Terracotta Server Administration Guide

Version 10.15

October 2022

Document ID: TC-SRV-AG-1015-20221101

Table of Contents

About This Documentation

Online Information and Support

Product Documentation

You can find the product documentation on our documentation website at https://documentation.softwareag.com.

In addition, you can also access the cloud product documentation via https://www.softwareag.cloud. Navigate to the desired product and then, depending on your solution, go to "Developer Center", "User Center" or "Documentation".

Product Training

You can find helpful product training material on our Learning Portal at https://knowledge.softwareag.com.

Tech Community

You can collaborate with Software AG experts on our Tech Community website at https://techcommunity.softwareag.com. From here you can, for example:

- Browse through our vast knowledge base.
- Ask questions and find answers in our discussion forums.
- Get the latest Software AG news and announcements.
- Explore our communities.
- Go to our public GitHub and Docker repositories at https://github.com/softwareag and https://hub.docker.com/u/softwareag and discover additional Software AG resources.

Product Support

Support for Software AG products is provided to licensed customers via our Empower Portal at https://empower.softwareag.com. Many services on this portal require that you have an account. If you do not yet have one, you can request it at https://empower.softwareag.com/register. Once you have an account, you can, for example:

- Download products, updates and fixes.
- Search the Knowledge Center for technical information and tips.
- Subscribe to early warnings and critical alerts.
- Open and update support incidents.
- Add product feature requests.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 Cluster Architecture

The Terracotta cluster can be viewed topologically as a collection of *clients* communicating with a Terracotta Server Array (TSA).

The server array is composed of one or more logically independent stripes. The total storage and computing capacity of the TSA can be increased with the addition of more stripes.

A stripe is composed of one or more servers. Each stripe contains a single *active server* and zero or more *passive servers*. These stripe members share a common configuration in order for any one of them to fill the "active" role. Refer to the section "The TerracottaConfiguration File" on page 45 for more details.

"Scale-up" is achieved by configuring the servers to utilize more storage (e.g. memory) from the machines that they are deployed on.

"Scale-out" is achieved by adding more stripes to the TSA.

Greater levels of "HA" (high availability) are achieved by adding more members and/or voters to each stripe.

For more information on clients, active servers and passive servers, see the sections "Clients in a Cluster" on page 19 and "Active and Passive Servers" on page 13.

TSA Topologies

There are multiple types of TSA topology, each offering different resource and availability capabilities.

TSA Topology	Description
Single-server	This is a TSA which consists of one stripe containing a single server. This server is always the active server. This scheme offers the least amount of both resource and availability capabilities: ■ If this server should become unavailable, your client end-points will fail to operate. ■ The resource services exposed to your clients are limited to those of the underlying server JVM and OS.

TSA Topology	Description
High-availability	This refers to a TSA where each stripe consists of at least two servers. In addition to the active server there will be at least one passive server. The stripe may continue operating in the event of an active server failure, as long as at least one passive server is available. Note that stripes do not share passive servers, so each stripe will need at least one passive server to possess high-availability. Depending upon your overall topology, the use of external voters may also help achieve greater levels of HA. See "Failover" on page 23 and "Failover Tuning" on page 99 for more discussion on HA topics.
Multi-stripe	Multi-stripe refers to a TSA that consists of multiple independent stripes. This scheme offers the ability for increased storage and computation resources, with each stripe contributing to the available total amount of storage. For a multi-stripe TSA to possess high-availability, each stripe must consist of more than one server. This setup offers the maximum of both resource and availability capabilities.

Client perspective

Each client is logically independent of other clients. It sees the TSA as a collection of one or more stripes. It connects to the active server of each stripe in order to issue messages to the cluster.

Stripe perspective

Each stripe is logically independent of other stripes in the TSA. Each stripe member only concerns itself with the clients connected to it and its sibling servers.

Specifically, the active server is the key point in each stripe: each stripe has exactly one active server and it is this server which interacts directly with each connected client and each passive server within the same stripe.

2 Active and Passive Servers

Introduction

Terracotta Servers exist in two modes, *active* and *passive*. The description of each mode is given below.

Active servers

Within a given stripe of a cluster, there is always an active server. A server in a single-server stripe is always the active server. A multi-server stripe will only ever have one active server at a given point in time.

The active server is the server which clients communicate with directly. The active server relays messages on to the *passive servers* independently.

How an active server is chosen

When a stripe starts up, or a failover occurs, the online servers perform an *election* to decide which one will become the active server and lead the stripe. For more information about elections, see the section "Electing an Active Server" on page 21.

How clients find the active server

Clients will attempt to connect to each server in the stripe, and only the active server will accept the connection.

The client will continue to only interact with this server until the connection is broken. It then attempts the other servers if there has been a *failover*. For more information about failover, see the section "Failover" on page 23.

Responsibilities of the active server

The active server differs from passive servers in that it receives all messages from the clients. It is then responsible for sending back responses to the calling clients.

Additionally, the active server is responsible for replicating the messages that it receives on the passive servers.

When a new server joins the stripe, the active server is responsible for synchronizing its internal state with the new server, before telling it to enter a standby state. This state means that the new server is now a valid candidate to become a new active server in the case of a failover.

Passive servers

Any stripe of a cluster which has more than one running server will contain passive servers. While there is only one *active* server per stripe, there can be zero, one, or several passive servers.

Passive servers are also referred to as "mirrors", because they contain a copy of the data that is present within the active server.

Passive servers go through multiple states before being available for failover:

UNINITIALIZED	This passive server has just joined the stripe and has no data.
SYNCHRONIZING	This passive server is receiving the current state from the active server. It has some of the stripe data but not yet enough to participate in *failover*.
STANDBY	This passive contains the stripe data and can be a candidate to become the active, in the case of a failover.

Passive servers only communicate with the active server, not with each other, and not with any clients.

How a server becomes passive

When a stripe starts up and a server fails to win the election, it becomes a passive server.

Additionally, newly-started servers which join an existing stripe which already has an active server will become passive servers.

Responsibilities of the passive server

The passive server has far fewer responsibilities than an active server. It only receives messages from the active server, not communicating directly with other passive servers or any clients interacting with the stripe.

Its key responsibility is to be ready to take over the role of the active server in the case that the active server crashes, loses power/network, or is taken offline for maintenance/upgrade activities.

All the passive server does is apply messages which come from the active server, whether the initial state synchronization messages when the passive server first joined, or the on-going replication of new messages. This means that the state of the passive server is considered consistent with that of the active server.

Lifecycle of the passive server

When a passive server first joins a stripe and determines that its role will be passive, it is in the *UNINITIALIZED* state.

If it is a *restartable* server and also discovers existing data from a previous run, it makes a backup of that data for safety reasons. Refer to the section "Clearing Automatic Backup Data" on page 15 for more details.

Refer to the section *Restarting a Stripe* in the *Terracotta Server Administration Guide* for information on the proper order in which to restart a *restartable* stripe.

From here, the active server begins sending it messages to rebuild the active server's current state on the passive server. This puts the passive server into the *SYNCHRONIZING* state.

Once the entire active state has been synchronized to the passive server, the active server tells it that synchronization is complete and the passive server now enters the *STANDBY* state. In this state, it receives messages replicated from the active server and applies them locally.

If the active server goes offline, only passive servers in the *STANDBY* state can be considered candidates to become the new active server.

Clearing Automatic Backup Data

After a passive server is restarted, for safety reasons, it may retain artifacts from previous runs. This happens when the server is restartable, even in the absence of restartable cache managers. The number of copies of backups that are retained is unlimited. Over time, and with frequent restarts, these copies may consume a substantial amount of disk space, and it may be desirable to clear up that space.

Backup rationale: If, after a full shutdown, an operator inadvertently starts the stripe members in the wrong order, this could result in data loss wherein the new active server initializes itself from the, possibly, incomplete data of a previous passive server. This situation can be mitigated by (1) ensuring all servers are running, and (2) the cluster is quiesced, prior to taking the backup. This ensures that all members of the stripe contain exactly the same data.

Clearing backup data manually: The old fast restart and platform files are backed up under the server's data directories in the format `terracotta.backup.{date&time}/ehcache/` and `backup-platform-data-{date&time}/platform-data` respectively. Simply change to the data root directory, and remove the backups.

It may be desirable to keep the latest backup copy. In that case, remove all the backup directories except the one with the latest timestamp.

3 Logical Server States

Possible server states

A server could be in any one of the following logical server states:

STARTING	server is starting.
UNREACHABLE	server is unreachable.
UNKNOWN	server state is unknown.
UNINITIALIZED	server has started and is ready for election.
SYNCHRONIZING	server is synchronizing its data with the current active server.
ACTIVE	server is active and ready to accept clients.
ACTIVE_RECONNECTING	server is active but waits for previously known clients to rejoin before accepting new clients.
ACTIVE_SUSPENDED	server is active but blocked in the election process (consistency mode).
PASSIVE	server is passive and ready for replication.
START_SUSPENDED	server startup is suspended for all of its peers to come up.
PASSIVE_SUSPENDED	server is passive but blocked in the election process (consistency mode).
DIAGNOSTIC	server is not activated (has no configuration) or has been asked to start in diagnostic mode. This mode is used to configure the nodes or repair them.

4 Clients in a Cluster

Within the overall structure of the cluster, the clients represent the application end-points. They work independently but can communicate through the *active* servers of the stripes to which they are connected.

Note that a client only ever interacts with an active server, never directly communicating with a *passive* server.

In a single-stripe cluster, each client is connected to the active server of that stripe. In a multi-stripe cluster, each client is connected to the active server of *each* stripe, interacting with them quasi-independently.

Within the logical structure of the cluster, the client isn't the process making the connection, but the connection itself. This means that a single JVM opening multiple connections to the same stripe will be seen by the stripe as multiple, independent clients.

How a client finds an active server

When establishing a connection to a stripe, the client must find the active server. It does this by attempting to connect to each server in the stripe, knowing that only the active server will not reject the connection attempt.

How a client handles failover or restart

If an active server to which a client is attached goes offline, the client will attempt to reconnect to one of the other servers in the stripe, if there are any. This is similar to what happens during its initial connection.

Note that there is no default time-out on this reconnection attempt. In the case that each stripe member is unavailable, this means that it is possible for all clients to wait, blocking their progress, until a server is restarted, potentially days later.

5 Electing an Active Server

When a new stripe comes online, the first thing the servers within it need to do is elect an *active* server which will coordinate the *client* interactions and *passive* servers within the stripe.

Additionally, if the active server of an existing stripe goes offline, the remaining passive servers need to elect a replacement active server. Note that only passive servers in the *STANDBY* state are candidates for this role. For related information, see the section "Failover" on page 23.

In either of these situations, the servers involved address this problem by holding an election.

A server that is started up from rest needs to get votes from all of its peer servers to get elected as an active server. In contrast, votes from a smaller set of peers are sufficient for a running *PASSIVE-STANDBY* server to become elected as an active server.

If for some reason, not all servers in a stripe can be started up, you can still forcefully get a candidate server elected as active using the cluster tool. For more information about this manual intervention using the cluster tool, see the section *The promote command* of the "Cluster Tool" on page 107.

High-level process

In an election, each server will construct a "vote" which it sends to the other involved servers. The vote with the highest score can be determined statically, so each server knows it has agreement on which server won the election.

In the case of a tie, the election is re-run until consensus is achieved.

Vote construction

The vote is a list of "weights" which represent the factors which should be considered when electing the most effective active server. The list is ordered in a way that the next element is only considered if the current element is a tie. This allows the earlier elements of the vote to be based around important concepts (such as how many transactions the server has processed), followed by concrete concepts (such as server up-time), then ending in more arbitrary concepts designed to break edge-case ties (such as a randomly generated number).

6 Failover

In a high-availability stripe, the failure of a single server represents only a small disruption, but not outright failure, of the cluster and the *client* operations (for related information on high availability, see the section "Cluster Architecture" on page 11).

In the case of a failing passive server, there is no disruption at all experienced by the clients.

In the case of a failing active server, however, there is a small disruption of client progress until a new active server is elected and the client can reconnect to it. *Failover* is the name given to this scenario.

Client Reconnect Window

When a failover happens, the clients connected to the previous active server automatically switch to the new active server. However, these clients have a limited window of time called the *client reconnect window* to complete the failover (120 seconds, by default). The new active server will stop processing any client requests until all the previously known clients connect back or until this window expires. This could cause all the clients to stall even if a single client fails or takes too long to fail over to the new active server.

If clients fail to connect back to the new active server within the reconnect window, the server will consider them unreachable and will continue processing requests from the connected clients. Clients reconnecting after the reconnect window will be rejected by the server and they will rejoin the cluster as a new client by establishing a new connection.

This reconnect window can be configured in the config file or during startup using the `client-reconnect-window` property.

Server-side implications

Once all clients have reconnected (or the reconnect window closes), the server will process all re-sent messages it had seen before for which the client had not been notified of completion.

After this, message processing resumes as normal.

Client-side implications

Clients will experience a slight stall while they reconnect to the new active server. This reconnection process involves re-sending any messages the client considers to be in-flight.

After this, client operations resume as normal.

7 Starting and Stopping the Terracotta Server

Starting the Terracotta Server

The command line script to start the Terracotta Server is located in the `server/bin/` directory of the server kit. UNIX users use `start-tc-server.sh` while Windows users use `start-tc-server.bat`. All arguments are the same for both.

Options to the script

1. `-port`

   Port to be used for this node. Default: `9410`.

2. `-bind-address`

   Bind address to be used for the node port. Default: `0.0.0.0`.

3. `-group-port`

   Port to be used for intra-stripe communication. Default: `9430`.

4. `-group-bind-address`

   Bind address to be used for node group port. Default: `0.0.0.0`.

5. `-hostname`

   Host name for this node. Must be a valid DNS name, or an IP address. Default: `%h`.

6. `-name`

   Name to be used for this node. Needs to be unique in the cluster. Default: a randomly generated string.

7. `-public-hostname`

   Public host name for this node. Needs to be set on all the nodes in the cluster. See "Terracotta in Network Environments with Subnets" on page 189 for more details about this feature.

8. `-public-port`

   Public node port for this node. Needs to be set on all the nodes in the cluster. See "Terracotta in Network Environments with Subnets" on page 189 for more details about this feature.

9. `-config-dir`

   Directory containing configuration bookkeeping information generated by the node. Default: `%H/terracotta/config`.

10. `-backup-dir`

    Directory to be used to contain cluster backups. Needs to be set on all nodes in the cluster. See the section "The "backup" Command" on page 112 of the cluster tool for more details about performing backups.

11. `-log-dir`

    Directory to be used to contain logs for this node. Default: `%H/terracotta/logs`.

12. `-metadata-dir`

    Directory to be used to contain server persistence data. Default: `%H/terracotta/metadata`. See the section "Server Persistence" on page 92 for more details about this setting.

13. `-repair-mode`

    Whether to start the node in repair mode. Default: `false`.

14. `-client-lease-duration`

    Lease duration for the client connections. Default: 150s. Needs to be the same throughout the cluster. See "Connection Leasing" on page 105 for more details about this property.

15. `-client-reconnect-window`

    Time window for clients to reconnect to a new Active server after failover. Default: 120s. Needs to be the same throughout the cluster.

16. `-cluster-name`

    Name to assign to the cluster this node will become a part of. Needs to be the same throughout the cluster.

17. `-config-file`

    Config file to start the node with. See the section "The TerracottaConfiguration File" on page 45 for more details about this property.

18. `-failover-priority`

    The failover priority setting to be used. Valid values are `availability` and `consistency`. Needs to be the same throughout the cluster. This is the only mandatory option if the node is started using console parameters. See "Failover Tuning" on page 99 for more details about this property.

19. `-data-dirs`

    Directory to contain client data. Default: `main:%H/terracotta/user-data/main`. Data directory names needs to be the same throughout the cluster - the disk locations could vary. See "Configuring the Terracotta Server" on page 91 for more details about this property.

20. `-offheap-resources`

    Offheap resources to be used. Default: `main:512MB`. Needs to be the same throughout the cluster. See "Configuring the Terracotta Server" on page 91 for more details about this property. Offheap resources is the only property which is validated against the license. The total `offheap-resources` for the cluster (i.e. offheap-resources for all stripes summed up) should be within the license limit. See "Licensing" on page 135 for more details about licensing.

21. `-audit-log-dir`

    Directory containing the node's security audit logs. Needs to be set on all the nodes. Needs some form of security to be enabled.

22. `-authc`

    Security authentication setting to be used. Valid values are `file`,`ldap` and `certificate`. Needs to be the same throughout the cluster.

23. `-security-dir`

    The security root directory for this node.

24. `-ssl-tls`

    Whether to enable SSL/TLS based security. Default: `false`. Needs to be the same throughout the cluster.

25. `-whitelist`

    Whether to enable IP whitelist security. Default: `false`. Needs to be the same throughout the cluster.

See "Cluster Security" on page 161 for more details about configuring security.

Use cases

The server startup script can be used in different ways, depending on the needs and convenience of the server admin:

Starting nodes with console parameters

A server can be started in unconfigured node, with console parameters alone (i.e. without the need of having any config files).

Example:

```
./start-tc-server.sh  -config-dir /data/tc/node1
   -port 9410 -group-port 9430  -name node1 -failover-priority consistency
   -offheap-resources primary:250GB,caching:100GB
```

Note: the config-dir in question must be empty, as it only makes sense to use the command with these parameters to load a new server with its initial set of configuration properties.

After startup, it will then be necessary to execute the config tool `attach` and `activate` commands respectively to define the cluster topology and to make the cluster ready for client operations.

Before activating, the config tool could also be used to set various other configuration options on the node.

Starting nodes with config file

An unconfigured server can be also be started with a parameter specifying a config file which was exported from an existing cluster, or constructed by hand. See the section "Export" on page 68 for related information.

Example:

```
./start-tc-server.sh -config-dir /data/tc/node1
  -config-file myCluster.properties
```

In this case it is not necessary to execute the config tool `attach` command since the cluster topology is already defined in the config file. The execution of the `activate` command, however, would still be required to make the cluster ready for client operations.

Note:
The config-dir in question must be empty, as it only makes sense to use the command with these parameters to load a new server with its initial set of configuration properties.

Note:
It is important to understand that the configuration file used in this command is not what will be used by the server each time it starts. Rather, it is simply a convenience for passing the initial configuration properties to the server, instead of specifying each config property on the command line and/or setting it via config tool. Once the config file is read, the server's internal configuration is stored in its config-dir. The config-dir contains the configuration that will be used on subsequent restarts.

Starting nodes with config dir

A previously configured server can be started with the specification of a configuration directory (config-dir), in activated mode directly.

Example:

```
./start-tc-server.sh  -config-dir /data/tc/node1
```

This is used when the server was restarted after having been activated previously, but can also be used on a fresh server startup after completing the migration process from an older Terracotta cluster. See the sections "Migrating from older Terracotta versions to 10.7" on page 145 or "Migrating from 10.7 to a newer 10.7 version" on page 147 for related details

Note that when you start a server this way (which will be the most common way, over time), any other parameters passed will be effectively ignored, because the server will use its internal configuration.

After starting a node this way, it is not necessary to run the config tool `attach` or `activate` commands since the cluster topology and configuration are ready.

Environment variables read by the script

- `JAVA_HOME` - Points to the JRE installation which the server should use (the Java launcher in this JRE will be used to start the server).
- `JAVA_OPTS` - Any additional options which should be passed to the underlying JVM can be passed via this environment variable and they will be added to the Java command line.

Stopping the Terracotta Server

If your server is not running in a Terracotta cluster, you can use the standard procedure offered by your operating system to terminate the server process.

If you are running the server as part of a Terracotta cluster, you can safely shut down all servers in the cluster using the cluster tool. See the section "Safe Cluster Shutdown" on page 31 of the cluster tool for details.

8 Safe Cluster Shutdown and Restart Procedure

Although the Terracotta Server Array is designed to be crash tolerant, like any distributed system with HA capabilities, it is important to consider the implications of shutting down and restarting servers, what sequence that is done in, and what effects that has on client applications and potential loss of some data.

The safest shutdown procedure

For the safest shutdown procedure, follow these steps:

1. Shut down all clients and ensure no critical operations such as backup are running on the cluster. The Terracotta client will shut down when you shut down your application.

2. Use the `shutdown` command of the cluster tool to shut down the cluster.

If you want to partially shut down a stripe with passive servers configured, you can use the partial shutdown commands provided by the cluster tool. See the section "Cluster Tool" on page 107 for details.

The safest restart procedure

To restart a stripe for which the failover priority is *consistency*, servers can be started up in any order as it is guaranteed that the last active server is re-elected as the active server, thus preventing data loss. This is guaranteed even if there are multiple former active servers in the stripe at the time of shutdown (for example, one active server and one or more suspended active servers or former active servers that were shut down, decommissioned or had crashed).

However, if the failover priority is *availability*, restarting the servers in any random order might result in data loss. For example, if an older active server is started up before the last active server, it could win the election and become the active server with its old data. To avoid such data loss scenarios, the last known active server must be restarted first. All other servers must be started up after this last known active server becomes the active server again.

However, if you do not know the most recent active server at the time of restart and still want to restart the stripe safely without data loss, it can still be done by starting all the servers in that stripe using the `--consistency-on-start` option of the server startup script. When the servers are started up using this option, they will wait for all peer servers to come up and then elect the most recent active server as the new active server.

If there are multiple active servers at the time of shutdown, which can happen if the failover priority of the cluster is *availability*, one of them will be chosen automatically on restart. This choice is made based on factors like the number of clients connected to those servers at the time of shutdown, the server that was started up first, etc.

Considerations and implications of not following the above procedure

Facts to understand:

- Servers that are in "active" status have the "master" or "source of full truth" copy of data for the stripe they belong to. They also have state information about in-progress client transactions, and client-held locks.
- Mirror servers (in "passive standby" state) have a "nearly" up to date copy of the data and state information. (Any information that they don't have is redundant between the active server and the client.)
- If the active server fails (or is shut down purposely), not only does the standby server need to reach active state, but the clients also need to reconnect to it and complete their open transactions, or data may be lost.
- A Terracotta Server Array, or "Cluster" instance has an identity, and the stripes within the TSA have a "stripe ID". In order to protect data integrity, running clients ensure that they only "fail over" to servers with matching IDs to the ones they were last connected to. If cluster or stripe is completely "wiped" of data (by purposely clearing persisted data, or having persistence disabled and having all stripe members stopped at the same time), that will reset the stripe ID.

What happens if clients are not shut down

If clients are not shut down:

- Client applications will continue sending transactions (data writes) to the active server(s) as normal, right up until the active server is stopped. This may leave some successful transactions unacknowledged, or falsely reported as failed to the client, possibly resulting in some data loss.
- Clients will continue to try and connect and when the server is restarted, the clients will fail the current operation and enter a reconnect path to try and complete the operation. When clients enter a reconnect path, it is left to the client to ensure idempotency of the ongoing operation as the operation might either have been made durable just before shutdown or it may have been missed during shutdown.

What happens if the active server is shut down explicitly

If the active server is shut down first:

- Before shutting down any other servers, or restarting the server, ensure that you wait until any other servers in the stripe (that were in 'standby' status) have reached *active* state, and that any running clients have reconnected and re-sent their partially completed transactions. Otherwise there may be some data loss.

9 Configuration Terms and Concepts

To have a solid grasp of how to configure a Terracotta Server Array (TSA), one must first have a strong understanding of the basic concepts of what a TSA is, what it uses as resources, and how its configuration system works.

Review of Terracotta Server Array Concepts

As a quick review of high-level TSA concepts:

- A Terracotta Server Array (TSA) is composed of one or more "stripes".
- A stripe is composed of one or more Terracotta Servers
- Each stripe contributes to the total storage and computing capacity of the TSA. If there are five stripes, then each one will contain roughly one-fifth of the stored data.
- Within a stripe, one server is "active" (serves workload from clients), and any others act as "mirrors" for HA purposes.
- Because any member of the stripe may be elected as the "active" server, the configuration and system resources of all stripe members must be equivalent.

Stripes have names (which can be assigned during configuration time), and nodes (servers) that are members of the stripe also have names. These names are useful when "targeting" configuration or operational commands.

For more information on the above concepts, please review the sections "Cluster Architecture" on page 11 and "Active and Passive Servers" on page 13.

Server Resource Concepts

Terracotta Servers utilize resources in order to provide services and features such as network connectivity, data storage, durability, backups, etc.

Notable items that need to be configured (or considered whether the default value is appropriate) include:

- Network ports - This includes a "port" for receiving client requests, and a "group-port" for communicating with other stripe members (servers). The default values are `9410` and `9430` respectively.

- Server metadata directory - This is a directory where the server stores important metadata about its internal state. The default location is `<user-home-dir>/terracotta/metadata`.
- Configuration directory - This is a directory where the server stores its internal configuration. The default location is `<user-home-dir>/terracotta/config`.
- Offheap storage resources - Servers need one or more offheap (memory) resources defined in order to have space in which to store data (via Caches or Datasets). For proper operation, all servers in a cluster (TSA) need to have the same set of offheap resources defined (because Cache and Dataset configurations will reference them for use). The default is to create one offheap resource named `main` with size `512MB`.
- Data directories - Optional, but commonly used, data directories are used for durable (persistent) storage of data. For proper operation, all servers in a cluster (TSA) need to have the same set of data directories defined (because Cache and Dataset configurations will reference them for use). The default is to create one data directory named `main` with location `<user-home-dir>/terracotta/user-data/main`.
- Backup directory - Used as the destination for backups.
- Logging directory - Used as the destination for server logs. Default location is `<user-home-dir>/terracotta/logs`.
- Failover priority - a cluster-wide setting that affects HA behavior when nodes are shut down or fail. A choice must be made as to whether the cluster should favor `availability` of service or `consistency` of data when situations occur that could lead to split-brain scenarios (e.g. when servers are still running but cannot communicate with each other).

