers>
 ...
</tc:tc-config>
```

Terracotta server memory allocation

Refer to the following table for store size guidelines for servers in the TSA.

When Off-heap is set between	Configure at least this much Heap
4 - 10 GB	1 GB (Note: The default heap size is 2 GB.)
10 - 100 GB	2 GB
100 GB - 1 TB +	3 GB +
(Off-heap is configured in the tc-config.xml)	(Heap is configured using the -Xmx Java option)

Disk usage

Fast Restartability requires a unique and explicitly specified path. The default path is the Terracotta server's home directory. You can customize the path using the `<data>` element in the server's tc-config.xml configuration file.

The Terracotta Server Array can be configured to be restartable in addition to including searchable caches, but both of these features require disk storage. When both are enabled, be sure that enough disk space is available. Depending upon the number of searchable attributes, the amount of disk storage required might be 3 times the amount of in-memory data.

It is highly recommended to store the search index (`<index>`) and the Fast Restart data (`<data>`) on separate disks.

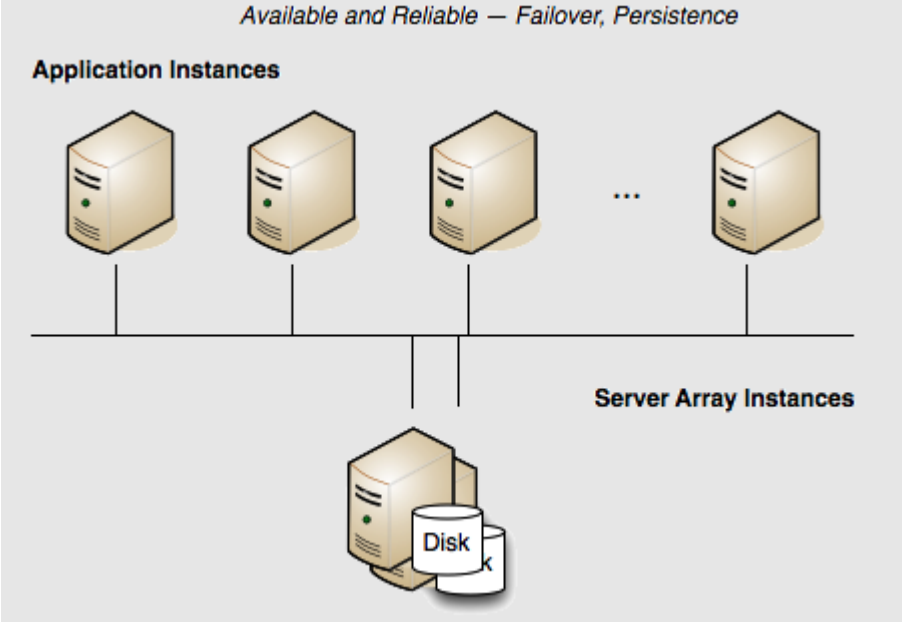
Client Reconnect Window

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 restartable mode is enabled.

Terracotta Cluster with High Availability

Persistence: Yes | Failover: Yes | Scale: No

The example above presents a reliable but *not* highly available cluster. If the server fails, the cluster fails. There is no redundancy to provide failover. Adding a mirror server adds availability because the mirror serves as a "hot standby" ready to take over for the active server in case of a failure.



In this array, if the active Terracotta server instance fails, then the mirror 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-9.xsd">
 <servers>
   <server name="Server1">
     <data>/opt/terracotta/server1-data</data>
     <tsa-port>9510</tsa-port>
     <jmx-port>9520</jmx-port>
     <tsa-group-port>9530</tsa-group-port>
     <management-port>9540</management-port>
     <dataStorage size="4g">
        <offheap size="4g"/>
     </dataStorage>
   </server>
   <server name="Server2">
     <data>/opt/terracotta/server2-data</data>
     <tsa-port>9510</tsa-port>
     <jmx-port>9520</jmx-port>
```

```
    <tsa-group-port>9530</tsa-group-port>
    <management-port>9540</management-port>
    <dataStorage size="4g">
       <offheap size="4g"/>
    </dataStorage>
  </server>
  <restartable enabled="true"/>
  <client-reconnect-window>120</client-reconnect-window>
 </servers>
 ...
</tc:tc-config>
```

You can add more mirror servers to this configuration by adding more `<server>` sections. However, a performance overhead may become evident when adding more mirror servers due to the load placed on the active server by having to synchronize with each mirror.

Note: Terracotta server instances must not share data directories. Each server's `<data>` element should point to a different and preferably local data directory.

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 mirrors (PASSIVE-STANDBY status). The election is recorded in the servers' logs.

If a server instance is started while an active server instance is already running, it syncs up state from the active server instance before becoming a mirror. The active and mirror servers must always be synchronized, allowing the mirror server to mirror the state of the active. The mirror server goes through the following states:

1. **PASSIVE-UNINITIALIZED** - The mirror 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 Management Console (TMC) switches from red to orange.
2. **INITIALIZING** - The mirror 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 TMC is orange.
3. **PASSIVE-STANDBY** - The mirror is synchronized and is ready to perform failover should the active server fail or be shut down. The server's status light in the TMC switches from orange to cyan.

The active server instance carries the load of sending state to the mirror 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 mirrors should be run on similarly configured machines for better throughput, and should be started together to avoid unnecessary sync ups.

The sequence in which servers startup does not affect data. Even if a former mirror server is initialized before the former active server, the mirror server's data is not erased. In the event that a mirror server went offline while the active server was still up, then when the mirror server returns, it remembers that it was in the mirror role. Even if the active server is offline at that point, the mirror server does not try to become the active. It waits until the active server returns, and clients are blocked from updating their data. When the active returns, it will restart the mirror. The mirror's data objects and indices are then moved to the dirty-objectdb-backup directory, and the active syncs its data with the mirror.

Failover

If the active server instance fails and two or more mirror server instances are available, an election determines the new active. Successful failover to a new active takes place only if at least one mirror 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 mirror is available can result in a cluster-wide failure.

If the `dataStorage` and/or `offheap` size on the mirror server is smaller than on the active server, then the mirror server will fail to start and the user will be alerted that the configuration is invalid. If there are multiple mirrors with differing amounts of storage configured, then the passive with the smallest `dataStorage` and `offheap` sizes (that are still greater than or equal to the active's `dataStorage` and `offheap` sizes) will be elected to be the new active.

Tip: Hot-Swapping Mirrors - A mirror can be hot-swapped if the replacement matches the original mirror's <server> block in the Terracotta configuration. For example, the new mirror should use the same host name or IP address configured for the original mirror. For information about swapping in a mirror with a different configuration, refer to "Changing Topology of a Live Cluster" on page 61.

Terracotta server instances acting as mirrors can run either in restartable mode or non-persistent mode. If a server instance running in restartable mode goes down, and a mirror takes over, the crashed server's data directory is cleared before it is 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.

If both servers are down, and clustered data is persisted, the last server to be active will automatically be started first to avoid errors and data loss.

In setups where data is not persisted, meaning that restartable mode is not enabled, then no data is saved and either server can be started first.

Note: Under certain circumstances pertaining to server restarts, the data directory should be manually cleared. For more information, refer to "Clearing Data from a Terracotta Server" on page 59

A Safe Failover Procedure

To safely migrate clients to a mirror server without stopping the cluster, follow these steps:

1. If it is not already running, start the mirror server using the start-tc-server script. The mirror server must already be configured in the Terracotta configuration file.
2. Ensure that the mirror server is ready for failover (PASSIVE-STANDBY status). In the TMC, the status light will be cyan.
3. Shut down the active server using the stop-tc-server script.

   Note: If the 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.

   Clients will connect to the new active server.
4. Restart any clients that fail to reconnect to the new active server within the configured reconnection window.

The previously active server can now rejoin the cluster as a mirror server. If restartable mode had been enabled, its data is first removed and then the current data is read in from the now active server.

A Safe Cluster Shutdown Procedure

A safe cluster shutdown should follow these steps:

1. Shut down the mirror servers 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, any of the servers could be started first as no data 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 mirror servers, causing the mirror 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 mirror.
- A minority of clients connect to the new active server - The original active server starts a reconnect timeout for the clients that it loses, while zapping the new active server. The new active restarts, wipes its database, and synchronizes again as a mirror. 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 mirror. 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.

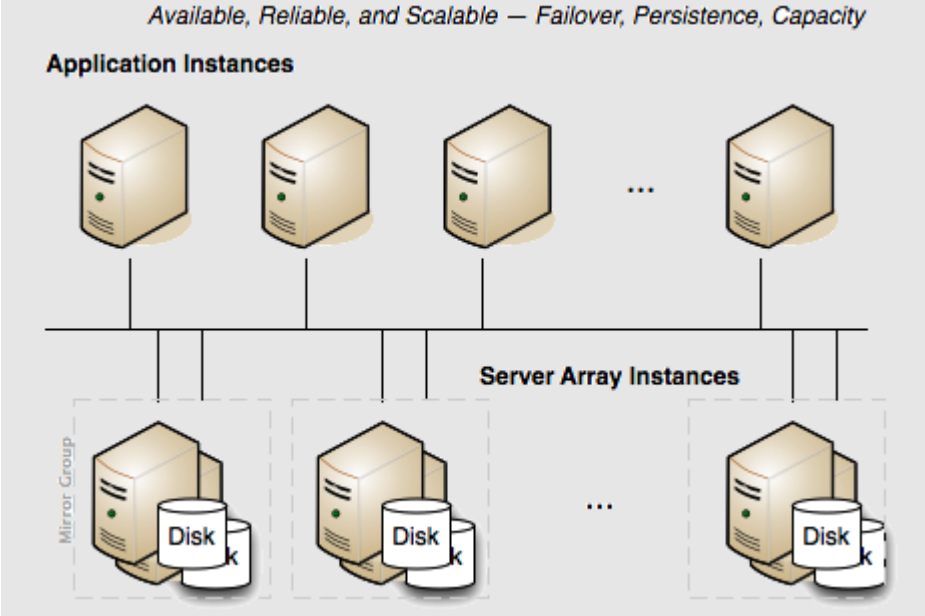
In the case of split-brain occurrences it is imperative to confirm the integrity of shared data after such an event.

Note: Under certain circumstances pertaining to server restarts, the data directory should be manually cleared. For more information, refer to "Clearing Data from a Terracotta Server" on page 59.

Scaling the Terracotta Server Array

Persistence: Yes | Failover: Yes | Scale: Yes

For capacity requirements that exceed the capabilities of a two-server active-mirror setup, expand the Terracotta cluster using a mirror-groups configuration. 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>
  <mirror-group election-time="10" group-name="groupA">
    <server name="server1">
    ...
    </server>
    <server name="server2">
    ...
    </server>
  </mirror-group>
  <mirror-group election-time="15" group-name="groupB">
    <server name="server3">
    ...
    </server>
    <server name="server4">
    ...
    </server>
  </mirror-group>
  <restartable enabled="true"/>
</servers>
...
```

In this example, the cluster is configured to have two active servers, each with its own mirror. If server1 is elected active in groupA, server2 becomes its mirror. If server3 is elected active in groupB, server4 becomes its mirror. server1 and server3 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

mirrors 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.

Note: Server vs. Mirror Group - Under `<servers>`, you may use either `<server>` or `<mirror-group>` configurations, but not both. All `<server>` configurations directly under `<servers>` work together as one mirror group, with one active server and the rest mirrors. To create more than one stripe, use `<mirror-group>` configurations directly under `<servers>`. The mirror group configurations then include one or more `<server>` configurations.

In a Terracotta cluster with mirror groups, each group, or "stripe", behaves in a similar way to an active-mirror setup (see "Terracotta Cluster with High Availability " on page 17. 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 mirror. A mirror cannot become an active server instance during a failure until it is fully synchronized. If an active server instance running in restartable mode goes down, and a mirror takes over, the data directory is cleared before bringing back the crashed server.

Election Time

The `<mirror-group>` configuration allows you to declare the election time window. 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 the work load of the servers should be taken into consideration when choosing an appropriate window.

In the above example, the servers in groupA can take up to 10 seconds to elect an active server, and the servers in groupB can take up to 15 seconds.

Stripe and Cluster Failure

If the active server in a mirror group fails or is taken down, the cluster stops until a mirror 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 the election-time setting), the entire cluster must be restarted.

3 Configuring the Terracotta Server Array

- About Terracotta Server Configuration 26
- How Terracotta Servers Get Configured 26
- How Terracotta Clients Get Configured 28
- Configuration in a Development Environment 30
- Configuration in a Production Environment 32
- Binding Ports to Interfaces 34
- Which Configuration? 35

About Terracotta Server Configuration

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 BigMemory Max 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 Array Architecture" on page 13.

For a comprehensive and fully annotated configuration file, see config-samples/tc-config-reference.xml in the Terracotta kit.

How Terracotta Servers Get Configured

To configure the Terracotta server, create a tc-config.xml configuration file, or update the one that is provided in the config-samples/ directory of the BigMemory Max kit. For example:

```
<?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-9.xsd">
  <servers>
    <server host="localhost" name="My Server Name1">
      <!-- Specify the path where the server should store its data. -->
      <data>/local/disk/path/to/terracotta/server1-data</data>
       <!-- Specify the port where the server should listen for client
       traffic. -->
       <tsa-port>9510</tsa-port>
       <jmx-port>9520</jmx-port>
       <tsa-group-port>9530</tsa-group-port>
       <management-port>9540</management-port>
       <!-- Enable BigMemory on the server. -->
       <dataStorage size="800g">
          <offheap size="200g"/>
          <!-- Hybrid storage is optional. -->
          <hybrid/>
       </dataStorage>
     </server>
    <server host="localhost" name="My Server Name2">
      <data>/local/disk/path/to/terracotta/server2-data</data>
       <tsa-port>9510</tsa-port>
```

```
          <jmx-port>9520</jmx-port>
          <tsa-group-port>9530</tsa-group-port>
          <management-port>9540</management-port>
          <dataStorage size="200g">
             <offheap size="200g"/>
          </dataStorage>
        </server>
       <!-- Add the restartable element for Fast Restartability (optional). -->
       <restartable enabled="true"/>
     </servers>
     <clients>
       <logs>logs-%i</logs>
     </clients>
   </tc:tc-config>
```

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 the Terracotta configuration file.
- <tsa-port> is the port that the Terracotta server listens to for client traffic.
- <jmx-port> is the port that the Terracotta server's JMX Connector listens to.

  Note: Listening on the <jmx-port> is deprecated. Alternatively, use the monitoring features provided by the Terracotta Management Console (see the *Terracotta Management Console User Guide*) and the WAN Replication Service (see the *WAN Replication User Guide*).

  <jmx-port> is disabled by default. If you want to enable it, add `jmx-enabled="true"` in the <server> elements. For example:

  ```
  <server host="localhost" name="My Server Name1" jmx-enabled="true">
  ```

- <tsa-group-port> is the port used by the Terracotta server to communicate with other Terracotta servers.
- <management-port> is the port that the Terracotta Management Console (TMC) uses.
- Under <servers>, use either <server> or <mirror-group> configurations, but not a mixture. You may configure multiple servers or multiple mirror groups. <server> instances under <servers> work together as a mirror group. To create more than one stripe, use <mirror-group> instances.
- Terracotta server instances must not share data directories. Each server's <data> element should point to a different and preferably local data directory.
- For data persistence, configure fast restartability. Enabling <restartable> means that the shared in-memory data is backed up and, in case of failure, it is automatically restored. Setting <restartable> to "false" or omitting the <restartable> element are two ways to configure no persistence.
- Each server requires an off-heap store, which allows all data to be stored in-memory, limited only by the amount of memory in your server. The minimum <offheap> size is 4 GB. Additional hybrid storage is optional. Specify the <dataStorage> size and the

<offheap> size. The <offheap> size can be set to the amount of memory available in your server for data. If you enable <hybrid>, then the <dataStorage> size can exceed the <offheap> size.

- All servers and clients should be running the same version of Terracotta and Java.

Note: For more information about the Terracotta configuration file, see "Terracotta Configuration Parameters" on page 101.

Server Startup Behavior

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>
```

where <*path_to_configuration_file*> can be a URL or a relative directory path. In Microsoft Windows, use start-tc-server.bat.

Note: Cygwin (on Windows) is not supported for this feature.

How Terracotta Clients Get Configured

At startup, Terracotta clients load their configuration from one of the following sources:

- "Local or Remote XML File" on page 29
- "Terracotta Server" on page 29
- An Ehcache configuration file (using the "<terracottaConfig> element" on page 33) used with BigMemory Max and BigMemory Go.

- A Quartz properties file (using the `org.quartz.jobStore.tcConfigUrl` property) used with Quartz Scheduler.
- A Filter (in web.xml) element used with containers and Terracotta Web Sessions.
- The client constructor (TerracottaClient()) used when a client is instantiated programmatically.

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. However, there are options you can set for server settings to override client settings. For details, see "How Server Settings Can Override Client Settings" in the *BigMemory Max Configuration Guide*.

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 the 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" on page 26.

To specify a configuration file for a Terracotta client, see "Clients in Development " on page 32.

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

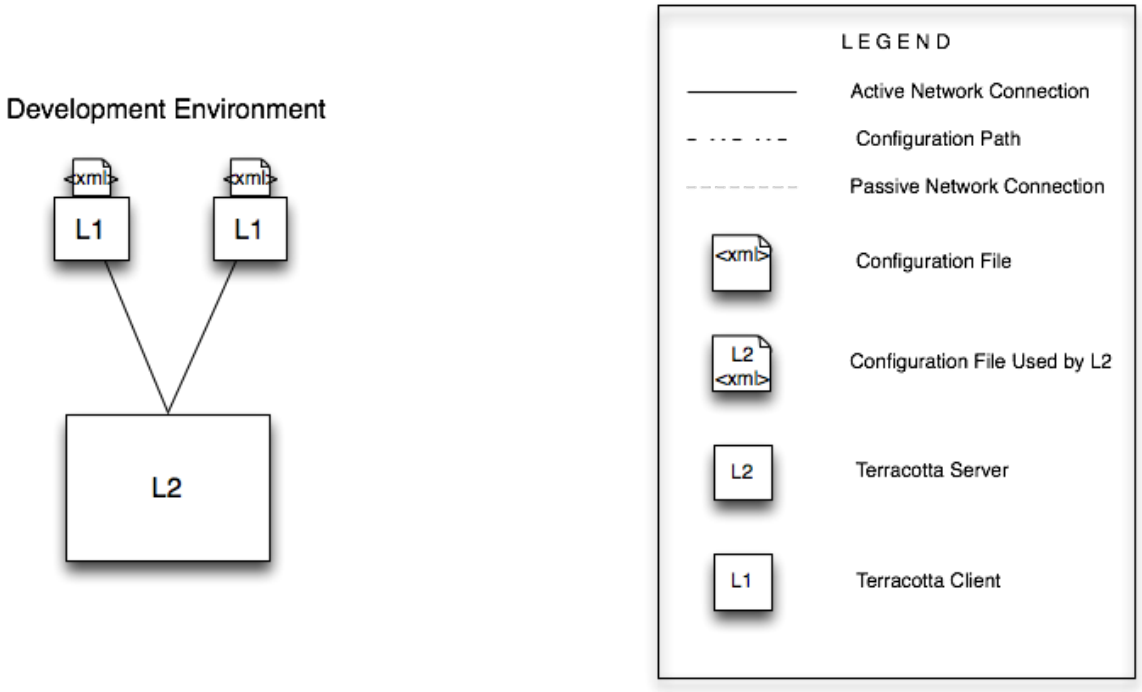
Terracotta clients can load configuration from an active Terracotta server by specifying its hostname and TSA port (see "Clients in Production" on page 33).

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.

One-Server Setup in Development

For one Terracotta server, the default configuration is adequate.



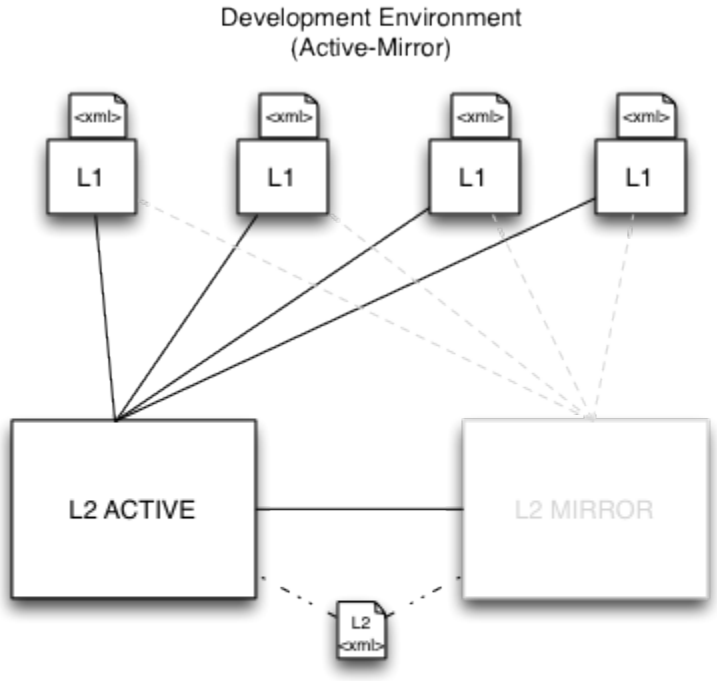
To use the default configuration settings, start your Terracotta server using the start-tc-server.sh (or start-tc-server.bat) script in a directory that does *not* contain the file tc-config.xml :

```
[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" on page 26.

Two-Server Setup in Development

A two-server setup, sometimes referred to as an active-mirror setup, has one active server instance and one "hot standby" (the mirror) 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-9.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 mirror and maintain the environment in case of failover. When running both servers on the same host, the <tsa-port> for each server should be different. The other ports are automatically generated from the <tsa-port>. Having a separate log location is a good idea, but failing to do so will not prevent the servers from starting. If the servers are set up to be restartable, setting the data directory to a different location would be a requirement.

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.

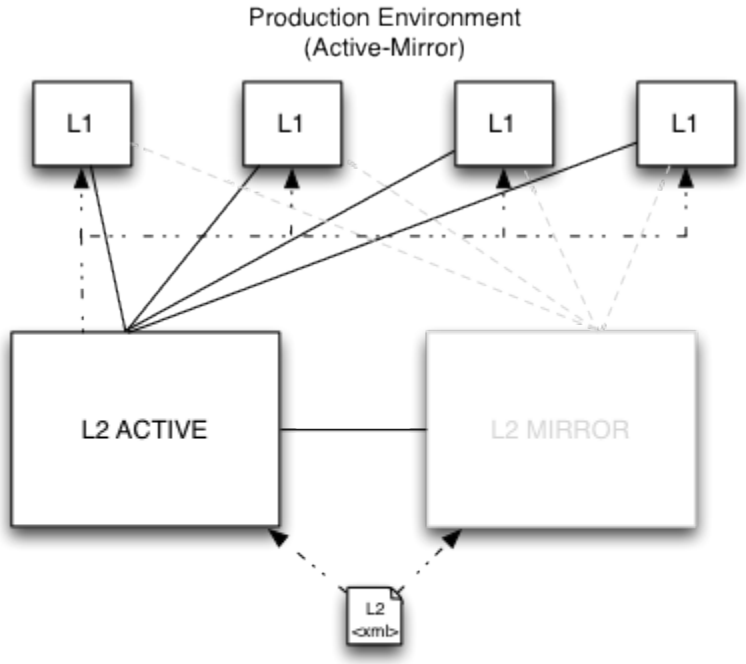
```
-Dtc.config=path/to/my-tc-config.xml -cp classes myApp.class.Main
```

where `myApp.class.Main` is the class used to launch the application you want to cluster with Terracotta.

If tc-config.xml exists in the directory in which you run Java, it can be loaded without `-Dtc.config`.

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.

Setting Up the Terracotta Environment

To start your application with the Terracotta client using your own scripts, first set the following environment variables:

```
TC_INSTALL_DIR=<path_to_local_Terracotta_home>
TC_CONFIG_PATH=<path/to/tc-config.xml>
```

or

```
TC_CONFIG_PATH=<server_host>:<tsa-port>
```

where `<server_host>:<tsa-port>` points to the running Terracotta server. The specified Terracotta server will push its configuration to the Terracotta client.

Alternatively, a client can specify that its configuration come from a server by setting the `tc.config` system property:

```
-Dtc.config=serverHost:tsaPort
```

If more than one Terracotta server is available, enter them in a comma-separated list:

```
TC_CONFIG_PATH=<server_host1>:<tsa-port>,<server_host2>:<tsa-port>
```

If <server_host1> is unavailable, <server_host2> is used.

Terracotta Products

Terracotta products can set a configuration path using their own configuration files.

For BigMemory Max and BigMemory Go, 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 the *Web Sessions User Guide*).

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">
 ...
 <tsa-port>9510</tsa-port>
 <jmx-port>9520</jmx-port>
 <tsa-group-port>9530</tsa-group-port>
 <management-port>9540</management-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">
 ...
 <tsa-port>9510</tsa-port>
 <jmx-port bind="12.345.678.9">9520</jmx-port>
 <tsa-group-port>9530</tsa-group-port>
 <management-port>9540</management-port>
</server>
```

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".