For more information on about these items, please review the sections "Config Tool" on page 51, "The TerracottaConfiguration File" on page 45, and "Configuring the Terracotta Server" on page 91.

Configuration Concepts

Perhaps the most important thing to understand about a Terracotta server's configuration, is that it is stored and updated "internally" to the server, not in a human-editable file that is read each time the servers starts.

After a node has been configured and is running, everything that is needed to restart it, and get it running again (with the same configuration and internal state) is stored within the node's config-dir.

The mechanisms for adding to or changing the internally stored configuration of servers is therefore the focus of what needs to be understood next.

Fundamental, Required Settings

There are a few very fundamental items related to a server instance (node) that are necessary for its existence. This includes: network ports, config-dir and metadata-dir. The ports are of course used to make the node accessible. The config-dir is where the server will store (and later find) its internal configuration. And the metadata-dir is where the server will store (and later find) its internal state information.

Necessarily Equivalent Settings

Some configuration settings need to be consistent or equivalent across all nodes in a stripe and/or across all nodes in all stripes of a cluster. The reasoning for this is fairly clear and logical, if we give a few examples.

Settings that need not be (or, in some cases must not be) the same across nodes include things such as the node name. Clearly, the node name must be unique, or it wouldn't be useful for identification of the node. The network port can be any legal and available port number, and there is no need for any two servers to use the same port, though it probably is most clear if they all use the same port. (Note that they obviously must not use the same port if they are on the same host, but for other reasons (resources, and HA) it is strongly recommended not to run servers on the same host).

Some examples of settings that need to be equivalent across nodes are offheap resources and data directories. This is because these are referenced (by name) in configurations for Datasets and Caches, and are expected to exist on all servers. For example, if a user configures a new Dataset to utilize (store its data in) an offheap resource named 'primary', then, as the Dataset is created on each server member of the cluster, an offheap resource named 'primary' must be found on each, or the creation of the dataset will fail. It also makes sense that the offheap resource named 'primary' should have an identical size on each server, such that mirrors can hold a copy of all the same data the active server has, etc. For data-dirs, it is similar: when a Cache or Dataset configuration instructs the usage of a data-dir, one with that name must exist on each server in the TSA. (However, in the case of a data-dir, while the data-dir name needs to be known by all servers, the file path that it refers to does not have to be *identical* on all servers. Hence, we say that all server nodes should have the *equivalent* set of data-dirs.)

Initial Configuration Steps

The typical steps for initial configuration of a TSA are:

1. Start each server node (as unconfigured servers, they will enter 'diagnostic mode', and await configuration)
2. Provide each server with configuration settings
3. Attach nodes to each other to form stripes
4. Attach stripes to each other to form a cluster
5. Activate the cluster

The first two steps can be accomplished in one command-line, if the user specifies configuration settings as parameters to the `start-tc-server` script.

All of the steps can be accomplished in one command-line, if the user specifies a config file containing all of the settings for all nodes of the cluster. Note that such a file is only used to initialize the set of configuration properties in each node's internal configuration (stored in its config-dir) - it is never read or utilized again.

After these steps, the server nodes will restart themselves (to leave diagnostic mode) and form the configured cluster. As the servers restart, they will use the configuration stored in their config-dir. With any subsequent restarts of a node, the user must specify (to the `start-tc-server` script) the location of the config-dir (or the script will attempt to use the default location).

Once the cluster is activated, some configuration properties (such as node name and config-dir) cannot be changed. Others can be changed or added later, as necessary (such as offheap resources).

Understanding the Configuration Directory and Config Tool

Configuration Directory

As previously noted, a server node's `config-dir` is where its internal storage, or "source of truth", for configuration is kept. The files under this directory are *not to be edited by the user*, as (for reasons that will be made more clear below) they are solely managed by the server node itself.

If you are restarting a server process, and want it to be the same server node that it was before, you need to ensure that the `start-tc-server` script specifies (or defaults to) the appropriate `config-dir`.

Config Tool

The config tool (see "Config Tool" on page 51) is used to add or modify configuration settings for servers, both before they are activated as part of a cluster, and afterward. It can also be used to see what a server's current configuration settings are, or export them for use as a template or backup for recreating clusters.

The config tool connects to server nodes and issues commands, such as to set a configuration property. Some config tool commands may target a single node, while others may target all nodes of a stripe, or all nodes of a cluster. In all cases, the server responds to the config tool's requests by reading and/or updating the configuration state files contained in the server's `config-dir`.

Configuration Operations and Outcomes

As noted in the previous paragraphs, a server node responds or reacts to config-tool requests by reading or updating the contents of its internal configuration (which is contained in the `config-dir`, once the cluster has been activated).

Because many configuration settings must be the same on all member nodes of a stripe and some must be the same on all members of the cluster, changes to configuration must be coordinated, and therefore complex outcomes are possible.

For example, if the number of "voters" for a stripe is to be changed, it is only safe for that change to go into effect if it does so at the same time on all nodes in the stripe. Otherwise, "bad things" could happen when a failover situation occurs (e.g. one node may make a bad decision as to whether or not it should move to "active" state.) Similarly the adjustment of offheap resource sizes, or the attachment of an additional node to a stripe needs to be synchronized across servers.

This coordination, or synchronization is accomplished via a two-phase commit protocol, wherein configuration changes are staged and validated on each server (to determine that the change can be successful on all servers), and then committed (or activated) on each server in the second phase.

Typically, (in most cases), the config tool and the server's internal configuration manager handle the complexities of the coordination just fine. However, if a failure occurs during a configuration change, or one of the nodes is not running when a configuration change is made, there are possible outcomes that may require follow-up on the user's part.

Most of the non-typical cases are automatically corrected by the servers, such as when a node restarts (if it was down during the configuration change, or if it crashed after the configuration change was staged but before it was committed or rolled back). When servers that are restarting connect to the other members of their stripe, they discover whether their configuration state is out of sync, and if so, then they receive the appropriate updates from the other server(s).

Very rarely the config tool's `repair` command may need to be used to force a commit or rollback of a config change after careful inspection of the configuration state via the config tool's `get` or `export` commands.

In all cases the config tool will inform you about the success or failure of a configuration operation, and hint to you any next steps that may be necessary.

10 Planning a Configuration

To be successful, most deployments require at least a little planning before beginning the configuration process in order to avoid missteps or even the need for starting over.

If you have not already done so, please familiarize yourself with the material presented in "Configuration Terms and Concepts" on page 33.

Naming and Addressing

Naming Servers and Stripes

Because Terracotta deployments typically involve at least two servers, and often very many more, you should put a little planning into how you name them. Doing so will help keep the nodes and stripes easily identifiable within configuration and management commands and monitoring views. It is not actually necessary to name them, in which case they will be assigned auto-generated names, but using names that are meaningful to you will likely be helpful.

Some things to consider when deciding upon the scheme for naming stripes and nodes:

- You may want to include within the name of a stripe or node something that hints at its purpose, such as whether it is part of a development, test, or production environment. For example, "DevStripe-A".
- You may want to include within a node's name something related to the name of the host upon which the server runs. On the other hand, in dynamic/container environments you may want to purposely avoid this.
- If you expect that you'll be changing your TSA topology in the future (i.e. adding or removing stripes from the cluster, or adding or removing servers from stripes), you may want to purposely avoid using sequential numbering in the names, such as "server-1" or "stripe-1", because over time you may end up with gaps or other oddities in the numbering. This will actually work fine, but may be confusing to users who try to assemble within their minds a mental map of the topology.

As you form your cluster, the cluster itself can also be named. It makes good sense to use a name that clearly identifies its purpose, e.g. "MyApp-PROD-TSA" or "MyApp-DEV-TSA", etc.

Addressing Servers

As you plan the set of servers that you will need, and where they are to be deployed, it would be wise to make, and keep handy during the configuration process, a clear listing of which host names (or addresses) and ports will be used for each server.

You have the choice of addressing servers by hostname or by IP address. Using host names (that can be resolved by DNS) is favorable. Note that you can also specify `bind-address` (for `port`) and `group-bind-address` (for `group-port`) in case of any ambiguity of which IP address the ports will be opened on (with the default being to open the ports on all of the host's addresses).

Data Consistency and Availability

One of the most important actions in planning your configuration is that of determining which guarantees you would like to favor in the case of a fail-over situation.

Please refer to "Failover Tuning" on page 99 for a full discussion of that feature.

The well-known CAP Theorem causes the need for a choice in which guarantees the TSA should sacrifice in order to preserve the others, in the case of a fail-over situation.

If you plan to store data in the TSA and have its integrity protected with priority, you should strongly consider using the `consistency` setting for the cluster's `failover-priority` setting.

If you plan only to cache data in the TSA, you may prefer to use the `availability` setting for `failover-priority`.

In either case, the likelihood of the cluster ever needing to resort to compromising on either data availability or data consistency can be greatly reduced by careful choices and resourcing related to High Availability.

High Availability

High Availability (HA) of the Terracotta Server Array is achieved through the use of mirror servers within each stripe, and the optional use of "voters".

When an active server is shut down or fails, other stripe members become eligible for becoming the new active server for that stripe's set of data. If there are no other members of the stripe running, then the stripe's data is not available, and that typically results in the complete unavailability of the Terracotta cluster, until the stripe is back online.

As you plan your TSA configuration, you should consider what levels of service are required, and plan the proper number of servers per stripe and any requisite external voters (to assist with tie-breaking during elections when quorum is not otherwise present).

For more information on these topics, see "Active and Passive Servers" on page 13, "Electing an Active Server" on page 21, and "Failover Tuning" on page 99 (including discussion of External Voters).

Storage and Persistence Resources

In-Memory Storage

As part of planning for your configuration, you need to put some thought into how you will organize the storage of your data.

Typically, data is stored within "offheap resources" which represent pools of memory reserved from the underlying operating system. You configure one or more offheap resources, giving each a name and a size (such as 700MB or 512GB, etc.). After your cluster is up and running, you can create Caches and Datasets for storing your data, and as you do so, you will need to indicate which offheap resource will be used by each.

There is nothing inherently wrong with simply defining only one offheap resource and having all Datasets and Caches us it. However, some users may find it useful to be sure particular amounts of memory are reserved for particular Datasets or Caches.

Note that the total amount of memory for a particular offheap resource is actually the configured size of the resource multiplied by the number of stripes in the TSA, because the configured amount is for a given server. Thus if you configure an offheap resource name 'primary' with a size of 50GB, and your TSA has 3 stripes, then you will be able to store a total of 150GB of data (including any related secondary indexes) within the 'primary' offheap resource.

See also the topic *Necessarily Equivalent Settings* in the section“Configuration Terms and Concepts” on page 33.

Disk Storage and Persistence

Most users desire to have at least some sets of their data persisted, or in other words, durable between restarts of the servers in the TSA. Terracotta's FRS and Hybrid features provide such capability.

FRS (Fast Restartable Store) is a transaction log structured in such a way that it can be very efficiently replayed upon server restart, in order to recover all of the stored data as it existed when the server went down. (Note that passive/mirror servers would instead sync the latest state of the data from the active server). When FRS is enabled, data writes (additions, updates, deletions) are recorded in FRS, but all data reads (gets and queries) occur within memory.

Hybrid storage mode utilized FRS capabilities, but also expands storage capacity to include the disk, not just memory. In Hybrid mode, memory (offheap resources) is used to store keys, pointers/references and search indexes (for extremely fast resolution of lookups and queries), but values are read from disk, such that memory does not need to have the capacity to contain them all. Like FRS, all data is recovered in its last state when the server restarts.

For your configuration planning, you should note that both FRS and Hybrid features require a location on disk where they can store the data. Because data is written to disk when modifications occur, the speed of the disk is a major factor on the latency and throughput of Dataset and Cache operations, and the speed of server restarts. Many users find it beneficial to dedicate a highly performant file system for FRS/Hybrid data, while having the server use a different file system for storing configuration, logs, etc. Some users find it useful to have multiple file system paths (e.g. mount points) for storing different sets of data (different Caches or Datasets) both for performance and organizational (e.g. for backups) purposes.

Locations for user data storage are specified with the `data-dirs` configuration property, which can contain a comma-separated list of one or more data directories. Each data-dir has an identifying name, that is used in the configuration of Caches and Datasets to enable persistence of the data that is put into them. Recall that an equivalent set of data-dirs (with the same names), should exist on all nodes of the cluster.

Your planning should consider what filesystem path(s) you will use for data persistence (if any), and what names you would like to identify each of those locations with.

See also the topic *Necessarily Equivalent Settings* in the section"Configuration Terms and Concepts" on page 33.

Config and Metadata Directories

Terracotta servers require locations for storing their internal configuration (set with the `config-dir` property), and their state metadata (set with the `metadata-dir` property).

For each server instance you'll want to make sure that the locations of these are always available to the server (perhaps ideally on a local disk).

You may want to plan to name the directories after the server node's name, or similar - in order to help keep things organized and clear for yourself and others who administer the system.

Backup Directory

In order to use the data backup feature of Terracotta, you will need to configure a location for the backup to be written to. This is done with the `backup-dir` config property.

The location should ideally be performant (such that the backup files can be written quickly, with minimal impact to the server), and large enough to contain the backup to be made, plus any other backups that you may have previously made and not removed.

See also: "Backup, Restore and Data Migration" on page 137.

Logging Directory

You should also put some planning into where your server's logs will be written. This is configured with the `log-dir` property.

Like the server's metadata directory and config directory, the log directory should be available to the server at all times, and you likely want to ensure that its path and name make clear sense to you and others who will be administering the system, as to which server's logs the directory contains.

Security

Your configuration planning should also consider whether you wish to enable security features on your cluster. Security features include encryption of network communications via TLS/SSL, and authentication, authorization and auditing (AAA) features.

If so, you will need to become familiar with these features to properly plan your configuration. See also: "Security Core Concepts" on page 156 and "Cluster Security" on page 161.

Public Addresses

Will the clients need to address the servers differently than the servers address each other (such as due to being within a managed container environment that has an "internal" network)?

If so, you may want to review whether hostnames will resolve to legal addresses both inside and outside of the containers, and whether you need to use the `public-address` configuration setting on your servers.

See also: "Terracotta in Network Environments with Subnets" on page 189.

11 The TerracottaConfiguration File

This document describes the elements of the Terracotta configuration file (a.k.a. *config file*), which serves to define the configuration of a Terracotta cluster including its stripes and nodes. Refer to " Cluster Architecture" on page 11 for details on different TSA topologies.

A sample config file is provided in the kit under `server/conf`. It can be used as the starting point. Some entries in the file have inline comments describing the configuration elements. Be sure to start with a clean file for your configuration.

Use cases

Unlike previous Terracotta releases, a config file isn't needed to configure or start the servers. However, a config file is handy for the following purposes:

1. "Export" on page 68 and viewing the configuration of a given cluster.
2. Backing up and version-controlling the configuration of a given cluster.
3. Using the current configuration as a foundation to build up a new configuration.
4. "Import" on page 70 the cluster configuration on running nodes of an unconfigured cluster.
5. "Starting nodes using the exported config" on page 25 in a new cluster.

The config file is only used as a convenience to *feed* the configuration settings into unconfigured servers. Once that is performed, the servers will never again utilize or reference the config file, but will instead utilize their internal configuration (which is persisted in the server's `config-dir`).

Configuration File Format

To help illustrate the format of the config file, consider a cluster that comprises two stripes and four nodes. Nodes **node1** and **node2** belong to **Stripe-A**. Nodes **node3** and **node4** belong to **Stripe-B**.

The configuration file format requires that all stripes defined for the cluster are declared with the **`stripe-names`** property:

```
stripe-names:<stripe_name_1>,<stripe_name_2>,...,<stripe_name_n>
```

The declaration of stripes for our example is:

```
stripe-names:Stripe-A,Stripe-B
```

The configuration file format also requires that all nodes belonging to a stripe are declared with the **node-names** property:

```
<stripe_name_1>:node-names:<node_name_1-1>,<node_name_1-2>,...,<node_name_1-n>
<stripe_name_2>:node-names:<node_name_2-1>,<node_name_2-2>,...,<node_name_2-n>
...
<stripe_name_m>:node-names:<node_name_m-1>,<node_name_m-2>,...,<node_name_m-n>
```

The declaration of nodes for our example is:

```
Stripe-A:node-names:node1,node2
Stripe-B:node-names:node3,node4
```

With the node and stripe names declared, settings are defined at cluster, stripe or node level by referencing the appropriate stripe name, node name or no name (in the case of cluster level setting):

1. Node-level scope:

   ```
   <node_name>:<setting>  OR  node:<node_name>:<setting>
   ```

2. Stripe-level scope:

   ```
   <stripe_name>:<setting>  OR  stripe:<stripe_name>:<setting>
   ```

3. Cluster-level scope:

   ```
   <setting>
   ```

4. When the same setting appears multiple times in the config file and that setting is directly or indirectly ascribed to the same node, lower scoped entries will take precedence over higher scoped entries when determining that setting's value for that particular node, where *node-scope* < *stripe-scope* < *cluster-scope*.

   For example, consider this snippet:

   ```
   backup-dir=pathA
   node2:backup-dir=pathB
   StripeB:backup-dir=pathC
   ```

   The above snippet will effectively impart the following configuration to the cluster:

   ```
   node1:backup-dir=pathA
   node2:backup-dir=pathB
   node3:backup-dir=pathC
   node4:backup-dir=pathC
   ```

   In the above example, the cluster-scoped entry `backup-dir=pathA` is applied to every node in the cluster. But the lower scoped entry `node2:backup-dir=pathB` overrides the cluster-scoped entry for **node2** and assigns its backup directory as **pathB**. Similarly, the stripe-scoped entry `StripeB:backup-dir=pathC` overrides the backup directory which **backup-dir=pathA** would have ascribed to nodes **node3** and **node4**, with the value **pathA**.

Examples

This section describes the various properties supported in a config file.

Minimal configuration

A config file can get pretty large, especially when the cluster contains a large number of nodes. However, most configuration properties have default values, because of which the config file can be reduced in size if the default values are acceptable. The only mandatory property for a single-node cluster is:

```
<node_name>:hostname=localhost
```

which specifies the `hostname` to `localhost`. The following default values are used:

Property	Default value	Comments
offheap-resources	main:512MB	Defines one "offheap resource" on page 91 with name `main` and size `512MB`.
client-lease-duration	150 seconds	Defines the "lease duration" on page 105 for the client connections as 150 seconds.
client-reconnect-window	120 seconds	Defines the client reconnect time window as 120 seconds.
<node_name>:hostname	%h	Sets the host name of the node to the fully-qualified host name of the machine.
<node_name>:port	9410	Sets the port for this server process to 9410.
<node_name>:bind-address	0.0.0.0	Sets the bind address for the port to the wildcard address 0.0.0.0
<node_name>:group-port	9430	Sets the intra-stripe communication port for this server process to 9430.
<node_name>:group-bind-address	0.0.0.0	Sets the bind address for the group port to the wildcard address 0.0.0.0
<node_name>:metadata-dir	%H/terracotta/metadata	Sets the "server persistence" on page 92 directory to `%H/terracotta/metadata`.
<node_name>:data-dirs	main:%H/terracotta/user-data/main	Defines a "user data directory" on page 91 with name `main` and path `%H/terracotta/user-data/main`.
<node_name>:log-dir	%H/terracotta/logs	Sets the server logging directory to `%H/terracotta/logs`

Note:

`%h` and `%H` in the above default values point to the hostname of the machine and home directory of the current user respectively. See the section "Parameter Substitution" on page 89 for more information.

Several other configuration properties are omitted (i.e. not assumed to have defaults), which are:

Property	Comments
cluster-name	The name of the cluster
whitelist	Whether to enable IP whitelist security
ssl-tls	Whether to enable SSL/TLS based security
authc	Security authentication setting to be used
security-dir	The security root directory for this node
audit-log-dir	Directory containing the node's security audit logs
backup-dir	Directory to be used to contain the "backup" on page 137 of this node
public-hostname	"Public hostname" on page 189 for this node
public-port	"Public port " on page 189for this node

Security configuration

The following snippet demonstrates how to enable IP whitelisting and SSL/TLS based "security" on page 155 along with security event auditing on a single node cluster (with node named **node1**):

)

```
failover-priority=availability
whitelist=true
ssl-tls=true
authc=certificate
node1:hostname=localhost
node1:audit-log-dir=/path/to/audit/dir
node1:security-dir=/path/to/security/dir
```

High availability configuration

High-availability can be enabled by configuring more than one node in a stripe. The following snippet defines two nodes in a cluster containing a single stripe:

```
failover-priority=availability
node1:hostname=localhost
node1:port=9410
node1:group-port=9430
node2:hostname=localhost
node2:port=9510
node2:group-port=9530
```

Multistripe with HA configuration

Stripes can be added to a Terracotta cluster to scale it out. Additionally, high-availability in a stripe can be enabled by configuring more than one node. The following snippet defines a cluster with two stripes with two nodes each:

```
failover-priority=availability
node1:hostname=localhost
node1:port=9410
node1:group-port=9430
node2:hostname=localhost
node2:port=9510
node2:group-port=9530
node3:hostname=localhost
node3:port=9610
node3:group-port=9630
node3:hostname=localhost
node3:port=9710
node3:group-port=9730
```

The preceding snippet can be simplified:

```
failover-priority=availability
hostname=localhost
node1:port=9410
node1:group-port=9430
node2:port=9510
node2:group-port=9530
node3:port=9610
node3:group-port=9630
node4:port=9710
node4:group-port=9730
```

12 Config Tool

Overview

The Config Tool is a command-line utility typically used by administrators of the Terracotta Server Array. It is used to perform a variety of cluster management tasks. The tasks are carried out by executing one or more Config Tool commands against Terracotta servers.

The Config Tool is executed by running the appropriate `config-tool` script located in the `tools/bin` folder inside the Terracotta installation directory:

- `config-tool.bat` - used on Windows platforms
- `config-tool.sh` - used on Unix/Linux platforms

Config Tool script executions utilize the following syntax:

```
config-tool.sh|bat [<common_options>] <command> <command_specific_options>
```

Note that `common_options` are optional, while one or more of the `command_specific_options` are required.

Commands

The supported Config Tool commands include:

Command	Description
"activate" on page 59	Activate unconfigured nodes in a cluster.
"attach" on page 61	Attach nodes to a stripe; attach stripes to a cluster.
"detach " on page 62	Detach nodes from a stripe; detach stripes from a cluster.
"get" on page 64	Read the value of one or more cluster, stripe or node settings in the cluster.
"set" on page 65	Write a value for one or more cluster, stripe or node settings in the cluster.
"unset" on page 67	Remove a previously set value for one or more cluster, stripe or node settings in the cluster.
"export" on page 68	Export the complete cluster definition to a configuration properties file.
"import" on page 70	Import a configuration properties file containing the definition of a cluster and its associated stripes and node.
"diagnostic" on page 71	Retrieve detailed status information for all nodes comprising the cluster.
"repair" on page 74	Repair a node within a cluster which is in an incorrect state.
"log" on page 74	Retrieve details about all changes made to a node in the cluster.

Refer to the "Configuration Terms and Concepts" on page 33 section to gain a deeper understanding of what the Config Tool commands do and under what circumstances you might use them.

Important:
Refer to the "Config Tool Troubleshooting Guide" on page 75 when encountering errors that prevent the desired command from executing successfully.

Common Command Options

Each Config Tool command supports a unique set of options (detailed in the sections throughout this document). But all commands support the following common options:

Option	Description	Default
-connection-timeout	Timeout value for connections to be established.	10 seconds
-request-timeout	Timeout value for command requests to be executed.	10 seconds
-security-dir	Specifies the location of the security root directory folder. Used to communicate with a server that is configured with any of the supported security schemes (e.g. TLS/SSL). For more details on configuring security in a Terracotta cluster see "Security Core Concepts" on page 156 and "Cluster Security" on page 161.	
-verbose	Generates a verbose output. Useful for debugging error conditions.	false
-help	Displays help information for commands and their options.	

Example Cluster

The Config Tool usage examples used throughout this document reference the example cluster shown below.