4 Automatic Resource Management

- What is Automatic Resource Management? 38
- Eviction 38
- Customizing the Eviction Strategy 42

What is Automatic Resource Management?

Terracotta Server Array resource management involves self-monitoring and polling to determine the real-time size of the data set and assess the amount of memory remaining according to user-configured limitations. In-memory data can be managed from three directions:

1. **Time** - TTI/TTL settings can be configured to expire entries that will then be evicted by the new TSA eviction implementation. You can also configure caches so that their entries are eternal, or you can pin entries or caches so that they are never evicted.
2. **Size** - The total amount of BigMemory managed by the TSA can be configured using the `dataStorage` and `offheap` elements in the tc-config.xml file.
3. **Count** - The total number of entries per cache can be configured using the maxEntriesInCache attribute in the ehcache.xml file.

Eviction

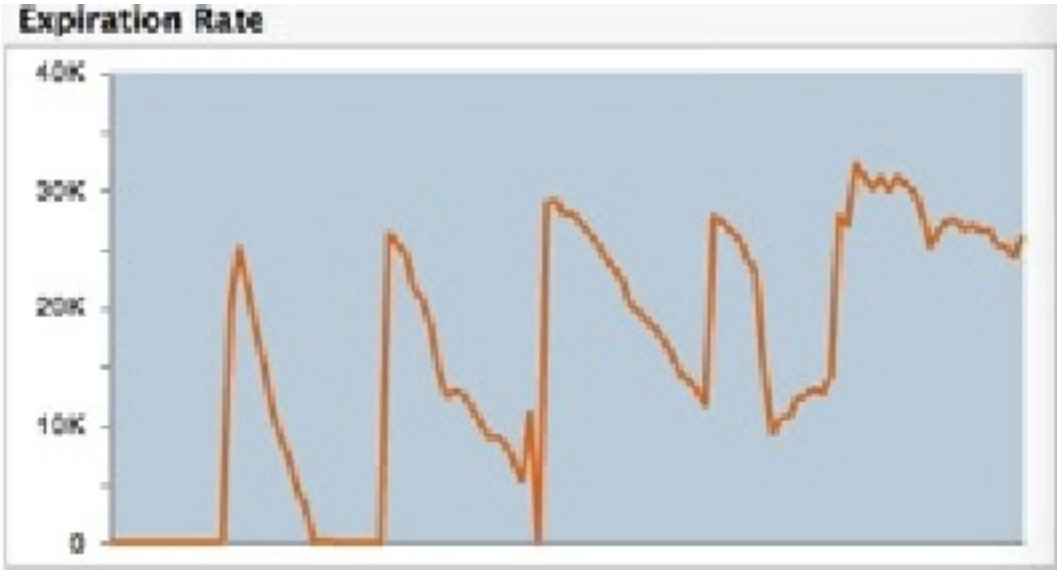
All data is kept in memory, and the TSA runs evictions in the background to keep the data set within its limitations. Eviction of entries from the data set reduces the amount of data before the memory becomes full. The criteria for an entry to be eligible for eviction are:

- It is not on a Terracotta client (L1).
- It is not pinned to a Terracotta server.
- It is held in a cache backed by a System of Record (SOR).

Note: Store vs. Cache - BigMemory's in-memory data is treated as a "store" when BigMemory owns the data, and as a "cache" when the data also resides in a System of Record (SOR). Generally, data that is created by BigMemory and run-time data created by your application are examples of data that is treated as a store. The TSA does not evict data stores because they are the only or primary records. The TSA can evict cached data because that data is backed up in an SOR. Distinctions in data structures are handled automatically.

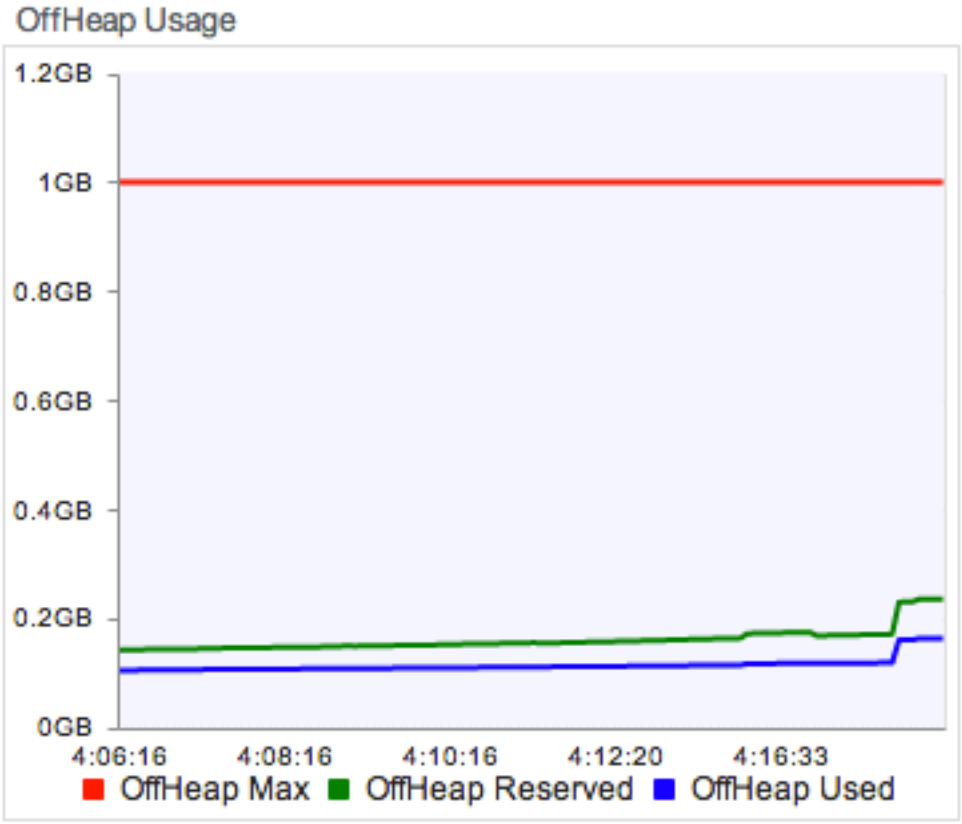
Eviction is done by the following evictors, which work together in the background:

1. The periodic evictor is activated on an as-needed basis. It removes expired entries based on TTI/TTL settings. The Server Expiration Rate graph in the TMC shows the activity of the periodic evictor.



2. The resource-based evictor is activated by the periodic TTI/TTL eviction scheduler, as well as by resource monitoring events. This evictor continuously polls BigMemory stores to check current resource usage. At approximately 10% usage of the disk as well as `dataStorage` and `offheap` sizes (configured in the tc-config.xml file), it starts looking for TTI/TTL-expired elements to evict. At approximately 80% usage, it evicts live as well as expired elements. If a monitored resource goes over its critical threshold, this evictor will work continually until the monitored resource falls below the critical threshold.

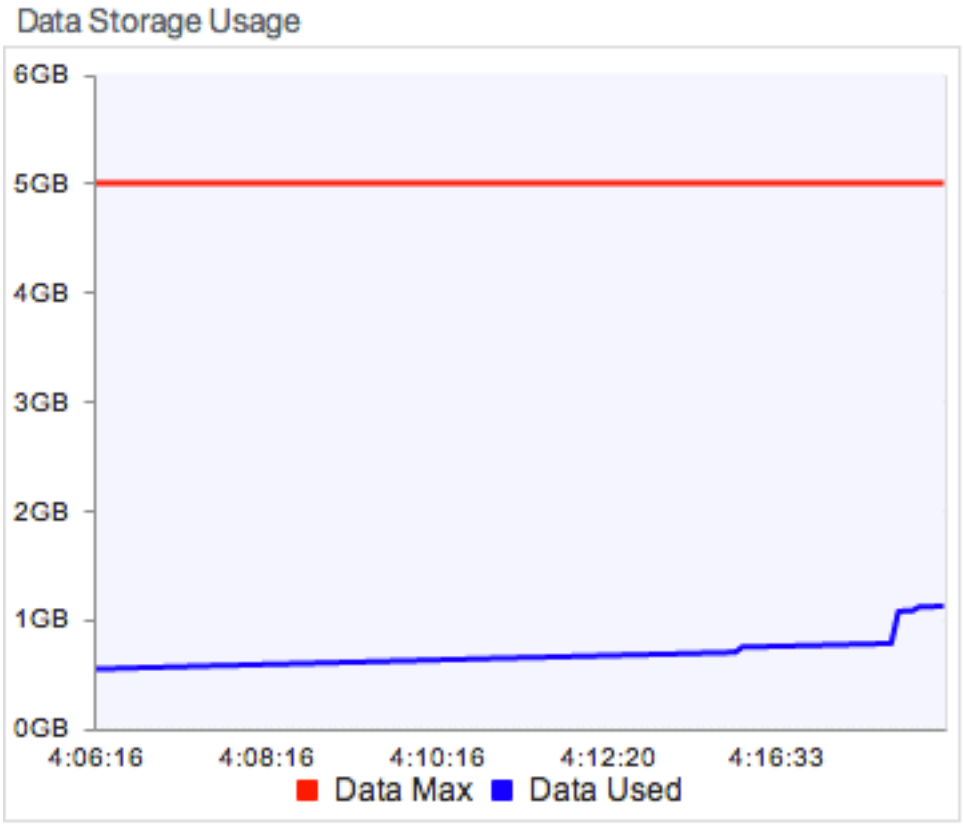
   This evictor monitors two off-heap thresholds -- used space and reserved space. Resource eviction is triggered if either the reserved or used space is above its threshold. Once resource eviction has started, both used and reserved spaces must fall below their respective thresholds before resource eviction ends.



The Offheap Usage graph in the TMC provides the following information:

- Off-heap Max is the configured `offheap` size
- Off-heap Reserved represents usage of the space that is reserved for the system
- Off-heap Used represents the amount of off-heap BigMemory that is in use

The Data Storage Usage graph in the TMC shows the configured `dataStorage` size and the amount in use. This usage includes off-heap and hybrid storage combined. (BigMemory Hybrid allows for a mix of a solid-state drive (SSD) with DRAM-based offheap storage.)



3. The capacity-based evictor is activated when a cache goes over its maximum count (as configured with `maxEntriesInCache`), plus an overshoot count, and it attempts to bring the size of the cache to the max capacity. The `maxEntriesInCache` attribute must be present in the Ehcache configuration (do not include `maxEntriesInCache` in your configuration if you do not want the capacity evictor to run). If `maxEntriesInCache` is not set, it gets the default value 0, which means that the cache is unbounded and will not undergo capacity eviction (but periodic and resource evictions are still allowed).

   The Server Eviction Rate graph in the TMC shows the activity of the resource and capacity evictors.

Customizing the Eviction Strategy

Based upon the three types of evictors, there are three strategies that you can employ for controlling the size of the TSA's data set:

1. Set the Time To Idle (TTI) or Time to Live (TTL) options for any entry in your data set. After the time has expired, the periodic evictor will clear the entry.
2. Set the `dataStorage` and `offheap` sizes to control how much BigMemory should be used before the resource-based evictor is activated.
3. Set the `maxEntriesInCache` attribute to control when the capacity-based evictor is activated.

5 Managing Near-Memory-Full Conditions

- Behavior of the TSA under Near-Memory-Full Conditions 44
- Restricted Mode Operations 45
- Recovery 45

Behavior of the TSA under Near-Memory-Full Conditions

In a near-memory-full condition, where evictions are not happening fast enough to keep the data set within its BigMemory size limitations, the TSA will put a throttle on operations for a temporary period while it attempts to automatically recover in the background. If unable to recover, the TSA will move into "restricted mode" to prevent out-of-memory errors. The Terracotta Management Console (TMC) uses events to report when the TSA enters restricted mode and allows you to execute additional recovery measures. See "Monitoring Cluster Events" on page 47.

Summary of TSA behavior in near-memory-full conditions:

- If usage reaches its critical threshold, T1, then it enters "throttle mode," where writes are slowed while the TSA attempts to evict eligible cache entries in order to bring memory usage within the configured range.
- If usage reaches its halt threshold, T2, then it enters "restricted mode," where writes are blocked, an exception is thrown, and operator intervention is needed to reduce memory usage.
- When usage falls below T1, then the TSA returns to normal operation.

	Throttle mode	Restricted mode
Entered when	Disk, `dataStorage` , and/or `offheap` usage crosses its critical threshold	Disk, `dataStorage` , and/or `offheap` usage crosses its halt threshold
Operator event	"TPS seems really low; marking us as being throttled'	"We're in restricted mode; waiting a while and retrying"
Data access	Modifications to in-memory data are slowed	Modifications to in-memory data are blocked
Allowed operations	All cache operations still allowed	Only gets, removes, and config changes are allowed
Actions	Evictions continue automatically in the background	Operator intervention required to make additional evictions
State of the data	Evictable data is still present	No more data present in memory that can be evicted

Throttle mode		Restricted mode
		by the evictor (all caches are pinned)
Recovery	Automatic	From the TMC or programmatically, clear caches and/or remove entries from a data set
Back to normal operation	As soon as the background evictions have time to catch up and reduce the data set to within its limitations	After user intervention clears space, the TSA will automatically continue with normal operation

Restricted Mode Operations

If the TSA is temporarily under restricted mode, any change to the data set which may result in increased resource utilization is not allowed, including all put and replace methods. Restricted mode does allow gets, removes, configuration changes, and other operations.

Recovery

Recovery from throttle mode is automatic, as soon as the background evictions have time to reduce the data set to within its limitations.

If the TSA enters restricted mode, operator events will be logged in the TMC, and user or programmatic intervention is necessary. In the TMC, you can initiate actions to manually reduce the data set. You can also anticipate operator events and use programmatic logic to respond appropriately.

The following actions are recommended for reducing the data set:

- Clear caches (from the TMC or programmatically)
- Remove entries from data sets programmatically

Note: Because eviction in restricted mode is resource-driven, changing TTI/TTL or maximum capacity will not move the TSA out of restricted mode.

To clear caches from the TMC, click the **Application Data** tab and the **Management** sub-tab. Each cache will have a clickable option to **Clear Cache**. Note that caution should be used when considering whether to clear a pinned cache.

6 Monitoring Cluster Events

- About Cluster Events 48
- Event Types and Definitions 48

About Cluster Events

Cluster events report topology changes, performance issues, and errors in operations. These events are logged by both Terracotta server (L2) and client (L1), and can also be viewed in the TMC (see the *Terracotta Management Console User's Guide*).

By default, the L2 stores a maximum of 100 events in memory, and this is the number pulled by the TMC. To edit that number, use the Terracotta property `l2.operator.events.store`. To set the property in the Terracotta configuration file, use:

```
<tc-properties>
  ...
  <property name="l2.operator.events.store" value="500" />
</tc-properties>
```

Event Types and Definitions

This section describes the types of events that can be found in logs or viewed in the TMC.

- **memory.longgc (Memory Manager)**
  - Level: WARN
  - Cause: A full garbage collection (GC) longer than the configured threshold has occurred.
  - Action: Reduce cache memory footprint in L1 (Terracotta client). Investigate issues with application logic and garbage creation.
  - Notes: The default critical threshold is 8 seconds, but it can be reconfigured in tc.properties using `longgc.threshhold`. For information about setting tc.properties, see the "Terracotta Configuration Parameters" on page 101.

    Occurrence of this event could help diagnose certain failures. For details, see "Configuring the HealthChecker Properties" in the *BigMemory Max High-Availability Guide*.
- **dgc.periodic.started (DGC)**
  - Level: INFO
  - Cause: Periodic distributed garbage collection (DGC), which was explicitly enabled in the configuration, has started a cleanup cycle.
  - Action: If periodic DGC is unneeded, disable it to improve overall cluster performance.
  - Notes: Periodic DGC, which is disabled by default, is mostly useful in the absence of automatic handling of distributed garbage.

- **dgc.periodic.finished (DGC)**
  - Level: INFO
  - Cause: Periodic DGC, which was explicitly enabled in the configuration, ended a cleanup cycle.
  - Action: If periodic DGC is unneeded, disable it to improve overall cluster performance.
  - Notes: Event message reads "DGC[{0}] finished. Begin Count : {1} Collected : {2} Time Taken : {3} ms Live Objects : {4}".
- **dgc.periodic.canceled (DGC)**
  - Level: INFO
  - Cause: Periodic DGC, wPeriodichich was explicitly enabled in the configuration, has been cancelled due to an interruption (for example, by a failover operation).
  - Action: If periodic DGC is unneeded, disable it to improve overall cluster performance.
  - Notes: Periodic DGC, which is disabled by default, is mostly useful in the absence of automatic handling of distributed garbage.
- **dgc.inline.cleanup.started (DGC)**
  - Level: INFO
  - Cause: L2 (Terracotta server) is starting up as ACTIVE with existing data, triggering inline distributed garbage collection (DGC).
  - Action: No action necessary.
  - Notes: Only seen when a server starts up as ACTIVE upon a recovery, using Fast Restartability.
- **dgc.inline.cleanup.finished (DGC)**
  - Level: INFO
  - Cause: Inline DGC operation completed.
  - Action: No action necessary.
  - Notes: Event message reads "Inline DGC [{0}] reference cleanup finished. Begin Count : {1} Collected : {2} Time Taken : {3} ms Live Objects : {4}".
- **dgc.inline.cleanup.canceled (DGC)**
  - Level: INFO
  - Cause: Inline DGC operation interrupted.
  - Action: Investigate any unusual cluster behavior or other events.
  - Notes: Possibly occurs during failover, but other events should indicate real cause.

- **topology.node.joined (Cluster Topology)**
  - Level: INFO
  - Cause: Specified node has joined the cluster.
  - Action: No action necessary.
  - Notes: None.
- **topology.node.left (Cluster Topology)**
  - Level: WARN
  - Cause: Specified node has left the cluster.
  - Action: Check why the node has left (for example: long GC, network issues, or issues with local node resources).
  - Notes: None.
- **topology.node.state (Cluster Topology)**
  - Level: INFO
  - Cause: L2 changing state (for example, from INITIALIZING to ACTIVE).
  - Action: Check to see that the state change is expected.
  - Notes: Event message reads "Moved to {0}", where {0} is the new state.
- **topology.handshake.reject (Cluster Topology)**
  - Level: ERROR
  - Cause: L1 is unsuccessfully trying to reconnect to cluster, but it has already been expelled.
  - Action: If the L1 does not go into a rejoin operation, it must be restarted manually.
  - Notes: Event message reads "An {0} client {1} tried to connect to {2} server. Connection refused!!"
- **topology.active.left (Cluster Topology)**
  - Level: WARN
  - Cause: Active server left the cluster.
  - Action: Check why the active L2 has left.
  - Notes: None.
- **topology.mirror.left (Cluster Topology)**
  - Level: WARN
  - Cause: Mirror server left the cluster.
  - Action: Check why the mirror L2 has left.

- Notes: None.

- **topology.zap.received (Cluster Topology)**
  - Level: CRITICAL
  - Cause: One L2 is trying to cause another L2 to restart ("zap").
  - Action: Investigate a possible "split brain" situation (a mirror L2 behaves as the ACTIVE) if the zapped L2 does not obey the restart order.
  - Notes: A "zap" operation happens only within a mirror group. Event message reads "SPLIT BRAIN, {0} and {1} are ACTIVE", where {0} and {1} are the two servers vying for the ACTIVE role.
- **topology.zap.accepted (Cluster Topology)**
  - Level: CRITICAL
  - Cause: The L2 is accepting the order to restart ("zap" order).
  - Action: Check the state of the zapped L2 to ensure that it restarts as a mirror, or manually restart it.
  - Notes: A "zap" order is issued only within a mirror group. Event message reads "{0} has more clients. Exiting!!", where {0} is the L2 that becomes the ACTIVE.
- **topology.db.dirty (Cluster Topology)**
  - Level: WARN
  - Cause: A mirror L2 is trying to join with data in place.
  - Action: If the mirror does not automatically restart and wipe its data, its data may need to be manually wiped and before it is restarted.
  - Notes: Restarted mirror L2s must wipe their data to resync with the active L2. This is normally an automatic operation that should not require action. Event message reads "Started with dirty database. Exiting!! Restart {0}", where {0} is the the mirror that is automatically restarting.
- **topology.config.reloaded (Cluster Topology)**
  - Level: INFO
  - Cause: Cluster configuration was reloaded.
  - Action: No action necessary.
  - Notes: None.
- **dcv2.servermap.eviction (DCV2)**
  - Level: INFO
  - Cause: Automatic evictions for optimizing Terracotta Server Array operations.
  - Action: No action necessary.

- Notes: Event message reads "DCV2 Eviction - Time taken (msecs)={0}, Number of entries evicted={1}, Number of segments over threshold={2}, Total Overshoot={3}".

- **system.time.different (System Setup)**
  - Level: WARN
  - Cause: System clocks are not aligned.
  - Action: Synchronize system clocks.
  - Notes: The default tolerance is 30 seconds, but it can be reconfigured in tc.properties using `time.sync.threshold`. For information about setting tc.properties, see "Terracotta Configuration Parameters" on page 101.

    Note that overly large tolerance can introduce unpredictable errors and behaviors.

- **resource.capacity.near (Resource)**
  - Level: WARN
  - Cause: L2 entered throttled mode, which could be a temporary condition (e.g., caused by bulk-loading) or could indicate insufficient allocation of memory.
  - Action: See "Managing Near-Memory-Full Conditions" on page 43.
  - Notes: After emitting this, L2 can emit `resource.capacityrestored` (return to normal mode) or `resource.fullcapacity` (move to restricted mode), based on resource availability. Event message reads "{0} is nearing capacity limit, performance may be degraded - {1}% usage", where {0} is the L2 identification and {1} is the % usage of the memory resources allocated to that L2.

- **resource.capacity.full (Resource)**
  - Level: ERROR
  - Cause: L2 entered restricted mode, which could be a temporary condition (e.g., caused by bulk-loading) or could indicate insufficient allocation of memory.
  - Action: See "Managing Near-Memory-Full Conditions" on page 43.
  - Notes: After emitting this, L2 can emit `resource.capacityrestored` (return to normal mode), based on resource availability. Event message reads "{0} is at over capacity limit, no further additive operations will be accepted - {1}% usage", where {0} is the L2 identification and {1} is the % usage of the memory resources allocated to that L2.

- **resource.capacity.restored (Resource)**
  - Level: INFO
  - Cause: L2 returned to normal from throttled or restricted mode.
  - Action: No action necessary.

- Notes: Event message reads "{0} capacity has been restored, performance has returned to normal - {1}% usage", where {0} is the L2 identification and {1} is the % usage of the memory resources allocated to that L2.

7 Backing Up Live In-Memory Data

- About Live Backup 56
- Creating a Backup 56
- The Backup Directory 56
- Restoring Data from a Backup 57

About Live Backup

Backups of the entire data set across all stripes (mirror groups) of the Terracotta Server Array can be made using the TMC Backup feature. This feature creates a time-stamped backup of each stripe's data, providing a snapshot of the TSA's in-memory data.

The Backup feature is available when fast restartability is enabled for the TSA (`<restartable enabled="true"/>` in the tc-config.xml).

Creating a Backup

From the TMC, select the **Administration** tab and the **Backups** sub-tab. Click the **Make Backup** button to perform a backup. The TMC sends a backup request to all stripes in the cluster.

In order to capture a consistent snapshot of the in-memory data, the backup function creates a pause in transactions, allowing any unfinished transactions to complete, and then the backup is written. This allows the backup to be a consistent record of the entries in-memory, as well as search and other indices.

Note that when backing up a cluster, each stripe is backed up independently and at a slightly different time than the other stripes.

When complete, a window appears that confirms the backup was taken and provides the time-stamped file name(s) of the backup.

The Backup Directory

Backups are saved to the default directory data-backup, unless otherwise configured in the tc-config.xml. Terracotta automatically creates data-backup in the directory containing the Terracotta server's configuration file (tc-config.xml by default).