- This three-stripe cluster utilizes HA with **Stripe-A** and **Stripe-C** employing a single mirror (**node2** and **node7** respectively) and **Stripe-B** employing two mirrors (**node3** and **node4**).
- Each node is configured to operate on a distinct port. This is only for illustrative purposes in order to provide clarity when reading the Config Tool commands. In fact, multiple Terracotta servers - each running on a distinct host - may be configured to use the same port, including the default port (`9410`).

```
| STRIPE: Stripe-A |
+-----------------+-----------------+-----------------+---------------+
|   Node Name     |   Host-Port     |   IP Address    +    Status     |
+-----------------+-----------------+-----------------+---------------+
|     node1       |   host1:9410    |    10.0.0.1     +    ACTIVE     |
----------------------------------------------------------------------
|     node2       |   host2:9412    |    10.0.0.2     +   PASSIVE     |
+-----------------+-----------------+-----------------+---------------+
| STRIPE: Stripe-B |
+----------------+----------------+----------------+--------------+
|   Node Name    |   Host-Port    |   IP Address   +    Status    |
+----------------+----------------+----------------+--------------+
|     node3      |   host3:9413   |    10.0.0.3    +   PASSIVE    |
-----------------------------------------------------+---------------
|     node4      |   host4:9414   |    10.0.0.4    +   PASSIVE    |
-----------------------------------------------------+---------------
|     node5      |   host5:9415   |    10.0.0.5    +    ACTIVE    |
+----------------+----------------+----------------+--------------+
| STRIPE: Stripe-C |
+----------------+----------------+----------------+--------------+
|   Node Name    |   Host-Port    |   IP Address   +    Status    |
+----------------+----------------+----------------+--------------+
|     node6      |   host6:9416   |    10.0.0.6    +    ACTIVE    |
-----------------------------------------------------+---------------
|     node7      |   host7:9417   |    10.0.0.7    +   PASSIVE    |
+----------------+----------------+----------------+--------------+
```

Namespace Syntax

Many of the settings permit their values to be accessed (read/write via get, set and unset commands) at different topology levels (i.e. cluster, stripe or node). This is achieved through the use of a `namespace` syntax where the *name* of the topology entity is used to qualify the setting's access level:

```
[namespace].<setting> --> [(stripe:<stripe_name>: | node:<node_name>:)]<setting>
or
[namespace].<setting> --> [(<stripe_name>: | <node_name>:)]<setting>
```

The following examples illustrate this namespace usage. For additional details describing this namespace syntax refer to "The Terracotta Configuration File" on page 45.

- Access **host7**'s `port` setting. The name of host7 is node7:

  ```
  node:node7.port   or   node7:port
  ```

- Access the backup directories (`backup-dir`) for all nodes belonging to **Stripe-B**:

  ```
  stripe:Stripe-B:backup-dir   or   Stripe-B:backup-dir
  ```

- The log directories (`log-dir`) for all nodes in the cluster is accessed by NOT including any namespace before the setting:

  ```
  log-dir
  ```

- If a cluster has a node and stripe possessing the same name (e.g. **SERVER-A**), then accessing a setting at the stripe or node level must incorporate the *stripe*: qualifier or *node*: qualifier, otherwise the request would be ambiguous. For example:

  ```
  node:SERVER-A:backup-dir   // the backup directory for node: SERVER-A
  stripe:SERVER-A:backup-dir        // the backup directory for all nodes belonging
   to stripe: SERVER-A
  SERVER-A:backup-dir                // ERROR: because SERVER-A is ambiguous when
  not qualified
  ```

Settings

A cluster topology is defined and behaves according to the configuration of the cluster, its stripes, and their constituent nodes. The configuration of each of these entities is established by ascribing values to one or more of their supported *settings*.

Each setting has the potential to be queried (via the `get` command), modified (via the `set` command) and in some cases 'removed' or 'undone' (via the `unset` command). Importation of settings (via the `import` command) is another avenue to configure an entity.

Each setting possesses a unique set of rules governing how it can be changed. These rules depend upon the setting's applicable *scope* (i.e. cluster, stripe, or node) and whether the entity is CONFIGURED (i.e. Activated) or UNCONFIGURED.

Configuration Rules for UNCONFIGURED (not Activated) Nodes and Clusters

An unconfigured cluster offers the greatest flexibility where modifying settings is concerned. This is because the settings of UNCONFIGURED nodes can always be modified and changing them will never require a restart. However, once a node is activated, certain settings can not be modified and in some cases, modifying a node's settings will require the node to be restarted.

The following table shows which Config Tool commands can be executed against each supported setting when the node is UNCONFIGURED (i.e. not Activated).

Setting	Node	Stripe	Cluster	Default
`license-file`			`set unset`	-
`cluster-name`			`get set unset import`	-
`client-reconnect-window`			`get set unset import`	`120 seconds`
`client-lease-duration`			`get set unset import`	`150 seconds`
`failover-priority`			`get set unset import`	-
`stripe-name`		`get set import`	`get`	`<generated>`
`name`	`get set import`	`get`	`get`	`<generated>`

Setting	Node	Stripe	Cluster	Default
hostname	get import	get	get	%h
port	get import	get	get	9410
public-hostname	get set unset import	get set unset	get set unset	-
public-port	get set unset import	get set unset	get set unset	-
group-port	get set unset import	get set unset	get set unset	9430
bind-address	get set unset import	get set unset	get set unset	0.0.0.0
group-bind-address	get set unset import	get set unset	get set unset	0.0.0.0
data-dirs	get set unset import	get set unset	get set unset	main:%H/terracotta/user-data/main
metadata-dir	get set unset import	get set unset	get set unset	%H/terracotta/metadata
log-dir	get set unset import	get set unset	get set unset	%H/terracotta/logs
backup-dir	get set unset import	get set unset	get set unset	-
tc-properties	get set unset import	get set unset	get set unset	-
logger-overrides	get set unset import	get set unset	get set unset	-

Setting	Node	Stripe	Cluster	Default
`security-dir`	get set unset import	get set unset	get set unset	-
`audit-log-dir`	get set unset import	get set unset	get set unset	-
`authc`			get set unset import	-
`ssl-tls`			get set unset import	FALSE
`whitelist`			get set unset import	FALSE
`offheap-resources`			get set unset import	main:512MB
`config-dir`				-
`node-uid`	get import	get	get	<generated>
`stripe-uid`		get import	get	<generated>
`cluster-uid`			get import	<generated>
`lock-context`			import	-

Configuration Rules for CONFIGURED (Activated) Nodes and Clusters

Once a cluster is activated and its underlying nodes are CONFIGURED, there is less flexibility as to which settings can be modified including whether the requested setting modification can be incorporated at runtime or if one or more Terracotta server restarts will be required.

The following table shows which Config Tool commands can be executed against each supported setting when the node is CONFIGURED (i.e. Activated).

Requirements (see Table below)

- CO - Cluster Online - All nodes must be online in order for the `set` or `unset` command to be accepted.

- CR - Cluster Restart - Every node in the cluster must be restarted after executing a `set` or `unset` command.
- NR - Node Restart - Only the node(s) directly impacted by the `set` and `unset` command must be restarted. The Config Tool will list these impacted nodes following successful execution of the command, asking for them to be restarted.
- AoN - All or None - All nodes in the cluster, or zero nodes in the cluster, can possess a configured value for the specified setting.

Note:
When executing `set` or `unset` against a setting which does not possess any requirements (**CO**, **CR**, **NR** or **AoN**) the setting *may* be automatically incorporated at runtime, without requiring a server restart. However, the system may determine that a restart of one or more nodes is required. Following successful execution of the command, the system will list the node(s), if any, that will require a restart.

Note:
Whenever the `set` or `unset` command is executed, the Active server for each stripe comprising the cluster must be online in order for the command to be accepted. That is, no setting can be modified unless every Active server in the cluster is online. This rule applies to all settings.

Setting	**Node**	**Stripe**	**Cluster**	**Requirements**
`license-file`			`set unset`	
`cluster-name`			`get set`	
`client-reconnect-window`			`get set unset`	
`client-lease-duration`			`get set unset`	
`failover-priority`			`get set`	CO, CR
`stripe-name`		`get`	`get`	
`name`	`get`	`get`	`get`	
`hostname`	`get`	`get`	`get`	
`port`	`get`	`get`	`get`	
`public-hostname`	`get set unset`	`get set unset`	`get set unset`	AoN
`public-port`	`get set unset`	`get set unset`	`get set unset`	AoN
`group-port`	`get`	`get`	`get`	
`bind-address`	`get`	`get`	`get`	
`group-bind-address`	`get`	`get`	`get`	
`data-dirs`	`get set`	`get set`	`get set`	AoN

Setting	Node	Stripe	Cluster	Requirements
`metadata-dir`	`get set`	`get set`	`get set`	
`log-dir`	`get set unset`	`get set unset`	`get set unset`	NR
`backup-dir`	`get set unset`	`get set unset`	`get set unset`	AoN
`tc-properties`	`get set unset`	`get set unset`	`get set unset`	NR
`logger-overrides`	`get set unset`	`get set unset`	`get set unset`	
`security-dir`	`get set unset`	`get set unset`	`get set unset`	NR, AoN
`audit-log-dir`	`get set unset`	`get set unset`	`get set unset`	NR, AoN
`authc`			`get set unset`	CO, CR
`ssl-tls`			`get set unset`	CO, CR
`whitelist`			`get set unset`	CO, CR
`offheap-resources`			`get set`	
`config-dir`				
`node-uid*`	`get`	`get`	`get`	
`stripe-uid*`		`get`	`get`	
`cluster-uid*`			`get`	
`lock-context*`			`set unset`	

*Internally used system property

Config Tool Commands

The following sections describe each supported Config Tool " command" on page 52 and provide examples on their usage.

In some cases, we also show examples of `start_tc_server.sh|bat` script usage. Refer to "Starting and Stopping the Terracotta Server" on page 25 for complete details on using this script.

Note:
Any command requiring a `hostname:port` parameter can be substituted with the `ipAddress:port` of the target node.

Cluster Activation and Topology Changes

Activate

The `activate` command is used to activate the nodes of a cluster when the nodes are UNCONFIGURED. The activation process will automatically perform the following steps:

1. Validate the configuration consistency
2. Validate the license
3. Write the validated cluster configuration inside the configuration directory of all nodes
4. Restart all activated nodes

Once a cluster is activated, it becomes usable for Terracotta clients.

Syntax:

```
activate [-connect-to <hostname[:port]> [-config-file <file>]] [-cluster-name <name>]
 [-license-file <file>] [-restart-wait-time <duration>] [-restart-delay <duration>]
[-restrict]
```

Option	Description
`-connect-to <hostname[:port]>`	Terracotta server instance to which the Config Tool will connect and execute the command.
`-cluster-name <name>`	The name given to the activated cluster.
`-config-file <file>`	Configuration file containing the node definitions, including their stripe topology, to be activated.
`-license-file <file>`	File path to the license file.
`-restart-wait-time <duration>`	Maximum time to wait for the nodes to restart. Default: 120 seconds.
`-restart-delay <duration>`	Delay before the server restarts itself. Default: 2 seconds.
`-restrict`	Restricts the activation process to only the -connect-to node.

Examples:

1. Activating an unconfigured cluster whose topology has already been defined (e.g. through a series of previously executed node/stripe/cluster `attach` commands):

   ```
   config-tool.sh activate -connect-to host1:9410 -cluster-name MyCluster -license-file
    /path/to/license.xml
   ```

2. Activating an unconfigured cluster whose topology has been defined inside a `cluster.cfg` configuration file. See "The Terracotta Configuration File" on page 45 for details.

   In this scenario, each of the nodes must be online and running in DIAGNOSTIC mode in order for the activation to proceed.

   Note that the `connect-to` option cannot be used as the node connection details used by the Config Tool are defined inside the configuration file.

This flag shouldn't be used as a general way to activate a cluster. To work, this flag requires that the targeted node is unconfigured and has a topology that is the same as the cluster it needs to join, and other nodes also need to know about this node.

This is an action that can help recover from a broken configuration state, or when an `attach` command has failed. Examples of scenario where this command can be useful:

- Repair a node when normal activation has failed
- Repair an `attach` command that did not complete (topology was updated in existing nodes, but new nodes failed to be activated)
- Re-activate a node after it has been reset
- attach new nodes to a cluster where some nodes have already been started with `-auto-activate`

```
config-tool.sh activate -license-file /path/to/license.xml -config-file cluster.cfg
```

3. Activating an unactivated node belonging to an already activated cluster.

   In this scenario, **node7** has entered an UNCONFIGURED state but is still part of the cluster and thus needs to be re-activated. Restricted activation will only run the activation process on the targeted node.

```
config-tool.sh activate -connect-to host7:9417 -config-file cluster.cfg -restrict
```

Attach

The `attach` command is used to add nodes to a cluster by either (1) adding nodes to a stripe or (2) adding whole stripes and their child nodes to the cluster.

Syntax:

```
attach (-to-cluster <hostname[:port]> -stripe <hostname[:port]> | -to-stripe
<hostname[:port]> -node <hostname[:port]>) [-restart-wait-time <duration>]
[-restart-delay <duration>]
```

Option	Description
`-stripe <hostname[:port]`	Stripe to add; identified by any node (as hostname:port) belonging to stripe.
`-to-cluster <hostname[:port]`	Destination cluster; identified by any node (as hostname:port) belonging to any other stripe within the cluster.
`-node <hostname[:port]>`	Node to add; identified as hostname:port.
`-to-stripe <hostname[:port]>`	Destination stripe; identified by any node (as hostname:port) belonging to to-stripe.
`-restart-wait-time <duration>`	Maximum time to wait for the nodes to restart. Default: 120 seconds.

Option	Description
`-restart-delay <duration>`	Delay before the server restarts itself. Default: 2 seconds.

Note:
In the event that errors are reported by the Config Tool when executing the `attach` command, note the error messages and refer to the "Troubleshooting Guide" on page 75 for guidance.

Examples:

1. Attaching two UNCONFIGURED nodes to form a single stripe:

   Start **node3** as UNCONFIGURED:

   ```
   start-tc-server.sh -failover-priority=availability -name=node3 -hostname=host3
   -port=9413 -config-dir=/path/to/node3/repository
   ```

   Start **node4** as UNCONFIGURED:

   ```
   start-tc-server.sh -failover-priority=availability -name=node4 -hostname=host4
   -port=9414 -config-dir=/path/to/node4/repository
   ```

   Attach **node4** to **node3** to form a stripe with two nodes:

   ```
   config-tool.sh attach -to-stripe host3:9413 -node host4:9414
   ```

2. Attaching a node to an existing stripe:

   Start **node5** as UNCONFIGURED:

   ```
   start-tc-server.sh -failover-priority=availability -name=node5 -hostname=host5
   -port=9415 -config-dir=/path/to/node5/repository
   ```

   Attach **node5** to the stripe containing nodes **node3** and **node4**:

   ```
   config-tool.sh attach -to-stripe host3:9413 -node host5:9415
   ```

3. Attaching a stripe to a cluster:

   In the previous step we created a 3-node stripe (**Stripe-B** in our "example cluster" on page 53). Assuming that **Stripe-A** has already been constructed, we can attach **Stripe-B** to the cluster by specifying any node belonging to **Stripe-B** (**node3** in this case) and any other node belonging to another stripe in the cluster (**node1** in **Stripe-A** in this case):

   ```
   config-tool.sh attach -to-cluster host1:9410 -stripe host3:9413
   ```

   Similarly, **Stripe-C** can be attached to the cluster. Here we pick **node7** from **Stripe-C** and **node3** from **Stripe-B**:

   ```
   config-tool.sh attach -to-cluster host3:9413 -stripe host7:9417
   ```

`Detach`

The `detach` command is used to remove nodes from a cluster by either (1) removing nodes from their parent stripes or (2) removing whole stripes and their child nodes from the cluster.

Syntax:

```
detach (-from-cluster <hostname[:port]> -stripe <hostname[:port]> | -from-stripe
<hostname[:port]> -node <hostname[:port]>) [-stop-wait-time <duration>] [-stop-delay
 <duration>]
```

Option	Description
`-stripe <hostname[:port]`	Stripe to remove; identified by any node (as `hostname:port`) belonging to `stripe`.
`-from-cluster <hostname[:port]`	Destination cluster; identified by any node (as `hostname:port`) belonging to any other stripe within the cluster.
`-node <hostname[:port]>`	Node to remove; identified as `hostname:port`.
`-from-stripe <hostname[:port]>`	Destination stripe; identified by any node (as `hostname:port`) belonging to `from-stripe`.
`-stop-wait-time <duration>`	Maximum time to wait for the nodes to restart. Default: 120 seconds.
`-stop-delay <duration>`	Delay before the server restarts itself. Default: 2 seconds.

Note:
In the event that errors are reported by the Config Tool when executing the `detach` command, note the error messages and refer to the "Troubleshooting Guide" on page 75 for guidance.

Examples:

1. Detach a node from a stripe. In our"example cluster" on page 53 we could detach **node4** from **Stripe-B** with:

   ```
   config-tool.sh detach -from-stripe host3:9413 -node host4:9414
   ```

2. Detach a stripe from a cluster. Specify any node (active or passive) within the cluster (`from-cluster`) that does not belong to the stripe we wish to detach, and specify any node (active or passive) belonging to the stripe we wish to detach. Here we detach **Stripe-C** from **Stripe-A:**

   ```
   config-tool.sh detach -from-cluster host2:9412 -stripe host7:9417
   ```

   Note:
   When detaching a stripe, all nodes in the detached stripe are stopped.

Performing configuration changes

Examples shown in this section follow the " Namespace Syntax" on page 54 where the `namespace` determines the scope of the `get`, `set` and `unset` commands. The formats are:

- `stripe.<stripeId>.node.<nodeId>` to read/write settings for a single, specific node.
- `stripe.<stripeId>` to read/write settings for all nodes within a stripe.

- `no namespace` to read/write settings for all nodes of the cluster.

Note:

Refer to "Configuration Rules for UNCONFIGURED (not Activated) Nodes and Clusters" on page 55 and " Configuration Rules for CONFIGURED (Activated) Nodes and Clusters" on page 57 to understand exactly which of `get`, `set` and `unset` are supported for each setting and at what cluster/stripe/node scope they are supported.

Note:

The `get`, `set`, and `unset` commands support specifying multiple `setting <setting>` parameters on the command line in order to read/write multiple settings simultaneously. Together, these settings constitute a *change-set* which is applied transactionally (i.e. applied fully or not applied at all).

Get

The `get` command is used to read the configuration data that defines the cluster, its stripes, and their nodes.

Syntax:

```
get -connect-to <hostname[:port]> [-runtime] [-output-format <cfg|properties>] -setting
 <[namespace:]setting> -setting <[namespace:]setting> ...
```

Option	Description
`-connect-to <hostname[:port]`	Terracotta server instance to which the Config Tool will connect and execute the command.
`-setting <[namespace:]setting>` `-setting <[namespace:]setting> …`	List of settings for which to retrieve the values.
`-runtime`	Returns the current, runtime values of the specified setting(s). When using this option, the `connect-to` node should be the node for which you wish to retrieve the runtime values.

Examples

1. Get the `offheap-resources` cluster-wide setting:

```
config-tool.sh get -connect-to host2:9412 -setting offheap-resources
```

2. Get each `offheap-resource` separately (there are two in our example cluster):

```
config-tool.sh get -connect-to host2:9412 -setting offheap-resources.main -setting
 offheap-resources.other
offheap-resources.main=512MB
offheap-resources.other=512MB
```

3. Get the `cluster-name`:

   ```
   config-tool.sh get -connect-to host2:9412 -setting cluster-name
   ```

4. Get the `hostname` for all nodes in **Stripe-B**:

   ```
   config-tool.sh get -connect-to host2:9412 -setting Stripe-B:hostname
   node:node3:hostname=host3
   node:node4:hostname=host4
   node:node5:hostname=host5
   ```

5. Get the `log-dir` location belonging to **node7** :

   ```
   config-tool.sh get -connect-to host2:9412 -setting node7:log-dir
   ```

6. Get the `runtime` value for a setting.

   In this example we assume the log directory for **node7** was previously updated to `dirs\node7\logs\updated`. But the node was not restarted, which is required for the change to take effect. In this case, since the node was not restarted, the current `runtime` log directory will continue to use the original folder location until the node is restarted. We can see this by querying for `log-dir` using the `runtime` option:

   ```
   config-tool.sh get -connect-to host7:9417 -setting node7:log-dir -runtime
   ```

Without specifying the `runtime` option, the updated value will be returned:

```
config-tool.sh get -connect-to host7:9417 -setting node7:log-dir
```

Note:

Whenever querying for a particular node's `runtime` setting value, the `connect-to` server should always be that particular node. For example, in the previous example, we queried for the node7 `runtime` value using the `stripe.3.node.2` namespace. Therefore, we needed to `connect-to` node7 as `host7:9417`. We could have connected-to any node in the cluster. But any node other than node7 would have returned the updated setting value, even if we had specified the `runtime` option.

Set

The `set` command is used to make configuration changes for the cluster, its stripes, and their nodes by writing values for the various supported settings.

Syntax:

```
set -connect-to <hostname[:port]> -setting <[namespace:]setting=value> -setting
<[namespace:]setting=value> ...
```

Option	Description
`-connect-to <hostname[:port]`	Terracotta server instance to which the Config Tool will connect and execute the command.

Option	Description
`-setting <[namespace:]setting =value>` `-setting <[namespace:]setting=value>` ...	List of settings and their values to set.
`-auto-restart`	Will automatically restart passive servers then active servers if a change requires some nodes to be restarted
`-restart-wait-time <duration>`	Maximum time to wait for the nodes to restart. Default: 120 seconds.
`-restart-delay <duration>`	Delay before the server restarts itself. Default: 2 seconds.

Examples

1. Change the name of the cluster:

   ```
   config-tool.sh set -connect-to host3:9413 -setting cluster-name=MyCluster
   ```

2. Establish a new offheap resource:

   ```
   config-tool.sh set -connect-to host3:9413 -setting offheap-resources.other=512MB
   ```

3. Simultaneously update the size of each of the cluster's offheap resources:

   ```
   config-tool.sh set -connect-to host4:9414 -setting offheap-resources.main=1GB
   -setting offheap-resources.other=1GB
   ```

4. Configure the cluster's failover priority (consistency) to include 1 voter. Refer to "Failover Tuning" on page 99.

   ```
   config-tool.sh set -connect-to host4:9414 -setting failover-priority=consistency:1
   ```

5. Attempt to change the name for **node4** (post activation):

   ```
   config-tool.sh set -connect-to host1:9410 -setting node4:name=new-node4-name
   Error: Invalid input: 'node4:name=new-node4-name'. Reason: Setting 'name' cannot
   be set when node is activated
   ```

   Node names are immutable after cluster activation. Therefore the `set` command is an invalid operation for the `name` setting (see " Configuration Rules for CONFIGURED (Activated) Nodes and Clusters" on page 57.

6. Update a cluster's license

   ```
   config-tool.sh set -connect-to host1:9410 -setting license-file=path/to/license.xml
   ```

   It is also possible to remove an installed license file. For example, a misplaced license file can be removed using:

   ```
   config-tool.sh unset -connect-to host1:9410 -setting license-file
   ```

Pay attention that if a license file is removed from an activated cluster with some connected clients, some features could then be refused and clients could malfunction because the cluster would become unlicensed.

7. Configure SSL/TLS security for the cluster. Note that in this example, every node in the cluster is assigned the same security root directory path on their local nodes:

   ```
   config-tool.sh set -connect-to host1:9410 -setting
   security-dir=path/to/server/security-dir
   -setting authc=certificate -setting ssl-tls=true
   ```

   For details on configuring security in a Terracotta cluster see " Security Core Concepts" on page 156 and "Cluster Security" on page 161.

8. After having configured security for our cluster in the previous example, we now set the backup directories for all seven nodes of our " example cluster" on page 53 to a unique folder location. In this example, we `connect-to` a node using a secure connection:

   ```
   config-tool.sh -security-dir path/to/client/security-dir set
   -connect-to host1:9410 -setting node1:backup-dir=node1/backup/folder
   -setting node2:backup-dir=node2/backup/folder
   -setting node3:backup-dir=node3/backup/folder
   -setting node4:backup-dir=node4/backup/folder
   -setting node5:backup-dir=node5/backup/folder
   -setting node6:backup-dir=node6/backup/folder
   -setting node7:backup-dir=node7/backup/folder
   ```

`Unset`

The `unset` command is used to remove (i.e. undo) previously configured setting for the cluster, its stripes, and their nodes.

Syntax:

```
unset -connect-to <hostname[:port]> -setting <[namespace:]setting> -setting
<[namespace:]setting> ... [-auto-restart] [-restart-wait-time <duration>]
[-restart-delay <duration>]
```

Option	**Description**
`-connect-to <hostname[:port]`	Terracotta server instance to which the Config Tool will connect and execute the command.
`-setting <[namespace:]setting>` `-setting <[namespace:]setting> …`	List of settings and their values to unset.
`-auto-restart`	Will automatically restart passive servers then active servers if a change requires some nodes to be restarted.
`-restart-wait-time <duration>`	Maximum time to wait for the nodes to restart. Default: 120 seconds.
`-restart-delay <duration>`	Delay before the server restarts itself. Default: 2 seconds.

Examples

1. Remove logger-overrides for a particular node:

   ```
   config-tool.sh unset -connect-to host1:9410 -setting node1:logger-overrides
   ```

2. Remove the backup directory configuration from each node in the cluster:

   ```
   config-tool.sh unset -connect-to host1:9410 -setting backup-dir
   ```

3. When configured as a secure cluster, the audit log directory is an optional, **all-or-none** setting. Here we remove the audit log directory configuration from all nodes in the cluster.

   ```
   config-tool.sh -security-dir path/to/client/security-dir unset -connect-to host1:9410
    -setting audit-log-dir
   ```

Export

The `export` command is used to export all information pertaining to the cluster, its stripes, and their nodes.