You can override the default directory by specifying a different backup directory in the server's configuration file using the `<data-backup>` property:

```
<servers>
   <server name="Server1">
      <data>/opt/terracotta/server1-data</data>
      <data-backup>path/to/my/backup/directory</data-backup>
      <offheap>
         <enabled>true</enabled>
         <maxDataSize>2g</maxDataSize>
      </offheap>
   </server>
   <restartable enabled="true"/>
</servers>
```

Restoring Data from a Backup

If the TSA fails, on restart it automatically restores data from its data directory, recreating the application state. If the current data files are corrupt or missing, or in other situations where an earlier snapshot of data is required, you can restore them from backups:

1. Shut down the Terracotta cluster.
2. (Optional) Make copies of any existing data files.
3. Delete the existing data files from your Terracotta servers.
4. Copy the backup data files to the directory from which you deleted the original (existing) data files.
5. Restart the Terracotta cluster.

8 Clearing Data from a Terracotta Server

- How to Clear Data from a Terracotta Server 60

How to Clear Data from a Terracotta Server

After a Terracotta server is restarted, under certain circumstances it will retain artifacts from previous runs and its data directory must be manually cleared. These circumstances include running with Fast Restartability disabled and BigMemory Hybrid enabled. This may also be the source of errors during "split brain" resolution or during a mirror server restart.

By default, the number of copies of a server's objectdb data that are retained is unlimited. Over time, and with frequent restarts, these copies may consume a substantial amount of disk space. You can manually delete these files, which are saved in the server's data directory under /dirty-objectdb-backup/dirty-objectdb-<timestamp>. You can also set a limit for the number of backups by adding the following element to the Terracotta configuration file's <tc-properties> block:

```
<property name="l2.nha.dirtydb.rolling" value="<myValue>" />
```

where <myValue> is an integer.

If you have Fast Restartability disabled and BigMemory Hybrid enabled, you can also automatically remove the older data before server shutdown by setting the property `l2.nha.dirtydb.autoDelete` to "true". It is important to note that a backup copy will still be made in the configured /dirty-objectdb-backup folder prior to the most recent "working" data being deleted. This can potentially consume all available disk space. If you want to disable the creation of backups altogether, you can set the property `l2.nha.dirtydb.backup.enabled` to "false".

9 Changing Topology of a Live Cluster

- About Changing the Topology 62
- Adding a New Server 62
- Removing an Existing Server 63
- Editing the Configuration of an Existing Server 63

About Changing the Topology

Using the TMC, 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> or <mirror-group> 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> or <mirror-group> 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>
  <tsa-port>9510</tsa-port>
  <management-port>9540</management-port>
</server>
```

2. Make sure you are connected to the TMC, and that the TMC is connected to the target cluster. See the *Terracotta Management Console User Guide* for more information on using the TMC.
3. With the target cluster selected in the TMC, click the **Administration** tab, then choose the panel.
4. Click **Reload**. A message appears with the result of the reload operation. A successful operation logs a message similar to the following:

```
2013-03-14 13:25:44,821 INFO - Successfully overridden server topology
```

```
from file at '/bigmemory-max-4/tc-config.xml'.
```

5. 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 Terracotta configuration file being used by the cluster. Make sure you are connected to the TMC, and that the TMC is connected to the target cluster. See the *Terracotta Management Console User's Guide* for more information on using the TMC.
3. With the target cluster selected in the TMC, click the **Administration** tab, then choose the **Change Topology** panel.
4. Click **Reload**. A message appears with the result of the reload operation. A successful operation logs a message similar to the following:

```
2013-03-14 13:25:44,821 INFO - Successfully overridden server topology
from file at '/bigmemory-max-4/tc-config.xml'.
```

The TMC will also display the event

```
Server topology reloaded from file at '/bigmemory-max-4/tc-config.xml'.
```

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 " on page 63. 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. In the TMC'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. Clients that load configuration from another source will fail to remain connected to the TSA due to a configuration mismatch.

10 Enabling Production Mode

■ Setting the Production Mode Property 66

Setting the Production Mode Property

Production mode can be set by setting the Terracotta property in the Terracotta configuration:

```
<tc-properties>
  ...
  <property name="l2.enable.legacy.production.mode" value="true" />
</tc-properties>
```

Production mode requires the `--force` flag to be used with the stop-tc-server script if the target is an active server with no mirror.

11 Managing Distributed Garbage Collection

- About Distributed Garbage Collection (DGC) 68
- Running the Periodic Distributed Garbage Collection 68
- Monitoring and Troubleshooting DGC 68

About Distributed Garbage Collection (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 Distributed Garbage Collection

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

- run-dgc shell script - Call the run-dgc shell script to trigger DGC externally.
- JMX - Trigger DGC through the server's JMX management interface.

By default, DGC is disabled 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 server returns to the cluster).

Monitoring and Troubleshooting DGC

DGC events (both periodic and inline) are reported in a Terracotta server instance's logs. DGC events can also be monitored using the Terracotta Management Console.

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.

12 Starting the Terracotta Server as a Windows Service

■ Configuring the Terracotta Server to Run as a Service 70

Configuring the Terracotta Server to Run as a Service

A Windows service supports scheduling and automatic start and restart. You might want to run the Terracotta Server Array or the Cross-Language Connector, which are Java applications, as a *Windows service*. If so, use the Service Wrapper located inside the kit at $installdir/server/wrapper (or, for 3.7, $installdir/wrapper).

Set JAVA_HOME

To start the service, set your `JAVA_HOME` in conf/wrapper-tsa.conf or conf/wrapper-clc.conf . For example:

```
set.JAVA_HOME=C:/Java/jdk1.7.0_21
```

The wrapper does not read your `JAVA_HOME` from the environment. For Windows, if you do not want to set it in the configuration file, comment it out and set `JAVA_HOME` in the registry instead.

Configuration Files

For the Cross-Language Connector, you need these configuration files:

- conf/cross-language-config.xml
- conf/ehcache.xml

For the TSA, you need this configuration file:

- conf/tc-config.xml

Overwrite those files with your own. If you want to change these file names, modify the names in the wrapper configurations.

Modify the TSA conf/wrapper-tsa.conf file to match the server name in your tc-config.xml:

```
set.SERVER_NAME=server0
```

where *server0* represents the name of the server you want to start.

Set Permissions

The services are controlled by an Administrator user, so you have to confirm for every action, such as install, start, stop, remove.

In addition, the Administrator user needs to have read/write permission for the "wrapper" directory.

Install and Start the Service

The wrapper service is located at $installdir/server/wrapper.

To install the service wrapper, run the script with the install parameter:

```
%> bin/tsa-service.bat install
```

Note: The examples in this section show the TSA script. For the Cross-Language Connector, use the clc-service or clc-service.bat script.

Then you can either start/stop the service:

```
%> bin/tsa-service.bat start
%> bin/tsa-service.bat stop
```

If you want to remove the service:

```
%> bin/tsa-service.bat remove
```

There are more commands available when you run the script without any parameter:

```
%> bin/tsa-service.bat
```

Changing Wrapper Configuration

There are comments in wrapper-tsa.conf and wrapper-clc.conf to explain each parameter. If you need to modify JVM system properties, classpath, or command line parameters, follow the current pattern. Pay close attention to their numerical order and parameter counts.

For more information, see http://wrapper.tanukisoftware.com/doc/english/properties.html.

13 Using BigMemory Hybrid

- About BigMemory Hybrid 74
- System Requirements 76
- Hardware Capacity Guidelines 76
- Configuring BigMemory Hybrid 76
- Using the TMC with BigMemory Hybrid 77
- Operator Events 78

About BigMemory Hybrid

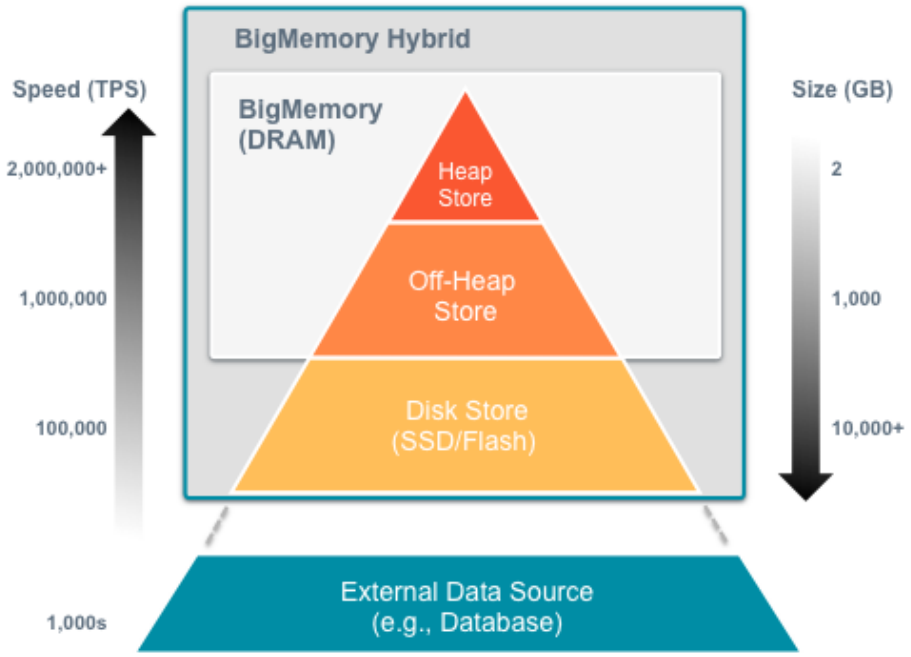
BigMemory Hybrid is an optional extension to BigMemory Max that:

- Enables scaling up to economical solid-state drive (SSD) flash memory motionless "disks" in conjunction with conventional dynamic random-access memory (DRAM) memory. This means that the cache size can exceed the available off-heap memory.
- Provides predictable low latency at very large scale.
- Manages the data flow seamlessly and automatically from DRAM to flash "disk" according to the size of the cache.
- Performs much faster than conventional hard disks, although not quite as fast as a pure DRAM in-memory solution.
- Supports searching, Fast Restart backup and recovery, Web Sessions, Quartz, and WAN replication.
- Works with industry-standard SSD devices from popular vendors, such as Fusion IO and Intel SSD.

How it works

For use in Terracotta servers, when using BigMemory Hybrid, all data is stored in SSD/Flash drives. The keys are stored in off-heap memory, providing optimal performance when using Hybrid (though there is an option to store cache keys on disk, but this comes with a performance penalty and is not recommended).

Figure 1. BigMemory Hybrid allows you to expand BigMemory in Terracotta servers, keeping more data closer to your application for increased transactions per second (TPS).



How is BigMemory Hybrid different than Overflow to Disk?

BigMemory Hybrid	Overflow to Disk
Available in version 4.1	Available in version 3.7
Leverages BigMemory's Fast Restart technology	Depends upon Berkeley DB to store data on disk
Manages all data in SSD/Flash for predictable performance. Only the cache keys are stored in off-heap memory.	If data does not fit in off-heap, it is pushed to disk, hence performance is less predictable
Optimized for SSD usage	No optimization done for SSDs

System Requirements

BigMemory Hybrid supports writing to one single mount, so all of the BigMemory Hybrid capacity must be presented to the Terracotta process as one continuous region, which can be a single device or a RAID.

The mount should be used exclusively for the Terracotta server process.

Note: System utilization is higher when using BigMemory Hybrid, and it is not recommended to run multiple servers on the same machine. Doing so could result health checkers timing out, and killing or restarting servers. Therefore, it is important to provision sufficient hardware, and it is highly recommended to deploy servers on different machines.

Hardware Capacity Guidelines

To account for the overhead necessary for consistent performance, the formulas below are suggested as initial starting points for sizing the amount of space allocated for BigMemory Hybrid operation.

Minimum SSD flash memory requirement = planned total data size * 3.2

Minimum DRAM requirement = planned maximum number of elements * (168 + key size)

Note: It is strongly recommended to configure enough offheap to accommodate all cache keys in DRAM.

Configuring BigMemory Hybrid

To configure BigMemory Hybrid, include the following elements in the tc-config.xml file:

- **dataStorage** - Specifies the maximum amount of data you plan to store on the server, using either DRAM alone or both DRAM and SSD flash memory.
- **offheap** - Specifies the maximum amount of data to hold in DRAM.
- **hybrid** - Enables the Hybrid option to use SSD flash memory in addition to off-heap DRAM.

For example:

```
<servers>
    ....
    <server host="hostname" name="server1">
        ...
        <dataStorage size="800g">
```

```
            <offheap size="200g"/>
            <hybrid/>
        </dataStorage>
    </server>
</servers>
```

For Terracotta servers, a minimum of 4 GB is recommended for the size attribute of the `offheap` element.

If the `hybrid` element is present, then the BigMemory Hybrid functionality is enabled. With Hybrid enabled, the value of the size attribute for the `dataStorage` element can exceed that of the size attribute for the `offheap` element. This enables SSD devices to supplement the DRAM and be many times larger than the DRAM.

If the `hybrid` element is absent, then BigMemory Hybrid functionality is off. With Hybrid off, the value of the size attribute for the `dataStorage` element must be less than or equal to the value of the size attribute for the `offheap` element. In this case, the `offheap` element is not required.

If the `dataStorage` element is absent, `dataStorage` size and `offheap` size default to 512 MB.

Although the `dataStorage` element is optional, if included, this element must have a value assigned to its size attribute.

Note: If you are migrating from BigMemory Max 4.0 to 4.1, the `dataStorage` element has replaced the `maxDataSize` element. The old element is still compatible for pure DRAM operation, but to enable Hybrid mode, you must use the new 4.1-compatible `dataStorage` element with the `hybrid` tag.

Disk Storage Path

BigMemory Hybrid requires a unique and explicitly specified path. The default path is the Terracotta server's home directory. You can customize the path using the `<data>` element in the server's tc-config.xml configuration file.

BigMemory Hybrid and Fast Restartability

If Fast Restartabillity is enabled, then if you have a restart, data will be loaded into BigMemory Hybrid in the same way as for BigMemory, with no difference in behavior or time required to get the system running again. See "Fast Restartability" on page 15.

If Fast Restartabillity is not enabled, then on restart, you will have some artifacts from the previous run left on disk, and you may want to remove them. For more information, see "Clearing Data from a Terracotta Server" on page 59.

Using the TMC with BigMemory Hybrid

When you use the Terracotta Management Console (TMC), you can see the effect of the BigMemory Hybrid feature in the Monitoring > Runtime Stats panel as "Data Storage

Usage". When the cache is operating at a steady state, the Data Used typically exceeds the OffHeap Max shown in the "OffHeap Usage" graph:

Operator Events

BigMemory Hybrid supports the existing operator events in the Terracotta Server Array (TSA), including

- a Throttle Mode when the amount of either RAM, Flash, or both is approaching its capacity limit. See "Managing Near-Memory-Full Conditions" on page 43.
- a Restricted Mode when the system has entered read-only mode.

For more information, see "Managing Near-Memory-Full Conditions" on page 43.

14 Monitoring and Management Using JMX

- About Using JMX 80
- MBeans 81
- JMX Remoting 81
- ObjectName Naming Scheme 82
- The Management Service 82
- JConsole Example 83
- JMX Tutorial 84
- Performance Considerations 84
- SSL-Secured Monitoring with JMX 84

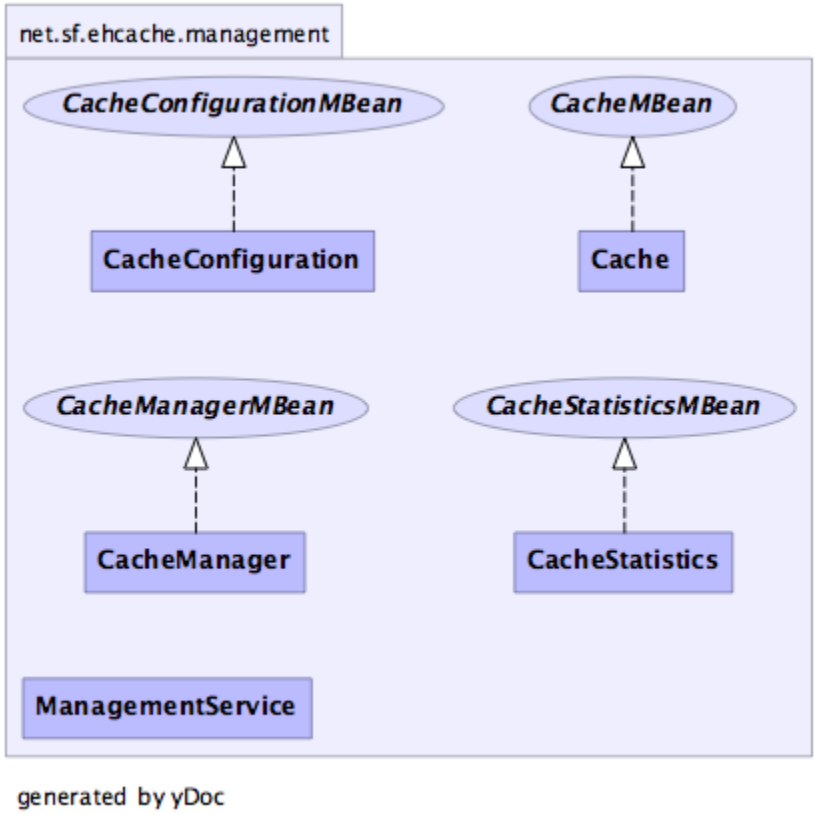
About Using JMX

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 Server for custom or third-party tools.

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 the net.sf.ehcache.management.ManagementService.registerMBeans(...) static method to register a selection of MBeans to the MBeanServer provided to the method. If you want 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.



Note: The JMX port configuration of the Terracotta server is disabled by default. To enable it for monitoring with JMX, add `jmx-enabled="true"` in the

`<server>` element in the Terracotta configuration file tc-config.xml. For example:

```
<server host="localhost" name="My Server Name1" jmx-
enabled="true">
```

For more information about monitoring with JMX , see "SSL-Secured Monitoring with JMX" on page 84.

For an alternative to monitoring with JMX, use the monitoring features provided by the Terracotta Management Console (see the *Terracotta Management Console User Guide*).

MBeans

BigMemory Max supports Standard MBeans. MBeans are available for the following:

- 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, 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

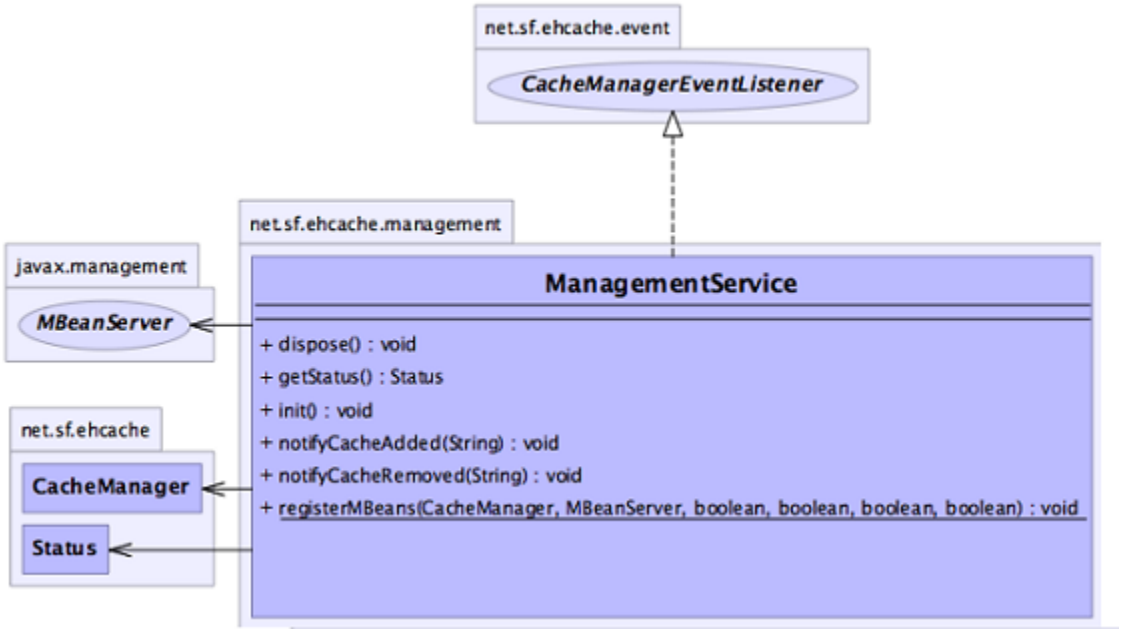
```
"net.sf.ehcache:type=CacheConfiguration,CacheManager=<cacheManagerName>,
   name=<cacheName>"
```

CacheStatistics

```
"net.sf.ehcache:type=CacheStatistics,CacheManager=<cacheManagerName>,
   name=<cacheName>"
```

The Management Service

The ManagementService class is the API entry point.



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.

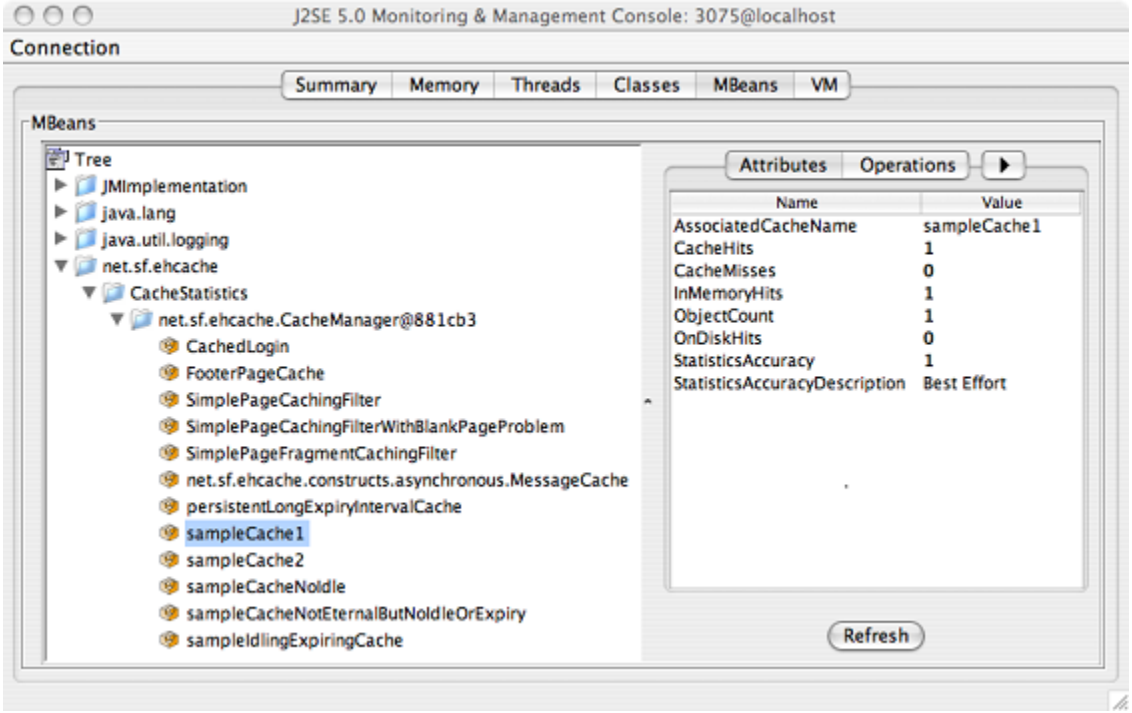
```
/**
* This method causes the selected monitoring options to be be registered
* with the provided MBeanServer for caches in the given CacheManager.
*
* While registering the CacheManager enables traversal to all of the other
* items, this requires programmatic traversal. The other options allow entry
* 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 shutsdown 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(
   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 following shows CacheStatistics MBeans in JConsole.



JMX Tutorial

See this online tutorial.

Performance Considerations

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 Monitoring with JMX

The following describes how to set up an SSL-enabled connection for remote monitoring of a Terracotta Server from a simple Java client using Java Management Extensions (JMX) technology.

Before creating this connection, be sure to complete the instructions described in the *BigMemory Max Security Guide*.

Note: The JMX port configuration of the Terracotta server is disabled by default. To enable it for monitoring with JMX, add `jmx-enabled="true"` in the

`<server>` element in the Terracotta configuration file tc-config.xml. For example:

```
<server host="localhost" name="My Server Name1" jmx-
enabled="true">
```

For an alternative to monitoring with JMX, use the monitoring features provided by the Terracotta Management Console (see the *Terracotta Management Console User Guide*).

Compile the Client

Use the sample client code below, but adapt the host, port, username, and password variables according to your setup.