Syntax:

```
export -connect-to <hostname[:port]> [-output-file <config-file>] [-include-defaults]
 [-runtime]
```

Option	Description
`-connect-to <hostname[:port]`	Terracotta server instance to which the Config Tool will connect and execute the command.
`-output-file <config-file>`	The output configuration file. If this option is not specified, the configuration is displayed on the console.
`-include-defaults`	Include the settings having system default values as well. Default: false
`-runtime`	Export the runtime configuration instead of the on-disk configuration. Default: false

Examples

1. Export the on-disk configuration of our 3-stripe, 7-node "example cluster" on page 53 to the console (line spacings added for readability):

   ```
   config-tool.sh export -connect-to host1:9410 -include-defaults
   # Timestamp of configuration export: 2021-05-01T12:00:00.000Z
   # User-defined configurations
   stripe-names=Stripe-A,Stripe-B,Stripe-C
   stripe:Stripe-A:node-names=node1,node2
   stripe:Stripe-B:node-names=node3,node4,node5
   stripe:Stripe-C:node-names=node6,node7
   cluster-name=MyCluster
   failover-priority=consistency
   offheap-resources=main:1GB,other:1GB
   ```

```
node:node1:backup-dir=dirs\node1\backup
node:node1:bind-address=0.0.0.0
node:node1:data-dirs=main:dirs\node1\main
node:node1:group-bind-address=0.0.0.0
node:node1:group-port=8430
node:node1:hostname=host1
node:node1:log-dir=dirs\node1\logs
node:node1:metadata-dir=dirs\node1\metadata
node:node1:port=9410
node:node2:backup-dir=dirs\node2\backup
node:node2:bind-address=0.0.0.0
node:node2:data-dirs=main:dirs\node2\main
node:node2:group-bind-address=0.0.0.0
node:node2:group-port=8432
node:node2:hostname=host2
node:node2:log-dir=dirs\node2\logs
node:node2:metadata-dir=dirs\node2\metadata
node:node2:port=9412
node:node3:backup-dir=dirs\node3\backup
node:node3:bind-address=0.0.0.0
node:node3:data-dirs=main:dirs\node3\main
node:node3:group-bind-address=0.0.0.0
node:node3:group-port=8433
node:node3:hostname=host3
node:node3:log-dir=dirs\node3\logs
node:node3:metadata-dir=dirs\node3\metadata
node:node3:port=9413
node:node4:backup-dir=dirs\node4\backup
node:node4:bind-address=0.0.0.0
node:node4:data-dirs=main:dirs\node4\main
node:node4:group-bind-address=0.0.0.0
node:node4:group-port=8434
node:node4:hostname=host4
node:node4:log-dir=dirs\node4\logs
node:node4:metadata-dir=dirs\node4\metadata
node:node4:port=9414
node:node5:backup-dir=dirs\node5\backup
node:node5:bind-address=0.0.0.0
node:node5:data-dirs=main:dirs\node5\main
node:node5:group-bind-address=0.0.0.0
node:node5:group-port=8435
node:node5:hostname=host5
node:node5:log-dir=dirs\node5\logs
node:node5:metadata-dir=dirs\node5\metadata
node:node5:port=9415
node:node6:backup-dir=dirs\node6\backup
node:node6:bind-address=0.0.0.0
node:node6:data-dirs=main:dirs\node6\main
node:node6:group-bind-address=0.0.0.0
node:node6:group-port=8436
node:node6:hostname=host6
node:node6:log-dir=dirs\node6\logs
node:node6:metadata-dir=dirs\node6\metadata
node:node6:port=9416
node:node7:backup-dir=dirs\node7\backup
node:node7:bind-address=0.0.0.0
node:node7:data-dirs=main:dirs\node7\main
node:node7:group-bind-address=0.0.0.0
node:node7:group-port=8437
node:node7:hostname=host7
```

```
node:node7:log-dir=dirs\node7\logs
node:node7:metadata-dir=dirs\node7\metadata
node:node7:port=9417
# Default configurations
client-lease-duration=150s
client-reconnect-window=120s
ssl-tls=false
whitelist=false
# Hidden internal system configurations (only for informational, import and repair
 purposes): please do not alter, get, set, unset them.
cluster-uid=bBxeCPytSX6zzu6QE17YUw
node:node1:node-uid=MDIUxIFuS3e9oljgAnqa6w
node:node2:node-uid=ezlBtxKHRMuQ8mff_fWjXg
stripe:Stripe-A:stripe-uid=eQR_OuF9QeupUaf1UH40yg
node:node3:node-uid=kA43NxI1QVi8T3AwZeOPiw
node:node4:node-uid=dtPHkx0CRJOfTd5_uaKqcg
node:node5:node-uid=8y6eHs1EQqayuK2iD-zgRQ
stripe:Stripe-B:stripe-uid=M7zsScxITXu68cA4_6vkAg
node:node6:node-uid=W5oQW761RhSU2da1zK4ElA
node:node7:node-uid=j2YmJoyRTGCSXZt4i3MFsA
stripe:Stripe-C:stripe-uid=jEZZ6hXyR-mpNWz0Jly4JA
```

2. Export the on-disk configuration of a cluster, omitting default values and saving it to a file:

```
config-tool.sh export -connect-to host1:9410 -output-file
/path/to/output/configuration.cfg
```

Import

The `import` command is used to define a cluster topology including its stripes and nodes by importing those configuration settings via a configuration file. This import operation can only be executed on an UNCONFIGURED cluster (i.e. an unactivated cluster).

Syntax:

```
import -config-file <configuration-file> [-connect-to <hostname[:port]>]
```

Option	Description
`-config-file <configuration-file>`	The configuration properties file holding the cluster, stripe, and node configuration information. See "The Terracotta Configuration File" on page 45
`-connect-to <hostname[:port]>`	A single node to which the configuration file will be imported. If this setting is omitted, then all nodes represented in the configuration file will be updated.

Examples

1. Assume we have multiple unconfigured nodes started in diagnostic mode for which we want to form an activated cluster. First we `import` a configuration file (similar to the one we exported in our `export` example above) containing the detailed cluster, stripe, and node configuration information for all the nodes making up the cluster. We import that file to one of the running nodes (**node1** in this example):

```
config-tool.sh import -config-file configuration.cfg -connect-to host1:9410
```

Then we activate the cluster as we showed earlier in our "activate" on page 59 example (note that we can omit the `cluster-name` setting because our `configuration.properties` file happened to already contain an entry for that):

```
config-tool.sh activate -connect-to host1:9410 -license-file /path/to/license.xml
```

Note:

If any one of the nodes listed in the `configuration.cfg` file is not online, running as an unconfigured node (i.e. started in diagnostic mode) or is already activated, then the `activate` command will fail.

Diagnosing and Repairing Problems

`Diagnostic`

The `diagnostic` command displays detailed status information for each node in the cluster including: online, activation, health, restart and state. Last changed configuration details are also displayed.

Syntax

```
diagnostic -connect-to <hostname[:port]>
```

Option	Description
`-connect-to <hostname[:port]`	Terracotta server instance to which the Config Tool will connect and execute the command.

Example of detailed status/diagnostic information for the cluster:

```
config-tool.sh diagnostic -connect-to host1:9410
Diagnostic result:
[Cluster]
 - Nodes: 7 (node1@host1:9410, node2@host2:9412, node3@host3:9413, node4@host4:9414,
 node5@host5:9415, node6@host6:9416, node7@host7:9417)
 - Nodes online: 7 (host1:9410, host2:9412, host3:9413, host4:9414, host5:9415,
host6:9416, host7:9417)
 - Nodes online, configured and activated: 7 (host1:9410, host2:9412, host3:9413,
host4:9414, host5:9415, host6:9416, host7:9417)
 - Nodes online, configured and in repair: 0
 - Nodes online, new and being configured: 0
 - Nodes pending restart: 7 (host1:9410, host2:9412, host3:9413, host4:9414, host5:9415,
 host6:9416, host7:9417)
 - Configuration state: The cluster configuration is healthy and all nodes are online.
 No repair needed. New configuration changes are possible.
[node1@host1:9410]
 - Node state: PASSIVE
 - Node online, configured and activated: YES
 - Node online, configured and in repair: NO
 - Node online, new and being configured: NO
 - Node restart required: YES
 - Node configuration change in progress: NO
 - Node can accept new changes: YES
 - Node current configuration version: 3
 - Node highest configuration version: 3
 - Node last configuration change UUID: 7a37809c-e6c0-42f8-8c61-e2a47142b38f
 - Node last configuration state: COMMITTED
 - Node last configuration created at: 2021-02-18T14:57:54.063
 - Node last configuration created from: COMPANY-PC
 - Node last configuration created by jdoe
 - Node last configuration change details: set backup-dir=new-backup-dir
 - Node last mutation at: 2021-02-18T14:57:54.333
 - Node last mutation from: COMPANY-PC
 - Node last mutation by: jdoe
[node2@host2:9412]
 - Node state: ACTIVE
 - Node online, configured and activated: YES
 - Node online, configured and in repair: NO
 - Node online, new and being configured: NO
 - Node restart required: YES
 - Node configuration change in progress: NO
 - Node can accept new changes: YES
 - Node current configuration version: 3
 - Node highest configuration version: 3
 - Node last configuration change UUID: 7a37809c-e6c0-42f8-8c61-e2a47142b38f
 - Node last configuration state: COMMITTED
 - Node last configuration created at: 2021-02-18T14:57:54.063
 - Node last configuration created from: COMPANY-PC
 - Node last configuration created by: jdoe
 - Node last configuration change details: set backup-dir=new-backup-dir
 - Node last mutation at: 2021-02-18T14:57:54.333
 - Node last mutation from: COMPANY-PC
 - Node last mutation by: jdoe
[node3@host3:9413]
 - Node state: ACTIVE
 - Node online, configured and activated: YES
 - Node online, configured and in repair: NO
 - Node online, new and being configured: NO
 - Node restart required: YES
 - Node configuration change in progress: NO
 - Node can accept new changes: YES
 - Node current configuration version: 3
 - Node highest configuration version: 3
 - Node last configuration change UUID: 7a37809c-e6c0-42f8-8c61-e2a47142b38f
 - Node last configuration state: COMMITTED
 - Node last configuration created at: 2021-02-18T14:57:54.063
 - Node last configuration created from: COMPANY-PC
 - Node last configuration created by: jdoe
 - Node last configuration change details: set backup-dir=new-backup-dir
 - Node last mutation at: 2021-02-18T14:57:54.333
 - Node last mutation from: COMPANY-PC
 - Node last mutation by: jdoe
[node4@host4:9414]
 - Node state: PASSIVE
 - Node online, configured and activated: YES
 - Node online, configured and in repair: NO
 - Node online, new and being configured: NO
 - Node restart required: YES
 - Node configuration change in progress: NO
 - Node can accept new changes: YES
 - Node current configuration version: 3
 - Node highest configuration version: 3
 - Node last configuration change UUID: 7a37809c-e6c0-42f8-8c61-e2a47142b38f
 - Node last configuration state: COMMITTED
 - Node last configuration created at: 2021-02-18T14:57:54.063
 - Node last configuration created from: COMPANY-PC
 - Node last configuration created by: jdoe
 - Node last configuration change details: set backup-dir=new-backup-dir
 - Node last mutation at: 2021-02-18T14:57:54.333
 - Node last mutation from: COMPANY-PC
 - Node last mutation by: jdoe
[node5@host5:9415]
 - Node state: PASSIVE
 - Node online, configured and activated: YES
 - Node online, configured and in repair: NO
 - Node online, new and being configured: NO
 - Node restart required: YES
 - Node configuration change in progress: NO
 - Node can accept new changes: YES
 - Node current configuration version: 3
 - Node highest configuration version: 3
 - Node last configuration change UUID: 7a37809c-e6c0-42f8-8c61-e2a47142b38f
 - Node last configuration state: COMMITTED
 - Node last configuration created at: 2021-02-18T14:57:54.063
 - Node last configuration created from: COMPANY-PC
 - Node last configuration created by: jdoe
 - Node last configuration change details: set backup-dir=new-backup-dir
 - Node last mutation at: 2021-02-18T14:57:54.333
 - Node last mutation from: COMPANY-PC
 - Node last mutation by: jdoe
[node6@host6:9416]
 - Node state: PASSIVE
 - Node online, configured and activated: YES
 - Node online, configured and in repair: NO
 - Node online, new and being configured: NO
 - Node restart required: YES
 - Node configuration change in progress: NO
 - Node can accept new changes: YES
 - Node current configuration version: 3
 - Node highest configuration version: 3
 - Node last configuration change UUID: 7a37809c-e6c0-42f8-8c61-e2a47142b38f
 - Node last configuration state: COMMITTED
 - Node last configuration created at: 2021-02-18T14:57:54.063
 - Node last configuration created from: COMPANY-PC
 - Node last configuration created by: jdoe
 - Node last configuration change details: set backup-dir=new-backup-dir
 - Node last mutation at: 2021-02-18T14:57:54.333
 - Node last mutation from: COMPANY-PC
 - Node last mutation by: jdoe
[node7@host7:9417]
 - Node state: ACTIVE
 - Node online, configured and activated: YES
 - Node online, configured and in repair: NO
 - Node online, new and being configured: NO
 - Node restart required: YES
 - Node configuration change in progress: NO
 - Node can accept new changes: YES
 - Node current configuration version: 3
 - Node highest configuration version: 3
 - Node last configuration change UUID: 7a37809c-e6c0-42f8-8c61-e2a47142b38f
 - Node last configuration state: COMMITTED
 - Node last configuration created at: 2021-02-18T14:57:54.063
 - Node last configuration created from: COMPANY-PC
 - Node last configuration created by: jdoe
 - Node last configuration change details: set backup-dir=new-backup-dir
 - Node last mutation at: 2021-02-18T14:57:54.333
 - Node last mutation from: COMPANY-PC
 - Node last mutation by: jdoe
```

In the above example, the `diagnostic` output contains the following `Configuration state` message near the top of the listing:

```
Configuration state: The cluster configuration is healthy and all nodes are online.
No repair needed. New configuration changes are possible.
```

This specific message represents a perfectly healthy cluster. But if the cluster is not healthy, then the reason for the unhealthy state will be captured in the 'Configuration state' message block. Refer to the " Troubleshooting Guide" on page 75 for guidance on how to interpret these messages and for guidance on how to effect the necessary repairs to the cluster.

Repair

The `repair` command is used to repair the cluster health by fixing cluster configuration inconsistency issues on one or more nodes targeting single node repairs. Repairs only work on activated nodes - not nodes that are running in diagnostic mode. Running this command on a healthy cluster has no effect.

Important:
Please contact Software AG support before running this command.

Syntax:

```
repair -connect-to <hostname[:port]>
```

Option	Description
`-connect-to <hostname[:port]`	Terracotta server instance to which the Config Tool will connect and execute the repair.

Important:

In the event that problems are reported by the Config Tool when executing the `repair` command, note the error message and refer to the " Troubleshooting Guide" on page 75 for guidance.

Example of repairing a node:

```
config-tool.sh repair -connect-to host1:9410
```

Log

Retrieve details about all changes made to a node in the cluster.

Syntax:

```
log -connect-to <hostname[:port]>
```

Option	Description
`-connect-to <hostname[:port]`	Terracotta server instance to which the Config Tool will connect and execute the command to retrieve the node's logged changes.

Example of a query to a node (**node7**) for all configuration changes:

```
config-tool.sh log -connect-to host7:9417
1 2021-02-18T14:21:46.030 c7ac4b3c-bb5e-4481-a259-bace0ce4abbb
c3745cb968cd289ddabab70c9634e749823289cd COMMITTED | jdoe@COMPANY-PC - Activating
cluster: MyCluster
2 2021-02-18T14:57:11.298 64efd21b-5806-4d38-b3d2-a68f9e045378
1f050acacbdb1732e76bce60a9494bdde5dd80a5 COMMITTED | jdoe@COMPANY-PC - unset backup-dir
3 2021-02-18T14:57:54.063 7a37809c-e6c0-42f8-8c61-e2a47142b38f
e3661632e53ae15245ad8632c0b0dacd9faf1449 COMMITTED | jdoe@COMPANY-PC - set
backup-dir=new-backup-dir
```

Config Tool Troubleshooting Guide

Overview

The Config Tool provides detailed validation and error reporting when executing any of its commands. Most of the reported messages also include guidance on how to address/correct the underlying issue.

Some of the more import occurrences of failed validation scenarios and how to address them are described here in" Validation and Warning Messages" on page 76.

Occurrences of unexpected errors and how to identify and correct them are described in" Diagnosing Unexpected Errors" on page 81.

Before Performing Any Troubleshooting

Before performing any troubleshooting, please read and understand this section as it contains important background information.

When a cluster is activated, any Config Tool mutative operation such as "attach" on page 61, "detach" on page 62, "set" on page 65, "unset" on page 67, or "activate" on page 59 will proceed through the following steps:

1. Validation / Sanity checks: validation of the CLI input and sanity checks against the cluster.
2. Execution of the change process in three phases:
    a. **DISCOVERY**: the cluster configuration is verified to ensure that all nodes are ready to accept a new change.
        a. **Success**: the PREPARE phase is started.

    b. **Error**: Nothing has been changed thus far; we can simply retry or consult this guide for additional help.

  b. **PREPARE**: the change is sent to the targeted nodes for validation purposes and is written into a file which keeps track of the changes of a node. This phase does not update the runtime configuration, or the configuration used at startup. This is a validation step that indicates to the nodes what will be the next configuration to use if the validation passes on all the nodes.

    a. **Success**: the COMMIT phase is started.

    b. **Failure**: the ROLLBACK phase is started and the change that has been prepared will be marked as rolled back.

  c. **COMMIT**: if all the nodes have validated and accepted the change, the Config Tool asks all nodes to commit the change, which will change the runtime configuration if the change supports being applied at runtime. The cluster configuration used to start the nodes will be changed accordingly.

    a. **Success**: the change process is completed.

    b. **Failure**: Failure during a commit should never occur because the change has been validated and accepted upfront. These failures are caused by either a programmatic error or an environment change between the time the change has been validated and the time the commit has commenced. Hopefully the repair command can be used to replay this commit phase.

  d. **ROLLBACK**

    a. **Success**: the change process is cancelled.

    b. **Failure**: Failure during a rollback should never occur. Hopefully the repair command can be used to replay this rollback phase.

3. **Restart of the nodes**: some commands require some or all nodes to be restarted - either automatically, or performed manually by the user after having been warning to do so by the Config Tool. This is especially true for changes requiring nodes to be restarted because the changes cannot be applied at runtime.

Validation and Warning Messages

Attaching a Node to a Stripe

Command	“attach” on page 61 a node to a stripe
Symptom	The following message is returned: `Source node: <node_name> cannot be attached since it is part of an existing cluster with name: <cluster_name>`

Diagnosis	The source node is active and already belongs to an existing cluster which is different than the one to which it is being attached.
Action	1. Detach the source node from its existing source stripe. 2. Re-run the original `attach` command which generated this error.

Command	"attach" on page 61 a node to a stripe
Symptom	The following message is returned: `Source node: <node_name> is part of a stripe containing more than 1 nodes. It must be detached first before being attached to a new stripe. Please refer to the` "Troubleshooting Guide" on page 75 `for more help.`
Diagnosis	The source node already belongs to a multi-node cluster which is different than the one to which it is being attached.
Action	**Option A:** 1. Detach the node, which is to be attached to the destination stripe, from its existing source stripe. 2. Re-run the original `attach` command which generated this error. **Option B:** 1. Re-run the original `attach` command which generated this error but include the `-force` option. For example: `config-tool.sh attach -to-stripe <destination_stripe:port> -node <source_node:port> -force`

Attaching a Stripe to a Cluster

Command	"attach" on page 61 a stripe to a cluster
Symptom	The following message is returned: `Source stripe from node: <node_name> is part of a cluster containing more than 1 stripes. It must be detached first before being attached to a new cluster. Please refer to the Troubleshooting Guide for more help.`
Diagnosis	The source stripe already belongs to a multi-node cluster which is different than the one to which it is being attached.
Action	**Option A:** 1. Detach the stripe that is to be attached to the destination cluster from its existing source cluster.

	2. Re-run the original `attach` command which generated this error. **Option B:** 1. Re-run the original `attach` command which generated this error but include the `-force` option. For example: `config-tool.sh attach -to-cluster <destination_cluster:port> -stripe <source_stripe:port> -force`

Attaching Nodes or Stripes

Command	"attach" on page 61 a node to a stripe or a stripe to a cluster
Symptom	The following message is returned: `Impossible to do any topology change. Node: <node_endpoint> is waiting to be restarted to apply some pending changes. Please refer to the Troubleshooting Guide for more help.`
Diagnosis	One or more nodes belonging to the destination cluster have pending changes that require a restart. Ideally, topology changes should only be performed on clusters where the nodes have no pending updates.
Action	**Option A:** 1. Restart the node identified by `<node_endpoint>`. 2. Re-run the original `attach` command which generated this error. **Option B:** 1. Re-run the original `attach` command which generated this error but include the `-force` option. For example: `config-tool.sh attach -to-cluster <destination_cluster:port> -stripe <source_stripe:port> -force`

Command	"attach" on page 61 a node to a stripe or a stripe to a cluster
Symptom	The following message is returned: `An error occurred during the attach transaction. The node/stripe information may still be added to the destination cluster: you will need to run the diagnostic / export command to check the state of the transaction. The node/stripe to attach won't be activated and restarted, and their topology will be rolled back to their initial value.`
Diagnosis	The transaction applying the new topology has failed (the reason is detailed in the logs). It can be caused by an environmental problem (such as network issue, node

	shutdown, etc) or a concurrent transaction. If the failure occurred during the commit phase (partial commit), some nodes may need to be repaired.
Action	An 'auto-rollback' will be attempted by the system. Examine output to determine if the auto-rollback was successful. If it was not, then run the “diagnostic” on page 71 command.

Detaching Nodes or Stripes

Command	“detach” on page 62 a node from a stripe, or a stripe from a cluster.
Symptom	The following message is returned: `Impossible to do any topology change. Node: <node_name> is waiting to be restarted to apply some pending changes. Please refer to the Troubleshooting Guide for more help.`
Diagnosis	One or more nodes belonging to the destination cluster have pending changes that require a restart. Ideally, topology changes should only be performed on clusters where the nodes have no pending updates.
Action	**Option A:** 1. Restart the node identified by `<node_name>`. 2. Re-run the original `detach` command which generated this error. **Option B:** 1. Re-run the original `detach` command which generated this error but include the `-force` option. For example: `config-tool.sh detach -from-cluster <destination_cluster:port> -stripe <source_stripe:port> -force`

Command	“detach” on page 62 a node from a stripe.
Symptom	The following message is returned: `Nodes to be detached: <node_names> are online. Nodes must be safely shutdown first. Please refer to the Troubleshooting Guide for more help.`
Diagnosis	Ideally, nodes should only be detached when they are not running. Note that when detaching a stripe, the system will automatically stop all detached nodes. But for node detachments, this must be performed manually.
Action	**Option A:** 1. Manually stop the node identified by `<node_name>`. 2. Re-run the original `detach` command which generated this error.

	Option B: 1. Re-run the original `detach` command which generated this error but include the `-force` option. For example: `config-tool.sh detach -from-stripe <destination_stripe:port> -node <source_node:port> -force`

Split Brain Warning

Command	Commands that alter a stripe's total node count including "attach" on page 61, "detach" on page 62, and "import" on page 70.
Symptom	The following message is returned: `IMPORTANT: The sum (<x>) of voter count (<y>) and number of nodes (<z>) in stripe <stripe_name> is an even number. An even-numbered configuration is more likely to experience split-brain situations.`
Diagnosis	Even-numbered counts of voters plus nodes for a given stripe can increase the chances of experiencing split-brain situations.
Action	Consider making the total count for the stripe an odd number by adding a voter.

Errors or unexpected issues

"Some nodes may have failed to restart within…"

Command	Any mutative command including "attach" on page 61, "detach" on page 62, "activate" on page 59, "set" on page 65, or "unset" on page 67
Symptom	The following message is returned: `Some nodes may have failed to restart within <wait_time> seconds. This should be confirmed by examining the state of the nodes listed below. Note: if the cluster did not have security configured before activation but has security configured post-activation, or vice-versa, then the nodes may have in fact successfully restarted. This should be confirmed. Nodes: <node_name_list>`
Diagnosis	Some mutative commands restart the nodes and then wait for the nodes to come back online. This error message is displayed when the Config Tool was not able to see the node be back online within a delay given by the Config Tool parameter `-restart-wait-time`. Make sure the value is not too low.
Action	Execute the following steps:

	1. Execute the “diagnostic” on page 71 command for all the nodes that have failed to restart. 2. Examine the `Node state` value (refer to “node states” on page 17 for more information about the different node states): a. If one of `ACTIVE`, `ACTIVE_RECONNECTING`, `PASSIVE`: the node has restarted correctly. The `-restart-wait-time` value used with the Config Tool was not high enough. b. If one of `ACTIVE_SUSPENDED`, `PASSIVE_SUSPENDED`: the node startup is blocked because the vote count if not correct to pass the desire level of consistency. c. If one of `STARTING`, `SYNCHRONIZING`: the node is still starting… Just wait. d. If one of `DIAGNOSTIC` or `UNREACHABLE`: the node was unable to start, or has been started in diagnostic mode. Please look at the logs for any error and seek support if necessary.