Note: The JMX port (specified below in the line `String port = "9520"`) should match the JMX port defined in the Terracotta configuration file tc-config.xml (for example, `<jmx-port>9520</jmx-port>`).

```
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.RMIConnectorServer;
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(RMIConnectorServer.RMI_CLIENT_SOCKET_FACTORY_ATTRIBUTE, csf);
    env.put(RMIConnectorServer.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

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 (see "Setting Up Server Security" in the *BigMemory Max Security Guide*).

```
% java -Djavax.net.ssl.trustStore=/your/path/to/truststore.jks \
  -Djavax.net.ssl.trustStorePassword=your_truststore_password \
  Main
```

About the Credentials

In the example above, 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.nami
    java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing impl
    at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.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 impl
    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.RMIConnector.findRMIServerJNDI(RMIConnector.java:1886)
    at javax.management.remote.rmi.RMIConnector.findRMIServer(RMIConnector.java:1856)
    at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:255)
    ... 7 more
Caused by: java.rmi.ConnectIOException: Exception creating connection to: localhost; nested e
    java.net.SocketException: java.security.NoSuchAlgorithmException: Error constructing impl
    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 constructi
    at javax.net.ssl.DefaultSSLSocketFactory.throwException(SSLSocketFactory.java:179)
    at javax.net.ssl.DefaultSSLSocketFactory.createSocket(SSLSocketFactory.java:192)
    at javax.rmi.ssl.SslRMIClientSocketFactory.createSocket(SslRMIClientSocketFactory.java:10
    at sun.rmi.transport.tcp.TCPEndpoint.newSocket(TCPEndpoint.java:595)
    ... 17 more
Caused by: java.security.NoSuchAlgorithmException: Error constructing implementation (algorit
    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(SslRMIClientSock
        at javax.rmi.ssl.SslRMIClientSocketFactory.createSocket(SslRMIClientSocketFactory.java:10
        ... 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$JKS.engineLoad(JavaKeyStore.java:38)
        at java.security.KeyStore.load(KeyStore.java:1185)
        at com.sun.net.ssl.internal.ssl.TrustManagerFactoryImpl.getCacertsKeyStore(TrustManagerFa
        at com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl.getDefaultTrustManager(DefaultSSLCo
        at com.sun.net.ssl.internal.ssl.DefaultSSLContextImpl.<init>(DefaultSSLContextImpl.java:4
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
        at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.ja
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccesso
        at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
        at java.lang.Class.newInstance0(Class.java:357)
        at java.lang.Class.newInstance(Class.java:310)
        at java.security.Provider$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.nami
        javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.I
        at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.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
        javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.I
        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.RMIConnector.findRMIServerJNDI(RMIConnector.java:1886)
        at javax.management.remote.rmi.RMIConnector.findRMIServer(RMIConnector.java:1856)
        at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:255)
        ... 7 more
    Caused by: java.rmi.ConnectIOException: error during JRMP connection establishment; nested ex
        javax.net.ssl.SSLException: java.lang.RuntimeException: Unexpected error: java.security.I
        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)
        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.sec
        at com.sun.net.ssl.internal.ssl.Alerts.getSSLException(Alerts.java:190)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1747)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.fatal(SSLSocketImpl.java:1708)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.handleException(SSLSocketImpl.java:1691)
        at com.sun.net.ssl.internal.ssl.SSLSocketImpl.handleException(SSLSocketImpl.java:1617)
        at com.sun.net.ssl.internal.ssl.AppOutputStream.write(AppOutputStream.java:105)
```

```
        at java.io.BufferedOutputStream.flushBuffer(BufferedOutputStream.java:65)
        at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:123)
        at java.io.DataOutputStream.flush(DataOutputStream.java:106)
        at sun.rmi.transport.tcp.TCPChannel.createConnection(TCPChannel.java:211)
        ... 16 more
    Caused by: java.lang.RuntimeException: Unexpected error: java.security.InvalidAlgorithmParame
        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.ja
        at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerI
        at com.sun.net.ssl.internal.ssl.X509TrustManagerImpl.checkServerTrusted(X509TrustManagerI
        at com.sun.net.ssl.internal.ssl.ClientHandshaker.serverCertificate(ClientHandshaker.java:
        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:
        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.security.InvalidAlgorithmParameterException: the trustAnchors parameter must
        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:202
    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)
    at sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.java:649)
    at java.util.concurrent.ThreadPoolExecutor$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:25
    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.RMIConnector.getConnection(RMIConnector.java:2327)
    at javax.management.remote.rmi.RMIConnector.connect(RMIConnector.java:277)
    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)
```

15 Logging

- SLFJ Logging 92
- Recommended Logging Levels 92

SLFJ Logging

BigMemory Max uses the SLF4J logging facade, so you can plug in your own logging framework. The following information pertains to Ehcache logging. For information about SLF4J in general, refer to the SLF4J website.

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 jar is in the kit along with the BigMemory Max 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 Max seeks to trade off informing production-support developers of 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.

A Operational Scripts

- Archive Utility (archive-tool) 94
- Database Backup Utility (backup-data) 94
- Backup Status (backup-status) 95
- Cluster Thread and State Dumps (debug-tool, cluster-dump) 95
- Distributed Garbage Collector (run-dgc) 96
- Start and Stop Server Scripts (start-tc-server, stop-tc-server) 97
- Server Status (server-stat) 98
- Version Utility (version) 99

Archive Utility (archive-tool)

The archive-tool is used to gather logs generated by a Terracotta server or client for the purpose of contacting Terracotta with a support query.

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\archive-tool.bat <args>
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/archive-tool.sh <args>
```

where `<args>` are:

- [-n] (No Data - excludes data files)
- [-c] (Client - include files from the client)
- [path to terracotta config xml file (tc-config.xml) or path to logs directory
- [output filename in .zip format]

Database Backup Utility (backup-data)

The backup utility creates a backup of the data being shared by your application by taking a snapshot of the data held by the Terracotta Server Array (TSA). Unless a different directory is specified in configuration, backups are saved to the default directory `${user.dir}/terracotta/backups`.

Configuring Backup

You can override this default behavior by specifying a different backup directory in the server's configuration file using the `<data-backup>` property:

```
<servers>
   <restartable enabled="true"/>
...
   <server host="%i" name="myServer">
   ...
     <data-backup>/Users/myBackups</data-backup>
   </server>
...
</servers>
```

Note that enabling `<restartable>` mode is required for using the backup utility.

Creating a Backup

The backup utility relies on the Terracotta Management Server (TMS) to locate and execute backups. The TMS must be running and connected to the TSA for the backup to take place. For more information about connecting the Terracotta Management Server to the TSA, see the *Terracotta Management Console User Guide*.

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\tools\management-console\bin\backup-data.bat <args>
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/tools/management-console/bin/backup-data.sh <args>
```

where `<args>` are:

- [l] <tms-host:port> – The host and port used to connect to the TMS. If omitted, `http://localhost:9889` is used by default.
- [u] <username> – If the TMS requires authentication, a username must be specified.
- [p] <password> – If the TMS requires authentication, a password must be specified.
- [a] <agentID> – Specify the agent ID of a TSA. The agent ID is set as a connection name when the connection to the TSA is configured on the TMS. If no agent ID is provided, the TMS returns a list of configured agent IDs.
- [k] This flag causes invalid TMS SSL certificates to be ignored.

For example, to initiate a backup on a cluster with the agent ID "someConnection":

```
${BIGMEMORY_HOME}/tools/management-console/bin/
backup-data.sh -l http://my-tms-host:9889 \ -u admin
-p admin -a someConnection -k
```

If initiation is successful, the script reports that the backup process has started. Once the backup is complete, the backup data files can be used to restore data in place of the current data files. For information about restoring data from a backup, see "Restoring Data from a Backup" on page 57.

Backup Status (backup-status)

The backup-status script is run from the tools/management-console/bin directory. This script complements the backup-data utility by checking on the status of executed backups for a specified cluster. For example, to return a list of backup operations on the agent myClusterAgent:

```
[PROMPT] ${BIGMEMORY_HOME}/tools/management-console/bin/backup-status
    -l http://myTMShost:9889 -a myClusterAgent
```

The backup-status script takes the same arguments as backup-data. For details, see "Database Backup Utility (backup-data)" on page 94.

Cluster Thread and State Dumps (debug-tool, cluster-dump)

The cluster and thread- and state-dump debug tools provide a way to easily generate debugging information that can be analyzed locally or forwarded to support personnel. These tools work against the Terracotta Management Server that is monitoring the target Terracotta cluster. All components must be running at the time a tool is used.

- `debug-tool` generates thread dumps for all nodes in the cluster, with each node's dump saved its log file. A flag is available for saving the thread dumps to a single zip file.
- `cluster-dump` provides a similar service, but adds each node's state, including information on locks. Note that these tools can generate a substantial amount of data.

Note: Server utility scripts do not work when a server is starting up or when a server is in the process of recovering using the Fast Restart feature.

For more information on operating these tools, run the associated script with the `-h` flag. For example:

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\tools\management-console\bin\debug-tool.bat -h
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/tools/management-console/bin/debug-tool.sh -h
```

Distributed Garbage Collector (run-dgc)

`run-dgc` is a utility that causes the specified cluster to perform distributed garbage collection (DGC). Use run-dgc to force a periodic DGC cycle in environments where inline DGC is not in effect. However, automated DGC collection is sufficient for most environments.

This utility relies on the Terracotta Management Server (TMS) to locate and execute backups. The TMS must be running and connected to the TSA for the DGC to be initiated. For more information about connecting the Terracotta Management Server to the TSA, see the *Terracotta Management Console User Guide*.

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\tools\management-console\bin\run-dgc.bat <args>
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/tools/management-console/bin/run-dgc.sh <args>
```

where `<args>` are:

- [l] <tms-host:port> – The host and port used to connect to the TMS. If omitted, `localhost:9889` is used by default.
- [u] <username> – If the TMS requires authentication, a username must be specified.
- [p] <password> – If the TMS requires authentication, a password must be specified.
- [a] <agentID> – Specify the agent ID of a TSA. The agent ID is set as a connection name when the connection to the TSA is configured on the TMS. If no agent ID is provided, the TMS returns a list of configured agent IDs.
- [k] This flag causes invalid TMS SSL certificates to be ignored.

Note: 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

For more information on distributed garbage collection, see "Managing Distributed Garbage Collection" on page 67.

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:

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\start-tc-server.bat ^
    [-n <name of server>] [-f <config specification>]
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/start-tc-server.sh \
    [-n <name of server>] [-f <config specification>]
```

`<config specification>` can be one of:

- Path to configuration file
- URL to configuration file
- <server host>:<tsa-port> of another running Terracotta Server

Note the following:

- 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.
- If no server is named, and more than one server exists in the configuration file used, an error is printed to standard out and no server is started.

Use the `stop-tc-server` script to cause the Terracotta Server to gracefully terminate:

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\stop-tc-server.bat <host-name> <jmx-port> <args>
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/stop-tc-server.sh <host-name> <jmx-port> <args>
```

where `<args>` are:

- [f] `<file-or-URL>` – Specifies the tc-config file to use, as a file path or URL. For an SSL-secured server, a valid path to the self-signed certificate must have been specified in the server's configuration file.
- [--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. For information about production mode, see "Enabling Production Mode" on page 65.

- [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. For an SSL-secured server, the user specified must have the "admin" role.
- [w] – Specify the JMX password.
- [k] – This flag causes invalid TMS SSL certificates to be ignored. Use this option to accept self-signed certificates (ones not signed by a trusted CA).

For more information, see "Setting up Server Security" in the *BigMemory Max Security Guide*.

Server Status (server-stat)

The status tool is a command-line utility for checking the current status of one or more Terracotta server instances.

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\server-stat.bat <args>
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/server-stat.sh <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 specified configuration file.
- [-k] – This flag causes invalid TMS SSL certificates to be ignored. Use this option to accept self-signed certificates (ones not signed by a trusted CA).

The status tool 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-mirror group. Single servers always show ACTIVE. Backups are shown as MIRROR or PASSIVE.
- *State* – The work state that the server is in. When ready, active servers should show ACTIVE-COORDINATOR, while mirror servers should show MIRROR-STANDBY or PASSIVE-STANDBY.
- *JMX port* – The TCP port the server is using to listen for JMX events.
- *Error* – If the status tool fails, the type of error.

Example

The following example shows usage of and output from the status tool.

```
[PROMPT] server-stat.sh -s myhost:9521
 localhost.health: OK
 localhost.role: ACTIVE
 localhost.state: ACTIVE-COORDINATOR
 localhost.jmxport: 9521
```

If no server is specified, by default the tool checks the status of localhost at JMX port 9520.

Version Utility (version)

The version tool is a utility script that outputs information about the BigMemory 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 the version tool to expedite the resolution of your issue.

For Microsoft Windows:

```
[PROMPT] %BIGMEMORY_HOME%\server\bin\version.bat
```

For UNIX/Linux:

```
[PROMPT] ${BIGMEMORY_HOME}/server/bin/version.sh&
```

Use the following flags to produce more information:

- [r] – Produces detailed, raw information in a "property=value" format.
- [v] – Produces more detailed information.

B Terracotta Configuration Parameters

- The Terracotta Configuration File 102
- The Servers Parameters 105
- The Clients Parameters 112

The Terracotta Configuration File

This document is a reference to all of the Terracotta configuration elements in the Terracotta configuration file, which is named `tc-config.xml` by default.

You can use a sample configuration file provided in the kit as the basis for your Terracotta configuration. Some samples have inline comments describing the configuration elements. Be sure to start with a clean file for your configuration.

The Terracotta configuration XML document is divided into the sections <servers> and <clients>.

- The <servers> section contains parameters you use to configure the behavior and characteristics of the Terracotta Server Array and its component servers.
- The <clients> section contains parameters you use to configure client behavior.

Configuration Variables

Certain variables can be used that are interpolated by the configuration subsystem using local values:

Variable	Interpolated Value
%h	The fully-qualified hostname
%i	The IP address
%o	The operating system
%v	The version of the operating system
%a	The CPU architecture
%H	The home directory of the user running the application
%n	The username of the user running the application
%t	The path to the temporary directory (for example, /tmp on *NIX)
%D	Time stamp (yyyyMMddHHmmssSSS)

Variable	Interpolated Value
%(*system property*)	The value of the given Java system property

These variables can be used where appropriate, including for elements or attributes that expect strings or paths for values:

- 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: 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. Test this variable in your production environment to check the value it interpolates.

Using Paths as Values

Some configuration elements take paths as values. Relative paths are interpreted relative to the current working directory (the directory from which the server was started). Specifying an absolute path is recommended.

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
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>`.

Note: The property `tc.management.mbeans.enabled` is known to load before initialization completes and cannot be overridden.

The Servers Parameters

/tc:tc-config/servers

This section defines the Terracotta server instances present in your cluster. One or more entries can be defined, either directly under the <servers> element or in the "/tc:tc-config/servers/mirror-group" on page 110. If this section is omitted, Terracotta configuration behaves as if there's a single server instance with default values.

This section also defines certain global settings that affect all servers, including the attribute `secure`. This is a global control for enabling ("true") or disabling ("false" DEFAULT) SSL-based security for the entire cluster.

For information about SSL-based security, see the *BigMemory Max Security 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
host	The address of the machine hosting the Terracotta server	Host machine's IP address or resolvable hostname	Host machine's IP address
name	The symbolic name of the Terracotta server; can be passed to Terracotta scripts such as start-tc-server using `-n <name>`	user-defined string	<host>:<tsa-port>
bind	The network interface on which the Terracotta server listens cluster traffic; 0.0.0.0 specifies all interfaces	interface's IP address	0.0.0.0

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.

Here is a sample configuration snippet

```
<server>
  <!-- my host is '%i', my name is '%i:tsa-port', my bind is 0.0.0.0 -->
  ...
</server>
<server host="myhostname">
  <!-- my host is 'myhostname', my name is 'myhostname:tsa-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 element specifies the path where the server should store its data for persistence.

Default: data (creates the directory `data` under the working directory)

/tc:tc-config/servers/server/logs

This section lets you declare where the server should write its logs.

Default: logs (creates the directory `logs` under the working directory)

You can also specify `stderr:` or `stdout:` as the output destination for log messages. For example:

```
<logs>stdout:</logs>
```

/tc:tc-config/servers/server/index

This element specifies the path where the server should store its search indexes.

Default: index (creates the directory `index` under the working directory)

/tc:tc-config/servers/server/data-backup

This element specifies the path where the server should store backups (if a backup call is initiated).

Default: data-backup (creates the directory `data-backup` under the working directory)

/tc:tc-config/servers/server/tsa-port

This section lets you set the port that the Terracotta server listens to for client traffic.

The default value of "tsa-port" is 9510.

Here is a sample configuration snippet:

```
<tsa-port>9510</tsa-port>
```

/tc:tc-config/servers/server/jmx-port

Note: Listening on the "jmx-port" is deprecated. Alternatively, use the monitoring features provided by the Terracotta Management Console (see the *Terracotta Management Console User Guide*) and the WAN Replication Service (see the *WAN Replication User Guide*).

"jmx-port" is disabled by default. To enable it, add `jmx-enabled="true"` to the `<server host>` sections of tc-config.xml. For example:

```
<server host="localhost" name="My Server Name1" jmx-
enabled="true">
```

This section lets you set the port that the Terracotta server's JMX Connector listens to.

The default value of "jmx-port" is 9520. If tsa-port was set to a value other than the default 9510, this port defaulted to the value of the tsa-port plus 10.

Here is a sample configuration snippet:

```
<jmx-port>9520</jmx-port>
```

/tc:tc-config/servers/server/tsa-group-port

This section lets you set the port that the Terracotta server uses to communicate with other Terracotta servers.

The default value of "tsa-group-port" is 9530. If tsa-port is set to a value other than the default 9510, this port defaults to the value of the tsa-port plus 20.

Here is a sample configuration snippet:

```
<tsa-group-port>9530</tsa-group-port>
```

/tc:tc-config/servers/server/management-port

This section lets you set the port that the Terracotta Management Console (TMC) uses.

The default value of "management-port" is 9540. If tsa-port is set to a value other than the default 9510, this port defaults to the value of the tsa-port plus 30.

Here is a sample configuration snippet:

```
<management-port>9540</management-port>
```

Prior to 4.2, the TMC used the ports specified in `/tc:tc-config/servers/server/tsa-port` (port 9510 by default) and `/tc:tc-config/servers/server/tsa-group-port` (port 9530 by default). Now the TMC uses only the port specified in `/tc:tc-config/servers/server/management-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.

For more information, see "About Security in a Cluster" in the *BigMemory Max Security Guide*.

/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/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.
- `<secret-provider>` – The fully qualified class name of the user implementation of `com.terracotta.management.security.SecretProviderBackEnd`. This class can read and provide 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. Alternatively, URIs for LDAP or Microsoft Active directory can also be used if one of these schemes is implemented instead.

- `<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. The default value is "terracotta".

/tc:tc-config/servers/server/security/management

This element contains the subelements needed to allow the Terracotta Management Server (TMS) to make a secure connection to the TSA:

- `ia` – The HTTPS URL with the domain of the TMS, followed by the port 9443 and the path `/tmc/api/assertIdentity`.
- `timeout` – The timeout value (in milliseconds) for connections from the server to the TMS.
- `hostname` – Used only if the DNS hostname of the server does not match server hostname used in its certificate. If there is a mismatch, enter the DNS address of the server here.

/tc:tc-config/servers/server/authentication

Turn on JMX authentication for the Terracotta server. An empty tag (`<authentication />`) defaults to the standard Java JMX authentication mechanism referring to password and access files in `$JAVA_HOME/jre/lib/management`:

```
$JAVA_HOME/jre/lib/management/jmxremote.password
$JAVA_HOME/jre/lib/management/jmxremote.access
```

You must modify these files as follows (or, if none exist create them).

jmxremote.password: Add a line to the end of the file declaring a username and password followed by a carriage return:

```
secretusername secretpassword
```

jmxremote.access: Add the following line (with a carriage return) to the end of your file:

```
secretusername      readwrite
```

Be sure to assign the appropriate permissions to the file. For example, in *NIX:

```
$ chmod 500 jmxremote.password
$ chown <user who will run the server> jmxremote.password
```

For information on alternatives to JMX authentication, see the *BigMemory Max Security Guide*.

Note that version 4.x does not support HTTP authentication.

/tc:tc-config/servers/dataStorage

This configuration block includes the required `offheap` element, as well as the optional `hybrid` element.

- `<offheap>` must be configured for each server; the minimum amount is 4 GB.
- `<dataStorage>` specifies the maximum amount of data to store on each server, and represents the hybrid sum of off-heap DRAM plus flash SSD.
- `<hybrid/>`, if included, enables the Hybrid option.

Here is a sample configuration snippet:

```
<dataStorage size="800g">
   <offheap size="200g"/>
   <hybrid/>
</dataStorage>
```

If the `hybrid` element is present, then `dataStorage` size may exceed `offheap` size, however if the hybrid element is not present, then the `dataStorage` size must be less than or equal to the `offheap` size.

/tc:tc-config/servers/mirror-group

A mirror group is a *stripe* in a TSA, consisting of one active server and one or more mirror (or backup) servers. A configuration that does not use the <mirror-group> element would produce a one-stripe TSA:

```
<servers>
  <server name="A">
  ...
  </server>
  <server name="B">
  ...
  </server>
  <server name="C">
  ...
  </server>
  <server name="D">
  ...
  </server>
...
</servers>
```

One of the named servers would assume the role of active (the one started first or that wins the election), while the remaining servers become mirrors. Note that in a typical stripe, having only one or two mirrors is sufficient and less taxing on the active server's resources (as it needs to sync with each mirror).

The following example shows the same servers split into two stripes:

```
<servers>
  <mirror-group group-name="team1">
    <server name="A">
    ...
    </server>
    <server name="B">
    ...
    </server>
  </mirror-group>
  <mirror-group group-name="team2">
    <server name="C">
    ...
```

```
    </server>
    <server name="D">
    ...
    </server>
  </mirror-group>
...
</servers>
```

Each stripe will have one active and one mirror server.

Note: Under <servers>, you may use either <server> or <mirror-group> configurations, but not both. All <server> configurations directly under <servers> work together as one mirror group, with one active server and the rest mirrors. To create more than one stripe, use <mirror-group> configurations directly under <servers>. The mirror group configurations then include one or more <server> configurations

For more examples and information, see "Configuring the Terracotta Server Array" on page 25.

/tc:tc-config/servers/garbage-collection

This section lets you configure the periodic distributed garbage collector (DGC) that runs in the TSA. The DGC collects shared data made garbage by Java garbage collection.

For many use cases, there is no need to enable periodic DGC. For caches, the more efficient automatic inline DGC is normally sufficient for clearing garbage. In addition, certain read-heavy applications will never require the periodic DGC as little shared data becomes garbage.

However, concerning certain data structures, the periodic DGC may need to be enabled. Inline DGC may not be available for all data structures.

For more on how DGC functions, see "Managing Distributed Garbage Collection" on page 67.

Here is a configuration snippet:

```
<garbage-collection>
 <!--    Default: false -->
  <enabled>true</enabled>
  <!-- If "true", additional information is logged when a
     server performs distributed garbage collection.
     Default: false
  -->
  <verbose>false</verbose>
  <!-- How often should distributed garbage collection
     be performed, in seconds?
     Default: 3600 (60 minutes)
  -->
  <interval>3600</interval>
</garbage-collection>
```

/tc:tc-config/servers/restartable

The fast-restart persistence mechanism must be explicitly enabled for the TSA using this element:

```
<restartable enabled="true"/>
```

In case of TSA failure, fast-restart persistence allows the TSA to reload all shared cluster data.

To function, this feature requires <offheap> to be enabled on each server. To make backups of TSA data, the backup feature requires this feature to be enabled.

/tc:tc-config/servers/client-reconnect-window

This section lets you declare the window of time servers will allow disconnected clients to reconnect to the cluster as the same client. Outside of this window, a client can only rejoin as a new client. The value is specified in seconds and the default is 120 seconds.

If adjusting value, note that a too-short reconnection window can lead to unsuccessful reconnections during failure recovery, while a too-long window lowers the efficiency of the cluster since it is paused for the time the window is in effect.

For more information on how client and server reconnection is executed in a Terracotta cluster, and on tuning reconnection properties in a high-availability environment, see the *BigMemory Max High-Availability Guide*.

The Clients Parameters

/tc:tc-config/clients/logs

This section lets you configure where the Terracotta client writes its logs.

Here is a 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 "Logging" on page 91.