`"Please run the 'diagnostic' command to diagnose the configuration state…"`

Command	Any mutative command including “attach” on page 61, “detach” on page 62, “activate” on page 59, “set” on page 65, “unset” on page 67, or “repair” on page 74
Symptom	The following message is returned: `Please run the 'diagnostic' command to diagnose the configuration state and try to run the 'repair' command. Please refer to the` “Troubleshooting Guide” on page 75 `for more help.`
Diagnosis	An inconsistency has been found in the cluster configuration and the operation cannot continue without a manual intervention or repair.
Action	Execute the following steps: 1. Execute the `diagnostic` command on the cluster. 2. Read the `'Configuration state'` message block near the top of the output. 3. Find the message in “ Diagnosing Unexpected Errors” on page 81 to understand the underlying problem and how to address it.

Diagnosing Unexpected Errors

Even if command validations pass, commands can still fail as a result of unexpected errors. The “diagnostic” on page 71 command is used to help identify the underlying error condition of the cluster.

The `diagnostic` command output reveals detailed status information for each node in the cluster including its: online, activation, health, restart, and state statuses. Last changed configuration details are also displayed.

The command also reveals details about certain unexpected failure conditions. This information is included in the `Configuration state` message block that appears near the top of the diagnostic output.

The following 'Configuration state' messages require attention and action.

Symptom	The `Configuration state` of the “diagnostic” on page 71 command output contains: `Failed to analyze cluster configuration.`
Diagnosis	The discovery process has failed. Possibly because another client is currently doing a mutative operation. This situation requires to retry the command.
Action	Run the command again

Symptom	The `Configuration state` message block of the `diagnostic` command output contains this message: `Cluster configuration is inconsistent`: Change `<change_uuid>` is committed on `<committed_nodes_list>` and rolled back on `<rolled_back_nodes_list>`.
Diagnosis	Certain changes were found that were committed on some servers and rolled back on other servers. This situation requires a manual intervention, possibly by resetting the node and then re-syncing it after a restricted activation.
Action	The repair of such a broken configuration state requires rewriting the configuration of certain nodes which will make them temporarily unavailable. To repair such issues, the nodes requiring a reset (nodes that have rolled back) and nodes requiring a reconfiguration (nodes that have committed the change) must be identified. There is no right or wrong answer as it depends on the specific case at hand and the user's intimate knowledge about what command(s) were issued. If the nodes that were committed have started satisfying requests in relation to the addition of a setting (e.g. offheap addition), then such changes need to be forced on the rolled-back node and it must be ensured that these nodes can accept such changes (e.g. enough offheap exists). At the opposite end, if it is known that a committed change has not been used then it can be safely removed. In this case you can consider maintaining the rolled-back nodes and resetting the committed ones. See “Repairing a Broken Configuration” on page 86

Symptom	The `Configuration state` of the `diagnostic` command output contains:

	`Cluster configuration is partitioned and cannot be automatically repaired. Some nodes have a different configuration that others.`
Diagnosis	Some nodes ending with a different change UUID leading to different configuration results have been found. Some nodes are running with one configuration, while other nodes are running with a different one. This situation requires a manual intervention, eventually by resetting the node and re-syncing it after a restricted activation.
Action	This requires a manual intervention analogous to the previously discussed 'Action' - i.e. resetting the configuration of certain nodes. See "Repairing a Broken Configuration" on page 86.

Symptom	The `Configuration state` message block of the `diagnostic` command output contains this message: `A new cluster configuration has been prepared on all nodes but not yet committed. No further configuration change can be done until the 'repair' command is run to finalize the configuration change.`
Diagnosis	All nodes are online and all online nodes have prepared a new change. This situation requires a commit to be replayed, or a rollback to be forced.
Action	Execute this command: `config-tool.sh repair -connect-to <host:port>`

Symptom	The `Configuration state` of the `diagnostic` command output contains: `A new cluster configuration has been prepared but not yet committed or rolled back on online nodes. Some nodes are unreachable, so we do not know if the last configuration change has been committed or rolled back on them. No further configuration change can be done until the offline nodes are restarted and the 'repair' command is run again to finalize the configuration change. Please refer to the` "Troubleshooting Guide" on page 75 `if needed.`
Diagnosis	Some nodes are online (not all) and all online nodes have prepared a new change. Because some nodes are down, we do not know if some offline nodes have some more changes in their append.log. This situation requires a commit or a rollback to be forced (only the user knows).
Action	Because some of the nodes are down, the Config Tool is not able to determine if the change process should be continued and committed, or if it should be rolled back. Only the user knows which action is required. The user must therefore provide the necessary hint to the Config Tool to either force a commit or force a rollback. `1) config-tool.sh repair -connect-to <host:port> -force commit`

	`2) config-tool.sh repair -connect-to <host:port> -force rollback`

Symptom	The `Configuration state` of the `diagnostic` command output contains:
	
	`A new cluster configuration has been partially prepared (some nodes didn't get the new change). No further configuration change can be done until the 'repair' command is run to rollback the prepared nodes.`
	
	`A new cluster configuration has been partially rolled back (some nodes didn't rollback). No further configuration change can be done until the 'repair' command is run to rollback all nodes.`

Diagnosis	A specific change has been prepared on some nodes, while other nodes, which didn't receive that specific change, are ending with a different change. This can happen if a transaction has ended during its prepare phase when the client asks the nodes to prepare themselves. This situation requires a rollback to be replayed.
Action	Execute this command: `config-tool.sh repair -connect-to <host:port>`

Symptom	The `Configuration state` of the `diagnostic` command output contains: `A new cluster configuration has been partially committed (some nodes didn't commit). No further configuration change can be done until the 'repair' command is run to commit all nodes.`
Diagnosis	A change has been prepared, then committed, but the commit process didn't complete on all online nodes. This situation requires a commit to be replayed.
Action	Execute this command: `config-tool.sh repair -connect-to <host:port>`

Symptom	The `Configuration state` of the `diagnostic` command output contains: `Unable to determine the global configuration state. There might be some configuration inconsistencies. Please look at each node details. A manual intervention might be needed to reset some nodes.`
Diagnosis	Unable to determine the configuration state of the cluster.
Action	The user might need to reset the configuration of some nodes. See “Repairing a Broken Configuration” on page 86. But to be able to determine which nodes to reset and how, some additional support is required. The user has to send all the server logs and configuration directories to the support team.

Errors from the repair command

The `repair` command is used to repair the cluster's health by fixing cluster configuration inconsistency issues on one or more nodes. Repairs only work on activated nodes - not on nodes that are running in diagnostic mode. The following errors might be observed when executing the `repair` command:

Symptom	Any of the following messages are observed when executing the `repair` command: ■ `Failed to analyze cluster configuration.` ■ `Cluster configuration is inconsistent: Change <change_uuid> is committed on <committed_nodes_list> and rolled back on <rolled_back_nodes_list>.` ■ `Cluster configuration is partitioned and cannot be automatically repaired. Some nodes have a different configuration that others.` ■ `Unable to determine the global configuration state. There might be some configuration inconsistencies. Please look at each node details. A manual intervention might be needed to reset some nodes`
Diagnosis	Refer to the same message in the "Diagnostic Command Troubleshooting" on page 75 section.
Action	Refer to the same message in the "Diagnostic Command Troubleshooting" on page 75 section.

Symptom	One of following messages is observed when executing the `repair` command: ■ `The configuration is partially prepared. A rollback is needed.` ■ `The configuration is partially rolled back. A rollback is needed.`
Diagnosis	The repair tool has detected that a rollback is necessary, but the user specified the wrong action.

Action	Execute one of these commands:
	`config-tool.sh repair -connect-to <host:port>`
	`config-tool.sh repair -connect-to <host:port> -force rollback`

Symptom	The following message is observed when executing the repair command: `The configuration is partially committed. A commit is needed.`
Diagnosis	The repair tool has detected that a commit is necessary, but the user specified the wrong action.
Action	Execute one of these commands:

	`config-tool.sh repair -connect-to <host:port>`
	`config-tool.sh repair -connect-to <host:port> -force commit`

Symptom	The following message is observed when executing the repair command: `Some nodes are offline. Unable to determine what kind of repair to run. Please refer to the Troubleshooting Guide.`
Diagnosis	The repair is unable to determine whether it needs to complete an incomplete change by committing or it needs to rollback because some nodes are down. This is up to the user to hint the repair command about what to do.

Action	Execute one of these commands:
	`config-tool.sh repair -connect-to <host:port>`
	`config-tool.sh repair -connect-to <host:port> -force commit`

Manual Repair

Unlocking a Locked Configuration

If a dynamic scale operation fails, it is possible for it to leave the cluster configuration locked, which will prevent any further mutation actions from being performed.

A locked cluster configuration can be unlocked with the following command:

```
config-tool.sh repair -connect-to <host:port> -force unlock
```

Repairing a Broken Configuration

If some nodes possess a broken configuration, which makes the cluster configuration inconsistent, these nodes can be repaired. This procedure will "force" the repaired node to acquire the same configuration as the nodes that are considered to be sane (i.e. correct).

Note:

It is up to the user to decide which nodes are considered to be correct and which nodes should be repaired.

When forcing a configuration onto certain nodes, the cluster configuration will become consistent, but other issues can occur if it is not verified that the environment can support the new configuration.

Steps:

1. Decide which nodes are sane and which nodes require repairing. In our example, `node1` is sane and `node2` will be repaired. We will attempt to force the configuration from `node1` to be installed in `node2`.

2. We backup the cluster configuration:

   ```
   config-tool.sh export -connect-to node1 -output-file <config-file>
   ```

3. If `node2` is down, restart it with the exact same command-line that was previously used but append `-repair-mode` at the end of the file in order for it to enter repair mode.

4. `node2` starts in `diagnostic` mode. Reset its configuration and stop it with:

   ```
   config-tool.sh repair -force reset -connect-to node2
   ```

5. Optional step: if you need to cleanup the data of this node, this is a good time to do that. To do so, backup and remove all the content of the `metadata-dir` and all the content of every `data-dir`. Backing up and removing the data will allow your node to start completely like new. You can keep your data ONLY in cases where you know the node can restart safely with it.

6. Start `node2` again, but this time start it normally, without `-repair-mode`. `node2` should start in `diagnostic` mode once again, waiting for a configuration to be pushed.

7. Run a restricted activation on `node2`. The node should activate, restart, sync with current active and become passive.

   ```
   config-tool.sh activate -restrict -connect-to node2 -config-file <config-file>
   ```

13 Parameter Substitution

Parameter substitution provides a way to substitute variables with pre-defined system properties in the Terracotta Server configuration file. Thus, a significant number of fields can be intelligently populated based on machine specific properties. Parameter substitution is commonly done for hostnames, IP addresses and directory paths.

The following predefined substitutions are available for use:

Parameter	Description
%h	the fully-qualified host name of the machine
%i	the IP address of the machine corresponding to localhost
%D	the time stamp corresponding to the current date-time in `yyyyMMddHHmmssSSS` format
%H	the user's home directory corresponding to the `user.home` Java system property
%n	the username corresponding to the `user.name` Java system property
%o	the operating system name corresponding to the `os.name` Java system property
%a	the processor architecture corresponding to the `os.arch` Java system property
%v	the operating system version corresponding to the `os.version` Java system property
%t	the temporary directory corresponding to the `java.io.tmpdir` Java system property
%d	unique temporary directory
%(system property)	a standard or custom Java system property. If a custom Java property needs to be used, it should be first set in the JVM by setting it in `JAVA_OPTS` in the `-Djava.custom.property=value` format.

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.

14 Configuring the Terracotta Server

Overview

For your application end-points to be useful, they must be able to utilize storage resources configured in your Terracotta Servers. The services offered make use of your server's underlying JVM and OS resources, including direct-memory (offheap) and disk persistence.

Offheap Resources

The use of JVM Direct-Memory (offheap) is a central part of the operation of a Terracotta Server. In effect, you **must** allocate and make available to your server enough offheap memory for the proper operation of your application.

In your config file you define one or more named *offheap resources* of a fixed size. These named resources are then referred to in the configuration of your application end-points to allow for their usage.

Refer to the section *Clustered Caches* in the *Ehcache API Developer Guide* for more details about the use of offheap resources.

Offheap Resource Configuration

Offheap resources can be configured with the `offheap-resources` property. By default, an offheap resource with name `main` and size `512MB` is defined on the server.

The following snippet shows the configuration of two offheap resources - `primary-server-resource` with size 384MB, and `secondary-server-resource` with size 256MB.

```
offheap-resources=primary-server-resource:384MB,secondary-server-resource:256MB
```

Data Directories

A data directory is a location on disk, identified by a name, and mapped to a disk location, where a Terracotta Server's data resides.

Data directories are commonly configured by server administrators and specified in the Terracotta Server configuration. Data directory names can be used by products that need durable storage for persistence and fast restart from crashes. For example, restartable cache managers need to be supplied with a data directory name to persist the restartable `CacheManager` specific data.

For information on restartable servers, see the section "Server Persistence" on page 92 below. See also the sections *Fast Restartability* and *Creating a Restartable Cache Manager* in the *Ehcache API Developer Guide*.

Data Directories Configuration

Data directories can be configured with the `data-dirs` property. By default, a data directory with name `main` is defined which maps to disk location `%(user.home)/terracotta/user-data/main`.

The following snippet shows the configuration of two data directories - `someData` with disk location `/mnt1/data`, and `otherData` with location `%(logs.path)/data`.

```
data-dirs=someData:/mnt1/data,otherData:%(logs.path)/data
```

`%(logs.path)` gets substituted with the logs path system property. Visit the section "Parameter Substitution" on page 89 for the complete list of substitutable parameters.

General Notes on Configuring Data Directories

- A data directory specified on a server must be specified on all the servers in the cluster.
- Each data directory must be given a unique mount point (or disk location).
- The data directories are created if they do not exist already.
- Changing the disk location of the data directory between server restarts, without copying the data, is equivalent to erasing that data. It will cause unpredictable runtime errors that depend on the exact data lost.

Server Persistence

The Terracotta server saves its internal state on a disk which enables server restarts without loss of data.

Care must be taken to avoid losing data when restarting the stripe. Refer to the section "Restarting a Stripe" on page 149 for more details. Passive restartable servers automatically back up their data at restart for safety reasons. Refer to the topic *Passive servers* in the section "Active and Passive Servers" on page 13 for more details.

Server Persistence Configuration

Server persistence is configured with the `metadata-dir` property. By default, it is set to `%(user.home)/terracotta/metadata`.

The following snippet shows the configuration of `metadata-dir` to disk location `/path/to/metadata-dir`:

```
metadata-dir=/path/to/metadata-dir
```

Relation to Fast Restartability

The Ehcache *Fast Restartability* feature depends on, and makes use of, server persistence.

Refer to the section *Fast Restartability* in the *Ehcache API Developer Guide* for more information.

15 System Recommendations for Hybrid Caching

Hybrid Caching supports writing to one single mount, so all of the 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. The software was designed for usage on local drives (SSD/Flash in particular) - SAN/NAS storage is not recommended. If you utilize SAN/NAS storage you will experience notably reduced and inconsistent performance - any support requests related to performance or stability on such deployments will require the user to reproduce the issue with local disks.

Note:
System utilization is higher when using Hybrid Caching, and it is not recommended to run multiple servers on the same machine. Doing so could result in 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.

Hybrid Caching is described in detail in the Developer Guide.

16 System Recommendations for Fast Restart (FRS)

Fast Restart (FRS) supports writing to one single mount, which can be a single device or a RAID.

The mount should be used exclusively for the Terracotta server process. The software was designed for usage on local drives (SSD/Flash in particular) - SAN/NAS storage is not recommended. If you utilize SAN/NAS storage you will experience notably reduced and inconsistent performance - any support requests related to performance or stability on such deployments will require the user to reproduce the issue with local disks.

Fast Restartability is described in detail in the Developer Guide.

17 Failover Tuning

Overview

A Terracotta Server Array (TSA), being a distributed system, is subject to the constraints of the *CAP Theorem*. The CAP Theorem states that it is impossible for a distributed system to simultaneously provide guarantees for consistency, availability, and partition tolerance. A TSA always seeks to be tolerant of network partitions so a choice must be made between data consistency and service availability. This choice is declared with the *failover-priority* setting which instructs the cluster to favor either *consistency* or *availability*. When a network or hardware failure occurs resulting in disconnected servers, the cluster's behavior is governed by this setting.

If a cluster is defined to favor consistency, then failures resulting in disconnected servers will result in all client requests being halted. This is due to an inability to guarantee consistent reads and writes when the cluster is partitioned.

If a cluster is defined to favor availability, then failures resulting in disconnected servers will still permit the cluster to respond to client requests. However, client responses are not guaranteed to be consistent because there can potentially be multiple active servers working on the same data set.

In the absence of any error-induced cluster partitioning, the cluster will provide both consistency and availability.

Elections and Split-Brains

When a failure that results in disconnected servers within a cluster configured to favor availability over consistency occurs, each set of partitioned servers will perform an *election*. Servers that can communicate with each other will elect an active server among themselves and continue operations. If the failure results in a single server being isolated, that server will elect itself as active. Opting to favor availability over consistency comes with the possibility of experiencing a so-called *split-brain* situation.

For a TSA, a split-brain occurs when multiple servers within a stripe are simultaneously acting as active servers. For example, if an active server becomes partitioned from its peers in the stripe, the active server will remain active and the passive servers on the other side of the partition, unable to reach the active server, will elect a new active server from their own isolated group. Any further operations performed on the data are likely to result in inconsistencies because there can be multiple active servers working on the same data set.

When configured for consistency, a stripe requires a majority of servers connected with each other to elect an active server. Thus, even if the stripe gets partitioned into two sets of servers due to some network failure, the set with the majority of servers will elect an active server among them and proceed. In the absence of a majority, an active server will not be elected and hence the clients will be prevented from performing any operations, thereby preserving data consistency by sacrificing availability.

Server configuration

When configuring the cluster, you must choose between availability or consistency as the failover priority of the cluster. To prevent split-brain scenarios and thereby preserve data consistency, failover priority must be set to consistency. However, if availability is preferred, `failover-priority` can be set to availability at the risk of running into split-brain scenarios. The following snippet shows how to configure a stripe for consistency:

Configured via config file:

```
failover-priority=consistency
```

Configured via command line during server startup (use `-failover-priority` option):

```
start-tc-server.sh -failover-priority consistency
```

Similarly, the cluster can be tuned for *availability* as follows:

```
failover-priority=availability
```

Note:
`failover-priority` is a mandatory parameter that must be provided during server startup if the cluster configuration contains more than one node.

Choosing Consistency versus Availability

For a TSA supporting only Ehcache users, choosing availability over consistency can be an effective and proper choice. If a network partition occurs leaving two servers unable to communicate with each other (to coordinate data) but each able to serve clients, choosing availability allows continuing operations using both servers running independently. Data residing on the separated servers will no longer be coordinated (consistency is abandoned) and will drift apart. Once the network partition is resolved and the servers are able to communicate with each other again, consistency rules are re-applied and one server is chosen as the holder of the *current* data - *the data on the other servers is discarded*. For a caching application, this may result in some cache misses and delays in some processing but, since a cache is not the system of record for the data being processed, no data is lost.

When using a TSA configured for availability over consistency to support TCStore users, data loss is a real possibility - if a network partition occurs and clients are actively updating TSA servers on both sides of the partition, data loss *will* occur once the network partition is resolved. If the applications using TCStore are tolerant of missing/inconsistent data - not an easy task - then configuring for availability over consistency is appropriate for the TSA. However, if a TCStore dataset is used as the system of record, this data loss or other inconsistency is generally undesirable if not catastrophic. If the TSA is used for a TCStore dataset which is either a system of record or for which loss/inconsistency is an undesirable outcome, then the TSA **must** be configured for consistency over availability.

You may use a single TSA supporting both Ehcache and TCStore but the choice of availability versus consistency is a cluster-level configuration - both Ehcache and TCStore users in a TSA are subject to that configuration. If your Ehcache users need high availability and your TCStore users need data consistency, you must use a separate TSA for each user base.

External voter for two-server stripes

In certain topologies, mandating a voter majority for an active server election can introduce availability issues. For example, in a two-server stripe the majority quorum is two votes. If these servers were to become disconnected from each other due to a network partition or because of a server failure, the surviving server would not be able to promote itself to the active server as it requires two votes to win the election. Since one of the two servers is not available/reachable, the missing second vote will render the stripe unavailable.

Adding a third server is the best option. A three-server stripe can provide data redundancy and high availability at the same time even when one server fails. In this topology, if one server fails, there is still a majority of surviving servers (2 out of 3) to elect an active server. If adding a third server is not feasible, an option to get high availability without risking data consistency (via split-brain scenarios) is to use an external voter. This configuration cannot offer data redundancy (like a three-server stripe) if a server fails.

An external voter is a client that is able to cast a vote in an election for a new active server in cases where a majority of servers in a stripe are unable to reach a consensus on electing a new active server.

External voter configuration

The number of external voters must be defined in the server configuration. It is recommended that the total number of servers and external voters be kept as an odd number.

External voters must be registered with the servers in order to get added as voting members in their elections. If there are *n* voters configured in the server, then the first *n* voting clients requesting registration will be added as voters. Registration requests of other clients will be declined and put on hold until one of the registered voters becomes deregistered.

Voters can de-register themselves from the cluster so that the voting rights can be transferred to other clients waiting to get registered, if there are any. A voting client can deregister itself by using APIs or by getting disconnected from the cluster.

When a voting client gets disconnected from the server it will automatically get deregistered by the server. When the client reconnects, it will only get re-registered as a voter if another voter has not taken its place while this client was disconnected.

Server configuration

A maximum count for the number of external voters allowed can optionally be added to the `failover-priority` configuration if the cluster is tuned for `consistency`, as follows:

```
failover-priority=consistency:3
```

1	Here, the total number of voting clients is restrictred to three.

The failover priority setting and the specified maximum number of external voters across the stripes must be consistent and will be validated during the cluster configuration step. For more information on how to activate a cluster, see the "Settings" on page 59 section.

Client configuration

External voters can be of two variants:

1. Standalone voter

2. Clients using the voter library (client voter)

Standalone voter

An external voter can be run as a standalone process using the `start_tc_voter` script located in the `tools/voter/bin/` folder under the product installation directory. The script accepts the `<hostname>:<port>` for the cluster's servers as arguments.

Each `-connect-to` option argument must be a comma separated list of `<hostname>:<port>` combinations of servers within a single stripe. To register a multi-stripe cluster, multiple `-connect-to` options can be provided for each stripe.

Usage:

```
start-tc-voter.(sh|bat) -connect-to hostname:port[,hostname:port]... [-connect-to
hostname:port[,hostname:port]...]...
```

To connect the voter to a secure cluster, the path to the security root directory must also be specified using the `-security-dir` option. For more details on setting up security in a Terracotta cluster, see "SSL / TLS Security Configuration in Terracotta" on page 155.

Client voter

Any TCStore or Ehcache client can act as an external voter as well by using a voter library distributed with the kit. A client can join the cluster as a voter by creating a `TCVoter` instance and registering itself with the cluster.

Note:
The cluster must be activated using the config tool before a client voter can be registered with it.

When the voter is no longer required, it can be deregistered from the cluster either by disconnecting that client, or by using the `deregister` API.

```
import com.terracottatech.voter.EnterpriseTCVoterImpl;
import org.terracotta.voter.TCVoter;
TCVoter voter = new EnterpriseTCVoterImpl();            //1
voter.register("my-cluster-0"                           //2
               "<hostname>:<port>","<hostname>:<port>");  //3
...
voter.deregister("my-cluster-0")                        //4
```

1	Instantiate a `TCVoter` instance
2	Register the voter with a cluster by providing a cluster name …
3	and <hostname>:<port > combinations of all servers in the cluster.
4	Deregister from the cluster using the same cluster name that was used to register it.

To connect to a secure cluster, the voter must be instantiated using the overloaded constructor of `EnterpriseTCVoterImpl` that accepts the security root directory path.

Manual promotion with override voter

Since an external voter is itself a running process, there is no guarantee that it too will always be up and available. The moment the client voter leaves, the external voter leaves with it.

In the rare event that a failure occurs (i.e. a partition splitting the active and passive servers or the active server crashing/stopping) with the external voter no longer being present, none of the surviving servers can automatically become an active server. The servers will be stuck in a *suspended* state whereby operations from regular clients will be stalled.

In this scenario, a manual intervention is required in order to move the cluster out of this partitioned state by either (1) fixing the cause of the partition or (2) restarting the crashed server. If neither option is feasible, then a third option of manually promoting a server to the active state by casting an override vote from an external voter is required.

The voter process can be started in *override* mode to promote a single server to become an active server, when that server is stuck in an intermediate state. When the voter process is started in this mode it will connect to the specified server to be promoted and give it an override vote. The voter process will then exit. The voter process is started in override mode as follows:

```
start-tc-voter.(sh|bat) -vote-for <hostname>:<port>
```

Running this command will forcibly promote the server at `<hostname>:<port>` to be an active server (if it is stuck in that intermediate state).

Note:
This override voting will work even if external voters are not configured in the server configuration.

Warning:
Be cautious not to start two different override voters on both sides of the partition separately so that both sides win and cause a split-brain.

Server startup

When the failover priority of the stripes is tuned for consistency, it has an impact on server startup as well. In a multi-server stripe, when the servers are started up fresh, a server will not get elected as an active server until it gets votes from all of its peers. This will require all the servers of that stripe to be brought up. Bringing up regular voters is not going to help as they need to communicate with all the active servers in the cluster to get registered. But if bringing up the other servers is not feasible for some reason, then an override voter can be used to forcibly promote that server.

18 Connection Leasing

Why Leasing

When a client carries out a write with IMMEDIATE or STRONG consistency, the server ensures that every client that could be caching the old value is informed, and the write will not complete until the server can ensure that clients will no longer serve a stale value.

Where network disruptions prevent the server communicating with a client in a timely manner, the server will close that client's connection to allow the write to progress.

To achieve this, each client maintains a lease on its connections to the cluster. If a client's lease expires, the server may decide to close that client's connection. A client may also close the connection if it realises that its lease has expired.

Lease Duration

When selecting the duration of lease, consider the range of possible client to server round-trip latencies over a network connection that can be considered as functional. The lease should be longer than the largest possible such latency.

On a server that is heavily loaded, there may be some additional delay in processing a client's request for a lease to be extended. Such a delay should be added into the round-trip network latency.

In addition, leases are not renewed as soon as they are issued, instead the client waits until some portion of the lease has passed before renewing. A guideline suitable for the current implementation is that leases should be approximately 50% longer to allow for this.

Setting long leases, however, has the downside that, when clients are unreachable by a server, IMMEDIATE writes could block for up to the duration of a lease.

The default duration of lease is currently 150 seconds.

Lease Configuration

To configure the lease duration, use the `client-lease-duration` property in the config file, or during server startup.

19 Cluster Tool

The cluster tool is a command-line utility that allows administrators of the Terracotta Server Array to perform a variety of cluster management tasks. For example, the cluster tool can be used to:

- Obtain the running status of servers
- Dump the state of running servers
- Take backups of running servers
- Promote a suspended server on startup or failover
- Shut down an entire cluster
- Perform a conditional partial shutdown of a cluster having one or more passive servers configured for high availability (for upgrades etc.)

The cluster tool script is located in `tools/bin` under the product installation directory as `cluster-tool.bat` for Windows platforms, and as `cluster-tool.sh` for Unix/Linux.

Cluster Tool commands

The cluster tool provides several commands. To list them and their respective options, run `cluster-tool.sh` (or `cluster-tool.bat` on Windows) without any arguments, or use the option `-help`.

The following section provides a list of options common to all commands, and thus need to be specified before the command name:

Precursor options

1. `-verbose`

   This option gives you a verbose output, and is useful to debug error conditions. Default: false.

2. `-security-dir`

   This option can be used to communicate with a server which has TLS/SSL-based security configured. For more details on setting up security in a Terracotta cluster, see the section "Security Core Concepts" on page 156.

3. `-connection-timeout`

This option lets you specify a custom timeout value (in milliseconds) for connections to be established in cluster tool commands. Default: 10s.

4. `-request-timeout`

   This option lets you specify a request timeout value for operations. Default: 10s.

The "status" Command

The `status` command displays the status of a cluster, or particular server(s) in the same or different clusters.

Syntax:

```
status [-cluster-name <cluster-name>] [-format json]
  -connect-to <hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-cluster-name <cluster-name>`

  The name of the configured cluster.

- `-format json`

  Output in JSON format. Default is tabular.

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-s` option. When provided with option `-n`, a reachable server in the provided list will be used. Otherwise, the command will be individually executed on each server in the list.

Examples

- The example below shows the execution of a cluster-level `status` command.

```
./cluster-tool.sh status -cluster-name tc-cluster -connect-to localhost
| STRIPE: 1 |
+---------------------+-----------------------+---------------------------+
|    Server Name      |       Host:Port       |          Status           |
+---------------------+-----------------------+---------------------------+
|      server-1       |    localhost:9410     |          ACTIVE           |
|      server-2       |    localhost:9510     |          PASSIVE          |
+---------------------+-----------------------+---------------------------+
| STRIPE: 2 |
+---------------------+-----------------------+---------------------------+
|    Server Name      |       Host:Port       |          Status           |
+---------------------+-----------------------+---------------------------+
|      server-3       |    localhost:9610     |          ACTIVE           |
|      server-4       |    localhost:9710     |          PASSIVE          |
+---------------------+-----------------------+---------------------------+
```

- The example below shows the execution of a server-level `status` command. No server is running at `localhost:9910`, hence the `UNREACHABLE` status.

```
./cluster-tool.sh status -connect-to localhost:9410 -connect-to localhost:9510
-connect-to localhost:9910
+-----------------+--------------+-------------------+---------------------------+
|       Host-Port          |        Status        |   Member of Cluster      |
 Additional Information                |
+-----------------+--------------+-------------------+---------------------------+
|    localhost:9410        |        ACTIVE        |        tc-cluster        |
           -                           |
----------------------------------------------------------------------------------
|    localhost:9510        |       PASSIVE        |        tc-cluster        |
           -                           |
----------------------------------------------------------------------------------
|    localhost:9910        |    UNREACHABLE       |            -             |
localhost:9910=Connection refused;     |
+-----------------+--------------+-------------------+---------------------------+
Error (PARTIAL_FAILURE): Command completed with errors.
```

To learn more about server states, visit the section "Logical Server States" on page 17.

The "promote" command

The `promote` command can be used to promote a server stuck in a *suspended* state. For more information about suspended states, refer to the topics *Server startup* and *Manual promotion with override voter* in the section "Failover Tuning" on page 99.

Syntax:

```
promote -connect-to <hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port(s)` or `hostname(s)` (default port being `9410`) of running servers, each specified using the `-s` option. The command will be individually executed on each server in the list.

Note:
There is no cluster-wide flavor for this command.

Examples

- The example below shows the execution of the `promote` command on a server stuck in suspended state at `localhost:9410`.

  ```
  ./cluster-tool.sh promote -connect-to localhost
  Following sub-operations were successful:
    localhost:9410: Server promotion successful
  Command completed successfully.
  ```

- The example below shows the erroneous execution of a server-level `promote` command. The server running at `localhost:9510` is not in a suspended state to be promoted, hence the failure.

  ```
  ./cluster-tool.sh promote -connect-to localhost:9510
  Following sub-operations were unsuccessful:
    localhost:9510:
  com.terracottatech.tools.clustertool.exceptions.ClusterToolException:
  ```

```
    Promote command failed as the server is not in a suspended state
Error (FAILURE): Command failed.
```

The "dump" Command

The `dump` command dumps the state of a cluster, or particular server(s) in the same or different clusters. The dump of each server can be found in its logs.

Syntax:

```
dump [-cluster-name <cluster-name>] -connect-to <hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-cluster-name <cluster-name>`

  The name of the configured cluster.

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-connect-to` option. When provided with option `-cluster-name`, a reachable server in the provided list will be used. Otherwise, the command will be individually executed on each server in the list.

Examples

- The example below shows the execution of a cluster-level `dump` command.

  ```
  ./cluster-tool.sh dump -cluster-name tc-cluster -connect-to localhost:9410
  Contacting servers: [localhost:9410]
  Using reachable server: localhost:9410 to carry out the operation
  Following sub-operations were successful:
    localhost:9410: Dump successful
    localhost:9510: Dump successful
    localhost:9610: Dump successful
    localhost:9710: Dump successful
  Command completed successfully.
  ```

- The example below shows the execution of a server-level `dump` command. No server is running at `localhost:9510`, hence the dump failure.

```
./cluster-tool.sh dump -connect-to localhost:9410 -connect-to localhost:9510 -connect-to
 localhost:9910
Following sub-operations were successful:
  localhost:9410: Dump successful
  localhost:9510: Dump successful
Following sub-operations were unsuccessful:
  localhost:9910:
    org.terracotta.diagnostic.client.connection.DiagnosticServiceProviderException:
    com.terracotta.connection.api.DetailedConnectionException:
    java.util.concurrent.TimeoutException: localhost:9910=Connection refused;
Error (PARTIAL_FAILURE): Command completed with errors.
```

The "ipwhitelist-reload" Command

The `ipwhitelist-reload` command reloads the IP whitelist on a cluster, or particular server(s) in the same or different clusters. See the section "IP Whitelisting" on page 163 for details.

Syntax:

```
ipwhitelist-reload [-cluster-name <cluster-name>] -connect-to
<hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-cluster-name <cluster-name>`

  The name of the configured cluster.

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-connect-to` option. When provided with option `-cluster-name`, a reachable server in the provided list will be used. Otherwise, the command will be individually executed on each server in the list.

Examples

- The example below shows the execution of a cluster-level `ipwhitelist-reload` command.

```
./cluster-tool.sh ipwhitelist-reload -cluster-name tc-cluster -connect-to localhost
Contacting servers: [localhost:9410]
Using reachable server: localhost:9410 to carry out the operation
Following sub-operations were successful:
  localhost:9410: IP whitelist reload successful
  localhost:9510: IP whitelist reload successful
  localhost:9610: IP whitelist reload successful
  localhost:9710: IP whitelist reload successful
Command completed successfully.
```

- The example below shows the execution of a server-level `ipwhitelist-reload` command. No server is running at `localhost:9910`, hence the IP whitelist reload failure.

```
./cluster-tool.sh ipwhitelist-reload -connect-to localhost:9410 -connect-to
localhost:9510 -connect-to localhost:9910
Following sub-operations were successful:
  localhost:9410: IP whitelist reload successful
  localhost:9510: IP whitelist reload successful
Following sub-operations were unsuccessful:
  localhost:9910:
    org.terracotta.diagnostic.client.connection.DiagnosticServiceProviderException:
     com.terracotta.connection.api.DetailedConnectionException:
     java.util.concurrent.TimeoutException: localhost:9910=Connection refused;
Error (PARTIAL_FAILURE): Command completed with errors.
```

The "backup" Command

The `backup` command takes a backup of the running Terracotta cluster. The backup is taken on active servers only. Before taking backup of a cluster, `backup-dir` needs to be set on each server. For more details about this feature, see "Backup, Restore and Data Migration" on page 137.

Syntax:

```
backup -cluster-name <cluster-name> -connect-to <hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-cluster-name <cluster-name>`

  The name of the configured cluster.

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-connect-to` option. A reachable server in the provided server list will be used for connection.

Note:
There's no server-level flavor of this command, as backup works at the cluster level only.

Examples

- The example below shows the execution of a successful `backup` command. Note that the server at `localhost:9610` was unreachable.

```
./cluster-tool.sh backup -cluster-name tc-cluster -connect-to localhost:9710
-connect-to localhost:9410
Contacting servers: [localhost:9710, localhost:9410]
Following sub-operations were unsuccessful:
  localhost:9710:
org.terracotta.diagnostic.client.connection.DiagnosticServiceProviderException:
com.terracotta.connection.api.DetailedConnectionException:
java.util.concurrent.TimeoutException: localhost:9710=Connection refused;
Using reachable server: localhost:9410 to carry out the operation
PHASE 0: SETTING BACKUP NAME TO : 996e7e7a-5c67-49d0-905e-645365c5fe28
localhost:9710: TIMEOUT
localhost:9410: SUCCESS
localhost:9510: SUCCESS
localhost:9610: SUCCESS
PHASE (1/4): PREPARE_FOR_BACKUP
localhost:9710: TIMEOUT
localhost:9410: SUCCESS
localhost:9510: NOOP
localhost:9610: SUCCESS
PHASE (2/4): ENTER_ONLINE_BACKUP_MODE
localhost:9410: SUCCESS
localhost:9610: SUCCESS
PHASE (3/4): START_BACKUP
localhost:9410: SUCCESS
localhost:9610: SUCCESS
PHASE (4/4): EXIT_ONLINE_BACKUP_MODE
localhost:9410: SUCCESS
```

```
localhost:9610: SUCCESS
Command completed successfully.
```

- The example below shows the execution of a failed `backup` command.

```
./cluster-tool.sh backup -cluster-name tc-cluster -connect-to localhost:9610
Contacting servers: [localhost:9610]
Using reachable server: localhost:9610 to carry out the operation
PHASE 0: SETTING BACKUP NAME TO : 93cdb93d-ad7c-42aa-9479-6efbdd452302
localhost:9410: SUCCESS
localhost:9510: SUCCESS
localhost:9610: SUCCESS
localhost:9710: SUCCESS
PHASE (1/4): PREPARE_FOR_BACKUP
localhost:9410: SUCCESS
localhost:9510: NOOP
localhost:9610: SUCCESS
localhost:9710: NOOP
PHASE (2/4): ENTER_ONLINE_BACKUP_MODE
localhost:9410: BACKUP_FAILURE
localhost:9610: SUCCESS
PHASE (CLEANUP): ABORT_BACKUP
localhost:9410: SUCCESS
localhost:9610: SUCCESS
Error (FAILURE): Unable to complete backup.
```

The "shutdown" Command

The `shutdown` command shuts down a running Terracotta cluster. During the course of the shutdown process, it ensures that:

- Shutdown safety checks are performed on all the servers. Exactly what safety checks are performed will depend on the specified options and is explained in detail later in this section.
- All data is persisted to eliminate data loss.
- All passive servers are shut down first before shutting down the active servers.

The `shutdown` command follows a multi-phase process as follows:

1. Check with all servers whether they are OK to shut down. Whether or not a server is OK to shut down will depend on the specified shutdown options and the state of server in question.
2. If all servers agree to the shutdown request, all of them will be asked to prepare for the shutdown. Preparing for shutdown may include the following:
   a. Persist all data.
   b. Block new incoming requests. This ensures that the persisted data will be cluster-wide consistent after shutdown.
3. If all servers successfully prepare for the shutdown, a shutdown call will be issued to all the servers.

The first two steps above ensure an atomic shutdown to the extent possible as the system can be rolled back to its original state if there are any errors. In such cases, client-request processing will resume as usual after unblocking any blocked servers.

In the unlikely event of a failure in the third step above, the error message will clearly specify the servers that failed to shut down. In this case, use the `--force` option to forcefully terminate the remaining servers. If there is a network connectivity issue, the forceful shutdown may fail, and the remaining servers will have to be terminated using operating system commands.

Note:
The shutdown sequence also ensures that the data is stripe-wide consistent. Although, it is recommended that clients are shut down before attempting to shut down the Terracotta cluster.

Syntax:

```
shutdown [ -cluster-name <cluster-name> [-force | -now] ] -connect-to
<hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-cluster-name <cluster-name>`

  The name of the configured cluster.

- `-force`

  Forcefully shut down the cluster, even if the cluster is only partially reachable.

- `-now`

  Do an immediate shutdown of the cluster, even if clients are connected.

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being 9410) of running servers, each specified using the `-connect-to` option.

If the `-cluster-name` option is not specified, this command forcefully shuts down only the servers specified in the list. For clusters having stripes configured for high availability (with at least one passive server per stripe), it is recommended that you use the partial cluster shutdown commands explained in the section below, as they allow conditional shutdown, instead of using the `shutdown` variant without the `-cluster-name` option.

If the `-cluster-name` option is specified (i.e. a full cluster shutdown), this command shuts down the entire cluster. Servers in the provided list will be contacted for connectivity, and the command will then verify the cluster configuration with the given cluster name by obtaining the cluster configuration from the first reachable server. If all servers are reachable, this command checks if all servers in all the stripes are safe to shut down before proceeding with the command.

A cluster is considered to be safe to shut down provided the following are true:

- No critical operations such as backup and restore are going on.
- No Ehcache or TCStore clients are connected.

- All servers in all the stripes are reachable.

If either the `-force` or `-now` option is specified, this command works differently than above as follows:

- If the `-now` option is specified, this command proceeds with the shutdown even if clients are connected.
- If the `-force` option is specified, this command proceeds with the shutdown even if none of the conditions specified for safe shutdown above are met.

For all cases, the shutdown sequence is performed as follows:

1. Flush all data to persistent store for datasets or caches that have persistence configured.
2. Shut down all the passive servers, if any, in the cluster for all stripes.
3. Once the passive servers are shut down, issue a shutdown request to all the active servers in the cluster.

The above shutdown sequence is the cleanest way to shut down a cluster.

Examples

- The example below shows the execution of a cluster-level successful `shutdown` command.

  ```
  ./cluster-tool.sh shutdown -cluster-name tc-cluster -connect-to localhost:9410
  Contacting servers: [localhost:9410]
  Using reachable server: localhost:9410 to carry out the operation
  Shutting down cluster: tc-cluster
  STEP (1/3): Preparing to shut down
  STEP (2/3): Stopping all passive servers first
  STEP (3/3): Stopping all active servers
  Command completed successfully.
  ```

- The example below shows the execution of a cluster-level successful `shutdown` command that fails as one of the servers in the cluster was not reachable.

  ```
  ./cluster-tool.sh shutdown -cluster-name tc-cluster -connect-to localhost:9410
  Contacting servers: [localhost:9410]
  Using reachable server: localhost:9410 to carry out the operation
  Error (FAILURE): Timed out trying to reach the server
  Detailed Error Status for Cluster `tc-cluster` :
     ServerError{host='localhost:9510', Error='Timed out trying to reach the server'}.
  Unable to process safe shutdown request.
  Command failed.
  ```

- The example below shows the execution of a cluster-level successful `shutdown` command with the force option. Note that one of the servers in the cluster was already down.

  ```
  ./cluster-tool.sh shutdown -force -cluster-name tc-cluster -connect-to localhost:9410
  Contacting servers: [localhost:9410]
  Using reachable server: localhost:9410 to carry out the operation
  Timed out trying to reach the server
  Detailed Error Status for Cluster `tc-cluster` :
     ServerError{host='localhost:9510', Error='Timed out trying to reach the server'}.
  Continuing forced shutdown.
  Shutting down cluster: tc-cluster
  ```

```
STEP (1/3): Preparing to shut down
Timed out trying to reach the server
Detailed Error Status :
   ServerError{host='localhost:9510', Error='Timed out trying to reach the server'}.
Continuing forced shutdown.
STEP (2/3): Stopping all passive servers first
STEP (3/3): Stopping all active servers
Command completed successfully.
```

Partial Cluster Shutdown Commands

Partial cluster shutdown commands can be used to partially shut down nodes in the cluster without sacrificing the availability of the cluster. These commands can be used only on a cluster that is configured for redundancy with one or more passive servers per stripe. The purpose of these commands is to allow administrators to perform routine and planned administrative tasks, such as rolling upgrades, with high availability.

The following flavors of partial cluster shutdown commands are available:

- `shutdown-if-passive`
- `shutdown-if-active`
- `shutdown-all-passives`
- `shutdown-all-actives`

As a general rule, if these commands are successful, the specified servers will be shut down. If there are any errors due to which these commands abort, the state of the servers will be left intact.

From the table of server states described in "Logical Server States" on page 17, the following are the different active states that a server may find itself in:

- `ACTIVE`
- `ACTIVE_RECONNECTING`
- `ACTIVE_SUSPENDED`

Note:
In the following sections, the term 'active servers' means servers in any of the active states mentioned above, unless explicitly stated otherwise.

Similarly, the following are the passive states for a server:

- `PASSIVE_SUSPENDED`
- `SYNCHRONIZING`
- `PASSIVE`

Note:
In the following sections, the term 'passive servers' means servers in any of the passive states mentioned above, unless explicitly stated otherwise.

The "shutdown-if-passive" Command

The `shutdown-if-passive` command shuts down the specified servers in the cluster, provided the following conditions are met:

- All the stripes in the cluster are functional and there is one healthy active server with no suspended active servers per stripe.
- All the servers specified in the list are passive servers.

Syntax:

```
shutdown-if-passive -connect-to <hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-connect-to` option.

Note:
There's no cluster-level flavor of this command.

Examples

- The example below shows the execution of a successful `shutdown-if-passive` command.

```
./cluster-tool.sh shutdown-if-passive -connect-to localhost:9510
Contacting servers: [localhost:9510]
Stopping passive node(s): [localhost:9510] of cluster: tc-cluster
STEP (1/2): Preparing to shutdown
STEP (2/2): Stopping if Passive
Command completed successfully.
```

- The example below shows the execution of a failed `shutdown-if-passive` command, as it tried to shut down a server which is not a passive server.

```
./cluster-tool.sh shutdown-if-passive -connect-to localhost:9410
Contacting servers: [localhost:9410]
Error (FAILURE): Unable to process the partial shutdown request.
One or more of the specified server(s) are not in passive state
  or may not be in the same cluster
Discovered state of all servers are as follows:
Reachable Servers : 2
Stripe #: 1
Node: {localhost:9410} State: ACTIVE
Node: {localhost:9510} State: PASSIVE
Please check server logs for more details.
Command failed.
```

The "shutdown-if-active" Command

The `shutdown-if-active` command shuts down the specified servers in the cluster, provided the following conditions are met:

- All the servers specified in the list are active servers.
- All the stripes corresponding to the given servers have at least one server in 'PASSIVE' state.

Syntax:

```
shutdown-if-active -connect-to <hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-connect-to` option.

Note:
There's no cluster-level flavor of this command.

Examples

- The example below shows the execution of a successful `shutdown-if-active` command:

  ```
  ./cluster-tool.sh shutdown-if-active -connect-to localhost:9410
  Contacting servers: [localhost:9410]
  Stopping active node(s): [localhost:9410] of cluster: tc-cluster
  STEP (1/2): Preparing to shut down
  STEP (2/2): Shut down if active server
  Command completed successfully.
  ```

- The example below shows the execution of a failed `shutdown-if-active` command as the specified server was not an active server.

  ```
  ./cluster-tool.sh shutdown-if-active -connect-to localhost:9510
  Contacting servers: [localhost:9510]
  Error (FAILURE): Unable to process the partial shutdown request.
  One or more of the specified server(s) are not in active state
    or may not be in the same cluster.
  Reachable Servers : 2
  Stripe #: 1
  Node : {localhost:9410} State : ACTIVE
  Node : {localhost:9510} State : PASSIVE
  Please check server logs for more details
  Command failed.
  ```

The "shutdown-all-passives" Command

The `shutdown-all-passives` command shuts down all the passive servers in the specified cluster, provided the following is true:

- All the stripes in the cluster are functional and there is one active server in 'ACTIVE' state with no suspended active servers per stripe.

All passive servers in all the stripes of the cluster will be shut down when this command is run.

Syntax:

```
shutdown-all-passives -cluster-name <cluster-name> -connect-to
<hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-cluster-name <cluster-name>`

  The name of the configured cluster.

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-connect-to` option. These host(s) need not be passive servers.

Note:
There's no server-level flavor of this command, as it can be used only to shut down all the passive servers in the entire cluster.

The command shuts down all the passive servers in a multi-phase manner as follows:

1. Check with all servers whether it is safe to shut down as a passive server.
2. Flush any data that needs to be made persistent across all servers that are going down and block any further changes.
3. Issue a shutdown request to all passive servers if all passive servers succeed in step 2.
4. If any servers fail in step 2 or above, the shutdown request will fail and the state of the servers will remain intact.

Examples

- The example below shows the execution of a successful `shutdown-all-passives` command.

  ```
  ./cluster-tool.sh shutdown-all-passives -cluster-name tc-cluster -connect-to
  localhost:9410
  Contacting servers: [localhost:9410]
  Using reachable server: localhost:9410 to carry out the operation
  Stopping passive node(s): [localhost:9510] of cluster: tc-cluster
  STEP (1/2): Preparing to shutdown
  STEP (2/2): Stopping if Passive
  Command completed successfully.
  ```

The "shutdown-all-actives" Command

The `shutdown-all-actives` command shuts down the active server of all stripes in the cluster, provided the following are true:

- There are no suspended active servers in the cluster.
- There is at least one passive server in 'PASSIVE' state in every stripe in the cluster.

The active server of all stripes of the cluster will be shut down when this command returns success. If the command reports an error, the state of the servers will be left intact.

Syntax:

```
shutdown-all-actives -cluster-name cluster-name -connect-to
<hostname[:port]>,<hostname[:port]>...
```

Parameters:

- `-cluster-name cluster-name`

  The name of the configured cluster.

- `-connect-to <hostname[:port]>,<hostname[:port]>...`

  The `hostname:port`(s) or `hostname`(s) (default port being `9410`) of running servers, each specified using the `-connect-to` option. These host(s) need not be active servers.

Note:
There's no server-level flavor of this command as it can be used only to shut down all the active servers in the entire cluster.

The command shuts down all the active servers in a multi-phase manner as explained below:

1. Check with all servers whether they are safe to be shut down as active servers.
2. Flush any data that needs to be made persistent across all servers that are going down and block any further changes.
3. Issue a shutdown request to all active servers if they succeed in step 2.
4. If any servers fail in step 2 or above, the shutdown request will fail and the state of the servers will remain as before.

Examples

- The example below shows the execution of a successful `shutdown-all-actives` command. Note that the specified host was a passive server in this example. As the specified host is used only to connect to the cluster and obtain the correct state of all the servers in the cluster, the command successfully shuts down all the active servers in the cluster, leaving the passive servers intact.

  ```
  ./cluster-tool.bat shutdown-all-actives -cluster-name tc-cluster -connect-to
  localhost:9510
  Contacting servers: [localhost:9510]
  Using reachable server: localhost:9510 to carry out the operation
  Stopping active node(s): [localhost:9410] of cluster: tc-cluster
  STEP (1/2): Preparing to shut down
  STEP (2/2): Shut down if active server
  Command completed successfully.
  ```

20 Importing and Exporting Datasets

Overview

Dataset records and their cells can be exported to file in both Apache Parquet format and TSON format. Records and cells can also be imported into a dataset from a TSON file. This import-export capability is provided through both a command-line utility and a TCStore API via libraries and scripts that are distributed with the kit.

Parquet is a commonly used, open source, column-oriented, binary data format designed for efficient data storage and retrieval. As a binary format, Parquet files are consumed by software systems, which in turn target other storage systems and formats.

The TSON format is a kind of *typed* JSON where each data value is described with a type declaration wherever it appears within the body of the file.

Import-Export Tool

The Import-Export Tool is executed by running the appropriate *import-export-tool script* located in the `tools/import-export/bin` folder inside the Terracotta installation directory:

- `import-export-tool.bat` - used on Windows platforms
- `import-export-tool.sh` - used on Unix/Linux platforms.

Import-Export Tool script executions utilize the following syntax:

```
import-export-tool.sh|bat [<common_options>] <command> <command_specific_options>
```

Note that the `common_options` are optional, while one or more of the `command_specific_options` are required.

The supported `commands` include:

Command	Description
"export-parquet" on page 123	Export records from a dataset into a parquet file.
"export-tson" on page 126	Export records from a dataset into a TSON file.
"import-tson" on page 127	Import records from a TSON file into a dataset.

Common Command Options

Each Import-Export Tool command supports a unique set of options (detailed in the sections throughout this document). All commands support the following common options:

Option	Description	Default
`-connection-timeout`	Timeout value for connections to be established.	10 seconds

Option	Description	Default
`-security-dir`	Specifies the location of the security root directory folder. Used to communicate with a server that is configured with any of the supported security schemes (e.g. TLS/SSL). For more details on configuring security in a Terracotta cluster, see "Security Core Concepts" on page 156 and "Cluster Security" on page 161.	
`-verbose`	Generates a verbose output. Useful for debugging error conditions.	false
`-help`	Displays help information for commands and their options.	

Import-Export Tool Commands

The following sections describe each supported command and provide examples on their usage.

Note:
When running the import-export tool, connecting to a Terracotta server is established via the `-connect-to` option and specifying a URI connection string (e.g. `terracotta://<host>:<port>`)

Exporting a Dataset to a Parquet File

The `export-parquet` command is used to export a dataset's records to a parquet file. A sample of the dataset's records (`-schema-sample-size`) is used to construct the parquet file's schema, which is based on the unique set of cells contained with the sampled records. Useful features include filtering the returned records via the `-filter-cell-name/type` option, explicitly excluding or including cells in the exported records, and truncating and nullifying String and Byte array type cells respectively based on size and length of those cell values.

Syntax:

```
export-parquet -connect-to <connectionURI>
       -dataset-name <datasetName>
       -dataset-type <datasetType>
       -output-folder <outputFolder>
       [-filter-cell-name <cellName>
        -filter-cell-type <cellType>
        -filter-low-range <lowValue>
        -filter-high-range <highValue>]
       [-schema-sample-size <schemaSampleSize>]
       [-append-cell-type]
       [-max-columns <maxColumns>]
       [-no-abort-when-exceed-max-columns]
       [-multi-output-files]
       [-max-string-length <maxStringLength>]
       [-max-byte-length <maxByteLength>]
       [-include-cells <cellname>, <celltype> [, ...]]
       [-exclude-cells <cellname>, <celltype> [, ...]]
       [-exclude-empty-records]
       [-log-stream-plan]
```

Option	Description
`-append-cell-type`	When constructing the parquet schema's field names, always append the cell's Type to the cell's Name (default: false - only append when required in order to avoid field name clashes)
`-connect-to`	Server URI to connect to
`-dataset-name`	Name of dataset to export
`-dataset-type`	Type of dataset to export [BOOL | CHAR | INT | LONG | DOUBLE | STRING]
`-exclude-cells`	A comma-separated list of cell definitions as <cellname, celltype> to be excluded from the export. Include cells (-include-cells) takes precedence over exclude cells
`-exclude-empty-records`	Do not export records that contain zero cells
`-filter-cell-name`	Cell name used as a range filter to apply to the queried dataset records
`-filter-cell-type`	Cell type of the range filter [INT | LONG | DOUBLE]
`-filter-high-range`	High value for the range filter
`-filter-low-range`	Low value for the range filter
`-help`	Help
`-include-cells`	A comma-separated list of cell definitions as <cellname, celltype> to be included in the export. Only these cells will be exported
`-log-stream-plan`	Log the details of the stream plan
`-max-byte-length`	For byte array values, the maximum byte length in byte count that will be exported, null otherwise (default: all byte arrays are exported regardless of length)

Option	Description
`-max-columns`	Maximum allowed number of columns (i.e. unique cell definitions) in the output file (default: 800 columns)
`-max-string-length`	For string values, the maximum string length in number of characters that will be exported, truncated otherwise (default: no strings are truncated)
`-multi-output-files`	Generate multiple output files when the number of cells exceed the maximum allowed number of columns (default: false - write all cells to a single file)
`-no-abort-when-exceed-max-columns`	Do not abort the export when the number of cells exceeds the maximum allowed number of columns
`-output-folder`	Output folder where exported file(s) will be written
`-schema-sample-size`	Number of dataset records to query upon which the schema will be based (default: 5 records)

Examples

1. Exporting an entire dataset to a parquet file:

```
import-export-tool.sh export-parquet -connect-to terracotta://localhost:9410
-dataset-name DS1 -dataset-type LONG
-output-folder /path/to/output_folder
```

2. Exporting a subset of a dataset to a parquet file using a filter cell:

```
import-export-tool.sh export-parquet -connect-to terracotta://localhost:9410
-dataset-name DS1 -dataset-type LONG
-output-folder /path/to/output_folder -filter-cell-name MyLongCell -filter-cell-type
 LONG -filter-low-range 10 -filter-high-range 50
```

3. Exporting a subset of a dataset to a parquet file using a filter cell and only including specific cells in the output for each record.

```
import-export-tool.sh export-parquet -connect-to terracotta://localhost:9410
-dataset-name DS1
-dataset-type LONG -output-folder /path/to/output_folder -filter-cell-name MyLongCell
 -filter-cell-type LONG -filter-low-range 10
-filter-high-range 50 -include-cells
MyLongCell,LONG,MyDoubleCell,DOUBLE,MyBooleanCell,BOOL
```

Note:
For parquet export, when specifying one or more `-include-cells` and a `-filter-cell`, the filter cell/type must also appear in the included-cells list.

Exporting a Dataset to a TSON File

The `export-tson` command is used to export a dataset to a TSON-formatted file. The file can be compressed if necessary.

Syntax:

```
export-tson -connect-to <connectionURI>
       -dataset-name <datasetName>
       -dataset-type <datasetType>
       -output-file <outputFile>
       [-filter-cell-name <cellName>
        -filter-cell-type <cellType>
        -filter-low-range <lowValue>
        -filter-high-range <highValue>]
       [-max-string-length <maxStringLength>]
       [-max-byte-length <maxByteLength>]
       [-include-cells <cellname>, <celltype> [, ...]]
       [-exclude-cells <cellname>, <celltype> [, ...]]
       [-pretty-print]
       [-compress]
       [-exclude-empty-records]
       [-log-stream-plan]
```

Option	Description
`-compress`	Compress the generated output file after export
`-connect-to`	Server URI to connect to
`-dataset-name`	Name of dataset to export
`-dataset-type`	Type of dataset to export [BOOL | CHAR | INT | LONG | DOUBLE | STRING]
`-exclude-cells`	A comma-separated list of cell definitions as <cellname, celltype> to be excluded from the export. Include cells (-include-cells) takes precedence over exclude cells
`-exclude-empty-records`	Do not export records that contain zero cells
`-filter-cell-name`	Cell name used as a range filter to apply to the queried dataset records
`-filter-cell-type`	Cell type of the range filter [INT | LONG | DOUBLE]
`-filter-high-range`	High value for the range filter
`-filter-low-range`	Low value for the range filter
`-help`	Help

Option	Description
`-include-cells`	A comma-separated list of cell definitions as <cellname, celltype> to be included in the export. Only these cells will be exported
`-log-stream-plan`	Log the details of the stream plan
`-max-byte-length`	For byte array values, the maximum byte length in byte cou+nt that will be exported, null otherwise (default: all byte arrays are exported regardless of length)
`-max-string-length`	For string values, the maximum string length in number of characters that will be exported, truncated otherwise (default: no strings are truncated)
`-output-file`	Full pathname of output file to create and write the exported dataset records to (parent folder must exist)
`-pretty-print`	Include line breaks and indentations when writing records to the file

Examples

1. Exporting an entire dataset to a TSON file:

   ```
   import-export-tool.sh export-tson -connect-to terracotta://localhost:9410
   -dataset-name DS1 -dataset-type LONG
   -output-file path\to\file\myfile.tson
   ```

2. Exporting an entire dataset to a compressed TSON file:

   ```
   import-export-tool.sh export-tson -connect-to terracotta://localhost:9410
   -dataset-name DS1 -dataset-type LONG
   -output-file path\to\file\myfile.tson.gz -compress
   ```

3. Exporting an entire dataset to a TSON file truncating all string values greater than 256 characters long and excluding all byte array values greater than 1024 bytes and excluding all data for a specific cell:

   ```
   import-export-tool.sh export-tson -connect-to terracotta://localhost:9410
   -dataset-name DS1 -dataset-type LONG
   -output-file path\to\file\myfile.tson -max-string-length 256 -max-byte-length 1024
    -exclude-cells MyBooleanCell,BOOL
   ```

Importing Data from a TSON File

The `import-tson` command is used to add records and cells into an existing dataset by importing them from a TSON-formatted file, which can be optionally compressed.

Note:
When importing a TSON file into a dataset, the target dataset must already exist within the cluster. If the specified dataset does not exist, the import operation will fail.

Syntax:

```
import-tson -connect-to <connectionURI>
       -dataset-name <datasetName>
       -dataset-type <datasetType>
       -input-file <inputFile>
       [-compressed]
       [-clear-dataset]
       [-exclude-empty-records]
```

Option	Description
`-clear-dataset`	Delete all records from the target dataset before performing the import
`-compressed`	Specifies that the input file is compressed
`-connect-to`	Server URI to connect to
`-dataset-name`	Name of dataset to import
`-dataset-type`	Type of dataset to import [BOOL | CHAR | INT | LONG | DOUBLE | STRING]
`-exclude-empty-records`	Do not import records that contain zero cells
`-help`	Help
`-input-file`	Full pathname of TSON-formatted input file whose records will be added to the specified dataset

Examples

1. Import a TSON file into a dataset, first clearing all contents from the target dataset:

```
import-export-tool.sh import-tson -connect-to terracotta://localhost:9410
-dataset-name DS2 -dataset-type LONG
-input-file path\to\input_file.tson -clear-dataset
```

2. Import a compressed TSON file into a dataset without clearing the target dataset's contents and excluding any records in the import file which have zero cells (i.e. empty records):

```
import-export-tool.sh import-tson -connect-to terracotta://localhost:9410
-dataset-name DS2 -dataset-type LONG
-input-file path\to\input_file.tson.gz -compressed -exclude-empty-records
output results:
Import Result: Success
1,000 records processed.
5 empty records (with no cells) were omitted
17 records failed to be added to the Dataset.
```

Note:
When importing records without first clearing the target dataset, any records in the import file that have existing keys in the target dataset will not be imported. The count of these skipped records will appear in the output results of the import operation as 'records failed to be added to the Dataset'.

Import-Export API

The ability to import and export datasets to and from a terracotta cluster can be performed by any TCStore client using the import-export libraries distributed with the kit. The following code sample shows the required import paths for the key import-export classes:

```
import com.terracottatech.store.importexport.api.parquet.ParquetDatasetExport; // (1)
import com.terracottatech.store.importexport.api.tson.TSONDatasetExport;      // (2)
import com.terracottatech.store.importexport.api.tson.TSONDatasetImport;      // (3)
```

1	Exporting to Parquet requires imports for `ParquetDatasetExport` (shown), `ParquetExportOptions` and `ParquetExportStats`.
2	Exporting to TSON requires imports for `TSONDatasetExport` (shown), `TSONExportOptions` and `TSONExportStats`.
3	Importing from TSON requires imports for `TSONDatasetImport` (shown), `TSONImportOptions` and `TSONImportStats`.

The following example illustrates how to export a dataset to a Parquet file format using the import-export API:

```
try (DatasetManager dsManager =
      DatasetManager.clustered(URI.create(connectionURI)).build()) { // (1)
  ParquetExportOptions options = new ParquetExportOptions();         // (2)
  options.setDatasetName("DS1");                                     // (3)
  options.setDatasetType(LONG);                                      // (4)
  options.setOutputFolder(Paths.get(outputFolderFullPath));          // (5)
  ParquetDatasetExport exporter
      = new ParquetDatasetExport(dsManager, options);                // (6)
  ParquetExportStats stats = exporter.exportDataset();               // (7)
  System.out.println(stats.toString());                              // (8)
}
```

1	Create a `DatasetManager` against a server in the cluster supplying a URI connection string (e.g. `terracotta://<hostname>:<hostport>`).
2	Create an `ExportOptions` instance corresponding to the desired file format (`ParquetExportOptions` in this example).

3	Specify the name of the dataset from which to export records (`DS1` in this example).
4	Specify the `Type` of the dataset identified in **3** above.
5	Specify the full `Path` of an existing folder where the generated output file will be created and into which records will be written.
6	Create a `DatasetExport` instance corresponding to the desired format (`ParquetDatasetExport` in this example) supplying the `DatasetManager` and `ExportOptions` instances.
7	Perform the export by calling `exportDataset()`.
8	Understand the results of the completed export operation contained within the returned `ExportStats` instance (`ParquetExportStats` in this example):

```
Export Result: Success
Output Files (1):
C:\temp\dataset1_2022-08-11-09-24-49-777.parquet
1,000 records processed.
1,000 complete records written to parquet file
0 partial records written to parquet file
0 entire records NOT written to parquet file
0 records failed writing to parquet file
0 empty records excluded writing to parquet file
0 string values were truncated
0 large-size byte arrays were omitted
```

Note:
In the above example, the system automatically created the export file with name `dataset1_2022-08-11-09-24-49-777.parquet`. In fact, for Parquet export, the system will always construct the filename. However, when exporting in TSON format, the name of the generated file must be supplied by the client, as illustrated in the next example.

The following example illustrates how to export a dataset to TSON file format using the import-export API. The example also illustrates how to configure cell filtering:

```
try (DatasetManager dsManager =
      DatasetManager.clustered(URI.create(connectionURI)).build()) {      // (1)
  TSONExportOptions options = new TSONExportOptions();                    // (2)
  options.setDatasetName("DS1");                                          // (3)
  options.setDatasetType(LONG);                                           // (4)
  options.setOutputFileName(outputFilenameFullPath);                      // (5)
  options.setFilterCell("myFilterCell", LONG);                            // (6)
  options.setFilterLowValue(0);                                           // (7)
  options.setFilterHighValue(50);                                         // (8)
  TSONDatasetExport exporter = new TSONDatasetExport(dsManager, options); // (9)
  TSONExportStats stats = exporter.exportDataset();                       // (10)
  System.out.println(stats.toString());                                    // (11)
}
```

1	Create a `DatasetManager` against a server in the cluster supplying a URI connection string (e.g. `terracotta://<hostname>:<hostport>`).
2	Create an `ExportOptions` instance corresponding to the desired file format (`TSONExportOptions` in this example).
3	Specify the name of the dataset from which to export records (`DS1` in this example).
4	Specify the `Type` of the dataset identified in **3** above.
5	Specify the full path filename which the system will create and into which records will be written. The file's parent directory must exist.
6	Specify a named cell and its `Type` that is present in the dataset identified in **3** above. That cell will be used to filter the exported records.
7	Specify the low range numeric value of the filter cell for which records containing the specified filter cell and whose value is greater than or equal to the low range will be included in the export file.
8	Specify the high range numeric value of the filter cell for which records containing the specified filter cell and whose value is less than the high range will be included in the export file.
9	Create a `DatasetExport` instance corresponding to the desired format (`TSONDatasetExport` in this example) supplying the `DatasetManager` and `ExportOptions` instances.
10	Perform the export by calling `exportDataset()`.
11	Understand the results of the completed export operation contained within the returned `ExportStats` instance (`TSONExportStats` in this example):

```
Export Result: Success
1,000 records processed.
0 string values were truncated
0 large-size byte arrays were omitted
```

```
0 empty records (with no cells) were omitted
```

The following example illustrates how to import dataset records contained within a TSON-formatted file using the import-export API:

```
try (DatasetManager dsManager =
      DatasetManager.clustered(URI.create(connectionURI)).build()) { // (1)
  TSONImportOptions options = new TSONImportOptions();               // (2)
  options.setDatasetName("DS2");                                     // (3)
  options.setDatasetType(LONG);                                      // (4)
  options.setInputFileName(inputFilenameFullPath);                   // (5)
  options.setCompressed(false);                                      // (6)
  options.setClearDatasetBeforeImport(true);                         // (7)
  TSONDatasetImport exporter
      = new TSONDatasetImport(dsManager, options);                   // (8)
  TSONImportStats stats = exporter.importDataset();                  // (9)
  System.out.println(stats.toString());                              // (10)
}
```

1	Create a `DatasetManager` against a server in the cluster supplying a URI connection string (e.g. `terracotta://<hostname>:<hostport>`).
2	Create an `ImportOptions` instance corresponding to the desired file format (`TSONImportOptions` in this example).
3	Specify the name of an existing dataset into which records will be added (`DS2` in this example).
4	Specify the `Type` of the dataset identified in **3** above.
5	Specify the full path filename of the file that you want to import.
6	Specify whether the input file identified in **5** above has been compressed (both ZIP and GZIP formats are supported).
7	Specify if all records present in the target dataset identified in **3** above should first be deleted before the new records are added from the import file.
8	Create a `DatasetImport` instance corresponding to the desired format (`TSONDatasetImport` in this example) supplying the `DatasetManager` and `ImportOptions` instances.
9	Perform the import by calling `importDataset()`.

10	Understand the results of the completed import operation contained within the returned `ImportStats` instance (`TSONImportStats` in this example):

```
Import Result: Success
1,000 records processed.
0 empty records (with no cells) were omitted
0 records failed to be added to the Dataset.
```

21 Licensing

This document describes the installation and update procedures for Terracotta Ehcache and Terracotta licenses.

Installing a license

A Terracotta license is installed on a Terracotta cluster using the config tool either:

- during the cluster activation process using the `activate` command, or
- any later desired time by using the `set` command

These commands ensure that:

- The license is a valid Software AG license.
- The license has not expired already.
- The Terracotta configuration files do not violate the license.

The following example activates a Terracotta cluster and installs the license file:

```
./config-tool.sh activate -license-file license.xml -cluster-name tc-cluster -connect-to
 localhost:9410
Activating nodes: localhost:9410
License installation successful
Restarting nodes: localhost:9410
Node: localhost:9410 has restarted in state: ACTIVE
All nodes came back up
Command successful!
```

See the set command for a detailed explanation of the command usage.

License expiration

License expiry checks are done every midnight (UTC time) to ensure that the license in use did not expire. Midnight here is the time at the start of the day, i.e. '00:00' hours. As an example, for a license which is valid till December 31, the midnight check on December 31 will pass, but the check on January 1 midnight will fail, and license will be deemed as expired. When a license expires, a warning message like the following will be logged every 30 minutes in the server logs:

```
ATTENTION!! LICENSE expired. Time since expiry 1 day(s)
```

The license must be renewed within 7 days of expiry. If it is not done, the cluster will be shut down with the following message in the server logs:

```
Shutting down the server as a new license is not installed within 7 days.
```

License renewal

If your license expires, a new license can be obtained by contacting Software AG support. The new license can then be installed using the config tool `set` command as follows:

```
./config-tool.sh set -setting license-file=license.xml -connect-to localhost:9410
Connecting to: localhost:9410 (this can take time if some nodes are not reachable)
License validation passed: configuration change(s) can be applied
Applying new configuration change(s) to activated cluster: localhost:9410
Command successful!
```

See the set command for a detailed explanation of the command usage.

22 Backup, Restore and Data Migration

Overview of Backup and Restore

The Backup and Restore feature enables you as an administrator of a Terracotta cluster to take a backup of the cluster and restore it from the backed up data when required.

Terracotta supports two ways of taking a backup:

1. Online backup using the cluster-tool. This is the recommended method.
2. Manual offline backup

Restore and Ehcache data migration are manual offline processes.

Note:

Migration of TCStore data is currently not supported.

When a passive server starts and discovers it has data, the data is automatically backed up for safety reasons. However, this data is not cluster-wide consistent, and **must not** be used for restoration. Refer to the section “Active and Passive Servers” on page 13 for more information.

Terms

Backup and Restore : Taking a snapshot of the cluster data such that it can later be installed back on the same cluster, bringing it back to the initial state.

Data Migration : Taking a snapshot of the cluster data, but installing it on a *different* cluster, bringing it to the state of the original cluster. Data Migration is also desirable in cases when only Ehcache data is needed, and not the platform data.

Data Directory Structure

Following is a sample data directory structure of a server containing Ehcache and TCStore data:

```
/tmp/data1/
└── server-1
    ├── ehcache
    │   └── frs
    │       └── default-frs-container
    │           ├── default-cachedata
    │           │   ├── FRS.lck
    │           │   ├── frs.backup.lck
    │           │   └── seg000000000.frs
    │           └── metadata
    │               ├── FRS.lck
    │               ├── frs.backup.lck
    │               └── seg000000000.frs
    ├── platform-data
    │   ├── entityData
    │   │   ├── FRS.lck
    │   │   ├── frs.backup.lck
    │   │   ├── seg000000000.frs
    │   │   └── seg000000001.frs
    │   │
    │       └── transactionsData
    │           ├── FRS.lck
    │           ├── frs.backup.lck
    │           ├── seg000000000.frs
    │           └── seg000000001.frs
    └── store
        ├── data
        │   ├── FRS.lck
        │   ├── frs.backup.lck
        │   └── seg000000000.frs
        └── meta
            ├── FRS.lck
            ├── frs.backup.lck
            └── seg000000000.frs
```

where:

1. `/tmp/data1` is the data directory path (for a given data directory) defined in the server config file
2. `server-1` is the server name defined in the server config file
3. `ehcache` is the directory containing Ehcache data
4. `platform-data` is the directory containing platform specific logs
5. `store` is the directory containing TCStore data

Online Backup

Online backup of a Terracotta cluster is performed by the cluster-tool, and is the recommended method to take a backup. The following section describes the online backup feature and the process:

Configuring the Backup feature

To be able to take cluster backups, a backup directory must be configured. If the directory specified by the backup location path is not present, it will be created during backup. This can be done using one of the following ways:

1. Using the CLI option `-backup-dir` during server startup
2. Using the `backup-dir` property in the config property file during server startup
3. Using the config tool `set` command to set the `backup-dir` property at runtime

Prerequisites

Before proceeding with the online backup, ensure that:

1. At least one server in each stripe is up and running.
2. The servers have read and write permissions to the backup location.
3. Backup location has enough space available to store the backup data.

4. cluster-tool has fast connectivity to all the servers and the cluster is not heavily loaded with application requests.

Taking an online Backup

A backup is taken using the cluster-tool. Visit "Cluster Tool" on page 107 for details on the `backup` command. If the backup fails for some reason, you can check the server logs for failure messages. Additionally, running the `backup` command using the `-verbose` option might help.

Backup directory structure

The following diagram shows an example of the directory structure that results from a backup:

```
/tmp/backup1/
└── 7c868f83-5075-4b32-bef5-56f29fdcc6f0
    └── server-1
        └── datadir
            ├── ehcache
            │   └── frs
            │       └── default-frs-container
            │           ├── default-cachedata
            │           │   └── seg000000000.frs
            │           └── metadata
            │               └── seg000000000.frs
            ├── platform-data
            │   ├── entityData
            │   │   ├── seg000000000.frs
            │   │   └── seg000000001.frs
            │   └── transactionsData
            │       ├── seg000000000.frs
            │       └── seg000000001.frs
            └── store
                ├── data
                │   └── seg000000000.frs
                └── meta
                    └── seg000000000.frs
```

where:

1. `/tmp/backup1/` is the backup location defined in the config file
2. `7c868f83-5075-4b32-bef5-56f29fdcc6f0` is an ID created by the backup command to uniquely identify a backup instance
3. `server-1` is the server name
4. `datadir` is the data directory name (for a given data directory) defined in the server config file

Offline Backup

In the rare scenario when an online backup cannot be taken, an offline backup can be taken. The process is described as follows:

Taking an offline Backup

Follow the steps in the specified order to back up cluster data:

1. Shut down the cluster, while taking a note of the current active servers.
2. Copy the contents of the required data directories of **all** the servers which were actives prior to the shutdown to a desired location.
3. Name the directories in the manner described in the "Backup directory structure" on page 140 section above. Although this step is optional, it helps identify different instances of backup, and keeps the restore steps consistent for online and offline backup procedures.
4. Save the config files as well. These files will be used to start the stripes after a restore is performed.

Restore

The restore operation is a manual operation. During the Restore operation, you use standard operating system mechanisms to copy the complete structure (directories, subdirectories and files) of the backup into the original location. Some small structural and/or naming changes are required in the restored directories after the copy, as described in the sections below.

Note:
Restoring cache data will bring back cache entries which might have become stale by the time a restore is finished.

Performing a Restore

Before you start the Restore operation, ensure that all activity has stopped on the cluster and that the cluster is not running.

If you compare the structure of the backup under `/tmp/backup1` with the original structure under `/tmp/data1` (see both structural diagrams above), you will see some differences. You will also see that this is a single stripe cluster. Therefore, when you copy the `/tmp/backup1/<backup-name>` directory structure back to `/tmp/data1`, you need to make the following changes:

1. First choose a server as the active server for your stripe.
2. Note down the `name` of that server.
3. Create an empty directory for each path specified by the data directory. This will be the target directory for your restored data. Repeat this step for every data directory path specified in your config file.
4. Create a sub-directory with the name of the server under the data directories created above. For example, if the `name` attribute is `server-2` for the chosen active server for this stripe and the location specified for the data directory `datadir` is `/tmp/data1`, your target directory should look like `/tmp/data1/server-2`.

5. From the backup, copy the contents of `<server-name>/<data-directory>` to this newly created directory. For example, in the example given above, copy from `/tmp/backup1/<backup-name>/server-1/data` to `/tmp/data1/server-2`

6. Start the server with the newly created data directory with the config file which was backed up from the original cluster.

7. You can now bring up the passive servers in the stripe. Please note that you don't need to copy the backup data to the passive servers as they will automatically receive the data when they synchronize with the active server. It is advisable to remove any old data on the passive servers before you bring up the passive servers.

8. Repeat the above steps for other stripes in the cluster.

Data Migration of Ehcache data

Note:
As noted above, data migration is currently not available for TCStore data.

Data migration can be performed to move Ehcache data to a new cluster without moving the platform data. Please note that only restartable caches contained in a restartable cache manager can be recovered. Since the data migration works at the data directory level, all the data of all restartable cache managers that use the same data directory will be recovered together.

How to perform an offline data migration

Follow the steps in the specified order to perform a migration of cluster data:

1. Shut down the source cluster and copy the contents of all `ehcache` directories from all required data directories of **all** active servers in the cluster. You can skip copying data directories containing restartable cache managers that you do not wish to migrate.

2. Start the target cluster (you can just start the active servers at this time) with the same number of stripes as the source cluster. Create the desired cache manager configuration using a client. The cluster URI (including the cluster tier manager name for the cache manager) can be different in the new cluster. If the name part of the URI is different, specify the old name as the restart identifier when using the cache manager configuration API, so that the system can map the data corresponding to a given cache manager correctly. If there are more than one cache managers under the same data directory, use the configuration API to create all the cache managers in the target cluster.

   For related information, see the section *Fast Restartability* of the *Ehcache API Developer Guide*.

3. Shut down the target cluster and copy the data to the matching data directories. The data directory paths can be different on the target cluster, but must have sufficient space to contain the data being copied over.

4. Once the data is available in all the stripes, you can start the target cluster. It now loads all the cache data that was moved from the source cluster.

Technical Details

Causal and Sequential Consistency across stripes

Since TCStore and Ehcache support only causal consistency (per key) and sequential consistency (across keys for a single thread of execution), the backup image across the cluster (be it single stripe or multi-stripe) must be consistent cluster wide for the point-in-time when the backup was taken.

For instance, suppose a single thread of execution from a single client synchronously made changes to keys A, then B, then C and then D in that order. Now if the backup was captured when the client had made changes to C, intuitively the backup MUST have all the previous changes made to A and B, regardless of the stripe where those mutations occurred. Thus on a restore of this point-in-time backup, if the restored data has C, then it MUST contain the changes made to A and B. Of course, it is to be expected that such a restoration may have permanently lost D, due to the point-in-time nature of restoring from backups.

As another example, say a system had long keys from 1 to 1000 and mutated them one by one exactly in that order. If the backup had 888 as the largest key, then all keys from 1 to 887 MUST also exist in the backup.

Causal consistency (per key) is always implied, as a key is always within a stripe. The backup taken must be consistent for a point in time snapshot, which implies that when a snapshot is taken, all mutations/changes that happen in the system AFTER the snapshot is taken MUST not reflect in the backup.

Consistency of multiple FRS logs within a stripe

Since platform data is also backed up, there are at least two FRS logs that needs to be backed up in a consistent fashion even within a single stripe.

23 Migrating from older Terracotta versions to 10.7

A major feature of the 10.7 release is dynamic configuration of the Terracotta cluster, which changes cluster startup and configuration mechanisms. This documents briefly lists the new concepts, tooling information, and the steps to be followed to migrate from an earlier 10.x to a 10.7 cluster.

Before proceeding with this document, please make sure that you have familiarized yourself with the material presented in "Configuration Terms and Concepts" on page 33.

What has changed

1. **Node startup mechanism**

   `start-tc-server.(bat|sh)` does not support a tc-config XML file anymore. Several new options have been added to this script. More information can be found in the startup script section.

2. **Cluster configuration format**

   Cluster configuration can no more be specified using tc-config XML files. The old tc-configs can be migrated to the new configuration format using config converter tool.

3. **Cluster configuration mechanism**

   The cluster tool `configure` command has been removed in favor of the config tool `activate` command. The cluster tool `reconfigure` command has been removed in favor of the config tool `set` command.

Converting old configuration files

The config converter tool can be used to convert from tc-config XML files used with Terracotta versions up to 10.5 to the new configuration format. The tool can be located under `tools/upgrade/bin` in the product installation directory, and has the following usage:

```
> config-converter.(bat|sh) convert -tc-config <tc-config>,<tc-config>... [
  -stripe-names <foo>,<bar>,<baz>...]( -format directory [-license <license-file>]
  -new-cluster-name <new-cluster-name> | -format properties [-new-cluster-name
<new-cluster-name>])
  [-destination <destination-dir>] [-force]
```

Warning:
Do not forget to assign stripe names in the same order of the xml config files, otherwise random default names will be generated for you.

When the output format is config directory (i.e. `-format directory` or no `-format` specification), a license file can be supplied.

Note:
In a fresh install of Terracotta using the Software AG installer, the license file can be found under the Terracotta installation root.

This way, the generated config directory will be ready-for-use by Terracotta clients. This is the recommended output format. Note that there is no change in the license file, or the licensing policies. Thus, existing Terracotta licenses will continue to work with 10.7 until their expiration.

When the output format is config properties (i.e. `-format properties`), the generated config file contains information from all the tc-config XML files, and can also be used to start up the nodes. However, an additional cluster activation step will be required after the migrated cluster is started using this config file.

Migrating old data

If you want to use old data with 10.7 server, you need to run the create permanent entities tool. One host-port per stripe needs to be specified so that the tool can connect to all the stripes.

```
> create-permanent-entities.(bat|sh) -connect-to <hostname[:port]>,...
```

Updates to the server startup script

The `start-tc-server` script does not support tc-config XML files anymore. Instead, it supports several options which let you specify a node's configuration via the command itself.

See the section “Starting and Stopping the Terracotta Server” on page 25 for details.

Migration steps

1. Shut down all Terracotta clients and ensure no critical operations (like backup) are running on the cluster. Note down the hosts the nodes are running on.
2. Use the cluster tool `shutdown` command to shut down the Terracotta cluster.
3. Use the config converter tool to convert tc-config.xml files to config directory format.
4. Copy the config directories generated from the step above to the hosts from the first step.
5. Start the nodes using the startup script with option `-config-dir`, supplying the config directory path.
6. Use the create-permanent-entity-tool if you want to use the old data with the 10.7 server. Ensure that you run this script before connecting any clients.
7. Replace the old client jars with 10.7 jars in the client classpath.
8. Connect the clients back with the cluster.

24 Migrating from 10.7 to a newer 10.7 version

This section applies when migrating from a 10.7 version using dynamic configuration (`config-tool`) to a new version.

Preparation steps

1. Ensure all your nodes are up and running.
2. Run `config-tool diagnostic` to ensure:
    a. that the configuration state is not broken on each node (nodes can accept changes).
    b. that all nodes have the same configuration (last change UUID).
    c. that no node need to be restarted (because of a pending change requiring a restart).

Important:

- Please restart the nodes that need a restart first.
- Please repair any incoherent state: refer to the "Config Tool Troubleshooting Guide." on page 75

Migration steps WITHOUT downtime

An upgrade without downtime is possible if not stated otherwise. There can be some cases sometimes where a node running with a newer version won't be compatible with a node still running with an old version.

1. First start by upgrading the passive nodes
    a. Shutdown passive nodes by using the cluster tool `shutdown-if-passive` command
    b. Swap their kit with the new version
    c. Restart them
    d. Wait until they become PASSIVE
2. Then upgrade the active nodes
    a. Shutdown active nodes by using the cluster tool `shutdown-if-active` command
    b. Existing PASSIVE nodes will become ACTIVE (failover)

   c. Swap the kit of the shutdown nodes with the new version
   d. Restart them
   e. Wait until they become PASSIVE

Migration steps WITH downtime

1. Shutdown nodes by using the cluster tool `shutdown` command
2. Swap their kit with the new version
3. Restart them

25 Restarting a Stripe

Restart behavior is closely related to *failover*, but the difference is that the interruption period is typically much longer. On a restart, every server waits for the last active server to return instead of electing a different server as active. If a server other than the last active is elected as the active on a restart, it could cause data loss. To avoid such a data loss, every restarted server will wait in a *suspended* state until all of its peers are also started up so that the last active can be found and elected as the leader.

Unless a timeout is set, the time the clients will wait for the server to return is indefinite.

Note that a stripe can be both restartable and possess high-availability, if it is configured for restart support but also contains multiple servers. In this case, failover will progress as normal unless the entire stripe is taken offline.

Comparison with failover

The process of a *client* reconnecting to a restarted server is very similar to a newly-promoted *active* server after a fail-over. Both scenarios involve the clients reconnecting to re-send their in-flight transactions. Also, both will progress as normal once all clients have reconnected or the reconnect window closes.

The primary difference is that restart only requires one server, whereas high-availability requires at least two.

26 IPv6 support in Terracotta

This document describes the changes needed in config files, tooling, and client side APIs toward enabling IPv6 connections between clients and Terracotta cluster.

Terracotta supports IPv6 addresses as defined in the RFC 5952. Thus, all of following are acceptable:

- Full IPv6 address, e.g. `2001:db8:a0b:12f0:0:0:0:1`.
- Full IPv6 address enclosed in square brackets, e.g. `[2001:db8:a0b:12f0:0:0:0:1]`.
- Full IPv6 address enclosed in square brackets along with port, e.g. `[2001:db8:a0b:12f0:0:0:0:1]:9410`.
- Shortened IPv6 address, e.g. `2001:db8:a0b:12f0::1`.
- Shortened IPv6 address enclosed in square brackets, e.g. `[2001:db8:a0b:12f0::1]`.
- Shortened IPv6 address enclosed in square brackets along with port, e.g. `[2001:db8:a0b:12f0::1]:9410`.

Note that enclosing an IPv6 address in square brackets is mandatory only if a port is to be specified along with it.

Terracotta Server

Consider the following example:

```
node1:hostname=2001:db8:a0b:12f0::1 (1)
node2:hostname=[2001:db8:a0b:12f0:0:0:0:2] (2)
node3:hostname=terracotta-host (3)
```

(1) Specifies a server host IP `2001:db8:a0b:12f0::1` as `hostname` for the node named `node1`.

(2) Specifies a server host IP `2001:db8:a0b:12f0:0:0:0:2` enclosed in square brackets as `hostname` for the node named `node2`.

(3) Specifies a server DNS host name `terracotta-host` which resolves to an IPv6 address as `hostname` for the node named `node3`.

IPv6 server sockets bind to `::0` by default, which can be overridden using the `bind` attribute in the <server> element.

Command-line tools

Command-line tools which accept server addresses need to provide the IPv6 addresses of Terracotta servers as shown in the examples below:

- This example below shows the execution of the cluster tool `status` command for two IPv6 addresses - one enclosed in square brackets, and the other not enclosed in square brackets (and using the default port).

```
./cluster-tool.sh status -connect-to [2001:db8:a0b:12f0::2]:9510 -connect-to
2001:db8:a0b:12f0::1
+----------------------------------+--------------------+-------------------------+
|             Host:Port            |       Status       |    Member of Cluster
   |
+----------------------------------+--------------------+-------------------------+
|    [2001:db8:a0b:12f0::2]:9510   |       ACTIVE       |        tc-cluster
   |
|       2001:db8:a0b:12f0::1       |      PASSIVE       |        tc-cluster
   |
+----------------------------------+--------------------+-------------------------+
```

Command line tools that don't accept server addresses directly do not need any change to work with IPv6.

Ehcache Client

An Ehcache client can specify the IPv6 addresses of the servers it wants to connect to either through Java APIs or XML configuration, as shown below:

API Example

```
InetSocketAddress firstServer =
  InetSocketAddress.createUnresolved("2001:db8:a0b:12f0::1", 0);
InetSocketAddress secondServer =
  InetSocketAddress.createUnresolved("2001:db8:a0b:12f0:0:0:0:2", 9510);
List<InetSocketAddress> servers = Arrays.asList(firstServer, secondServer);
String cacheManagerIdentifier = "cacheManager-1";
PersistentCacheManager cacheManager = CacheManagerBuilder
    .newCacheManagerBuilder()
    .with(EnterpriseClusteringServiceConfigurationBuilder.enterpriseCluster(servers,

        cacheManagerIdentifier)  // (1)
        .autoCreate())
    .build(true);
```

1	`EnterpriseClusteringServiceConfigurationBuilder.enterpriseCluster(Iterable, String)` lets you create a `CacheManager` by specifying IPv6 addresses of the servers. The first argument is the `Iterable<InetSocketAddress>` of the servers in the cluster, while the second argument is the cache manager identifier.

Like other Ehcache APIs, the above API has a secure variant as well with the signature `EnterpriseClusteringServiceConfigurationBuilder.enterpriseSecureCluster(Iterable, String, Path)`, where the last argument is the path to the client's security root directory.

XML Example

```
<ehcache:config
    xmlns:ehcache="http://www.ehcache.org/v3"
    xmlns:tc="http://www.terracottatech.com/v3/terracotta/ehcache">
    <ehcache:service>
        <tc:cluster>
            <tc:cluster-connection cluster-tier-manager="cacheManager-1">  <!-- 1 -->
                <tc:server host="[2001:db8:a0b:12f0::1]"/>                <!-- 2 -->
                <tc:server host="2001:db8:a0b:12f0:0:0:0:2" port="9510" /> <!-- 3 -->
            </tc:cluster-connection>
        </tc:cluster>
    </ehcache:service>
</ehcache:config>
```

1	Cache manager identifier.
2	Terracotta server IP `[2001:db8:a0b:12f0::1]`. Since the port is not specified, it will default to 9410.
3	Terracotta server IP `2001:db8:a0b:12f0:0:0:0:2` and port `9510`.

TCStore Client

```
InetSocketAddress firstServer =
   InetSocketAddress.createUnresolved("2001:db8:a0b:12f0::1", 0);
InetSocketAddress secondServer =
   InetSocketAddress.createUnresolved("2001:db8:a0b:12f0:0:0:0:2", 9510);
List<InetSocketAddress> servers = Arrays.asList(firstServer, secondServer);
DatasetManager datasetManager = DatasetManager.clustered(servers)  // 1
    .build();
```

1	`DatasetManager.clustered(Iterable)` lets you create a `DatasetManager` by specifying IPv6 addresses of servers.

Like other TCStore APIs, the above API has a secure variant as well with the signature `DatasetManager.clustered(Iterable, Path)`, where the last argument is the path to the client's security root directory.

XML Example

```
<clustered xmlns="http://www.terracottatech.com/v3/store/clustered"> <!-- 1 -->
    <cluster-connection>
        <server host="[2001:db8:a0b:12f0::1]"/>                 <!-- 2 -->
        <server host="2001:db8:a0b:12f0:0:0:0:2" port="9510" />  <!-- 3 -->
    </cluster-connection>
</clustered>
```

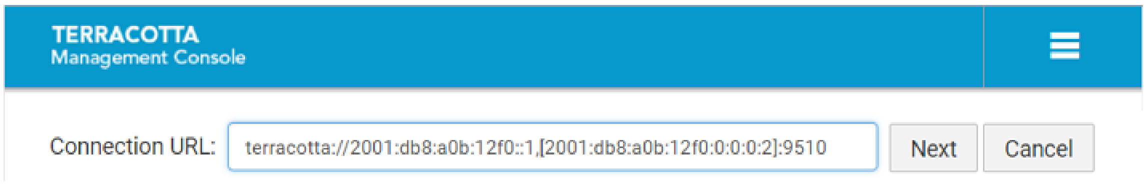
1	Declares a clustered `DatasetManager` configuration.
2	Terracotta server IP `[2001:db8:a0b:12f0::1]`. Since the port is not specified, it will default to `9410`.

3	Terracotta server IP `2001:db8:a0b:12f0:0:0:0:2` and port `9510`.

Terracotta Management Console

To connect to a Terracotta cluster running on IPv6, you can specify the server IPv6 addresses in the **Connection URL** as shown in the snapshot:

Screenshot: Terracotta Management Console, field for Connection URL



Connection URL: terracotta://2001:db8:a0b:12f0::1,[2001:db8:a0b:12f0:0:0:0:2]:9510

27 SSL / TLS Security Configuration in Terracotta

Security Core Concepts

Security features in Terracotta

Terracotta provides the following security features:

1. **Connection encryption** - encrypts client-server and server-server connections using the SSL/TLS protocol.
2. **Cluster authentication** - validates the identity of processes initiating connections to a server in a Terracotta cluster.
3. **IP whitelisting** - restricts access to only allow clients from known IP addresses.
4. **TMS authentication** - validates the identity of users attempting to use the TMS.
5. **TMS authorization** - determines if an authenticated user has access to perform a given operation in TMS.
6. **Auditing** - writes security-relevant events to audit logs.

Security Root Directory

To configure security features in Terracotta, each server and, in most cases, each client, must have a security root directory. The security root directory is a filesystem location for certificates, passwords and other security-related files.

The security root directory, and the files and subdirectories contained in it should be readable by the respective client or server process.

Important:
Ensure that the security root directory is accessible *only* by users who are permitted to run the respective Terracotta client or server.

Structure of the Security Root Directory

The directory structure below lists all possible subdirectories and files that can be present in a security root directory. Note that any one security root directory will have only a subset of these.

```
<security-dir>
├── access-control
│   ├── ldap.properties
│   └── users.xml
├── identity
│   ├── credentials.properties
│   └── <common-name>-<timestamp>.jks
├── trusted-authority
│   └── trusted-authority-<timestamp>.jks
└── whitelist.txt
```

Note:

It is recommended to keep only the required files and directories under the security root directory to prevent configuration errors. If any unidentifiable files or directories are found, an appropriate warning message will be logged.

Access control subdirectory

The `access-control` subdirectory should only be specified on the server-side. It contains files to configure authentication and authorization:

`ldap.properties` - configuration properties for using an LDAP server for authentication or authorization

`users.xml` - used for file-based authentication or authorization

Identity subdirectory

The `identity` subdirectory can be specified on both the client-side and the server-side. It contains certificates or password-based credentials to prove the identity of the process.

`credentials.properties` - a standard Java properties file containing two properties: `username` and `password`. These credentials are sent to the server when the connection is established, in order to authenticate. This file should be present when file-based authentication or LDAP-based authentication is configured.

`<common-name>-<timestamp>.jks` - a keystore containing a certificate for proving identity. Note that the `<common-name>` part of the filename must match the Common Name specified in the certificate and the `<timestamp>` represents the time that the certificate was created.

Trusted authority subdirectory

The `trusted-authority` subdirectory can be specified both client-side and server-side. It contains trusted root certificates.

`trusted-authority-<timestamp>.jks` - a truststore containing a trusted root certificate for validating identity certificates. Note that the <timestamp> represents the time that the certificate was created.

The whitelist.txt file

The `whitelist.txt` file should only be specified on the server-side. It contains details of client IP addresses permitted to establish connections with the cluster.

Certificates

This section assumes a good understanding of SSL/TLS fundamentals.

Certificate creation

Note:
Certificates must be created using the RSA algorithm, preferably with a key size of 4096.

The keystores and truststores, that are deployed to the `identity` and `trusted-authority` directories respectively, can be created in any way desirable as long as the following rules are followed:

Keystore creation rules

- Keystore must be of type `jks`.
- Keystore filename must be in `${common name}-${yyyyMMddThhmmss}.jks` format (e.g. `com.organization.host-20180223T102319.jks`). `yyyyMMddThhmmss` should represent the time of creation (timestamp) of the file. When multiple keystores are present, the keystore with the latest timestamp is used.
- Keystore must have only one `terracotta_security_alias` entry, and it should contain the identity certificate and the corresponding private key.
- Common Name field in the Distinguished Name in the certificate must match the common name fragment in the keystore filename. For a server the common name must match the host name.
- The identity certificate must be within its period of validity.
- The password for the keystore and the `terracotta_security_alias` store entry must be `terracotta_security_password`.
- The certificate must be created using the RSA algorithm, preferably with a key size of 4096.

Truststore creation rules

- Truststore must be of type "jks".
- Truststore filename must be in `trusted-authority-${yyyyMMddThhmmss}.jks` format (e.g. `trusted-authority-20180223T102319.jks`). When multiple truststores are present, all of them are used irrespective of their timestamps.
- Truststore must have only one `terracotta_security_alias` entry, and it should contain a trusted certificate.
- The trusted certificate must be within its period of validity.
- The password for the truststore must be `terracotta_security_password`.
- The certificate must be created using the RSA algorithm, preferably with a key size of 4096.

Certificate rotation

If the certificates expire or get compromised, they must be rotated. Following are the different ways of rotating them:

Certificate rotation with cluster shutdown

The followed steps need to be performed *in order*:

1. Generate new keystore and truststore files following the rules mentioned in the *Keystore creation* and *Truststore creation* sections above.
2. Shut down all the servers and clients.
3. Replace the old keystores and truststores with the new keystores and truststores in the corresponding `identity` and `trusted-authority` directories of each client and server.
4. Start all the servers and the clients for the new certificates to take effect.

The above sequence has the advantage that it is simple to perform, with the drawback of requiring the entire cluster and the clients to be restarted.

Certificate rotation with rolling restarts

The followed steps need to be performed *in order*:

1. Generate new keystore and truststore files following the rules mentioned in the *Keystore creation* and *Truststore creation* sections above.
2. Deploy new truststores in corresponding `trusted-authority` directories of each client and server.
3. Restart all the passive servers and clients. Once the passive servers reach `PASSIVE-STANDBY` status, restart all the active servers.
4. Replace old keystores with the new keystores in corresponding `identity` directories of each client and server.
5. Restart all the passive servers and clients. Once the passive servers reach `PASSIVE-STANDBY` status, restart all the active servers.
6. Delete old truststores from corresponding `trusted-authority` directories of each client and server.
7. Restart all the passive servers and clients. Once the passive servers reach `PASSIVE-STANDBY` status, restart all the active servers.

The above sequence has the advantage that it does not require cluster downtime, with the drawback of having a significant number of steps.

Auditing

The Terracotta server can audit when security-relevant events occur. You can configure this by specifying an audit directory.

Important:
Ensure that the audit directory is accessible only by users who are permitted to run the respective Terracotta server or by users who are allowed to read the audit log.

Audit directory structure

The auditing process creates a hierarchical filesystem structure under the audit directory:

```
<audit-directory>
├── cluster-audit-logs
│   └── <server-name>
│       └── <yyyy-MM-dd>
│           ├── terracotta-server-audit-<yyyy-MM-dd>.0.log
│           └── terracotta-server-audit-<yyyy-MM-dd>.1.log
└── tmc-audit-logs
    └── <TMC spring app name>
        └── <yyyy-MM-dd>
            ├── tmc-audit-<yyyy-MM-dd>.0.log
            └── tmc-audit-<yyyy-MM-dd>.1.log
```

When an auditable event occurs, an appropriate message is written to the audit log file. Each line in the audit log corresponds to a single event.

Audit log files generated by a server in a Terracotta cluster are stored under the `cluster-audit-logs` directory. Audit log files generated by the TMS are stored under the `tmc-audit-logs` directory.

Each server has a separate directory named after the server name. This allows multiple servers to use the same base audit directory.

Under the server-specific directory, logs are organised by date. If any audit log file exceeds 10 MB, a new audit log file is created.

Audit entry sanitization

Audit entries are sanitized to prevent attackers rendering the audit logs illegible. The sanitization process follows the rules :

- All ASCII control characters, and extended ASCII characters are sanitized.
- All non-ASCII characters will be sanitized.
- Tilde (`~`), asterisk (`*`), pipe (`|`), plus (`+`) and backtick (`) characters are sanitized.
- Any number of occurrences of the disallowed characters, and more than one consecutive occurrence of space (' ') is sanitized.
- Input larger than 2000 characters is truncated, and a '*' is added at the end to indicate the truncation.

If the sanitization process finds no unacceptable characters or character sequences, then the information is logged surrounded with backticks. For example, if the user 'alex' successfully authenticated, that is audited as:

```
LDAP authentication success:- IP: `203.0.113.1`, User: `alex`
```

However, if the sanitization process finds unacceptable characters or character sequences, the information is logged surrounded with tildes and prefixed with `sanitized`. Characters replaced in the sanitization process appear in the format `|U+xxxx|` where the `xxxx` is the unicode codepoint of the sanitized character in hexadecimal. For example, if the user `jürgen` successfully authenticated, that is audited as:

```
LDAP authentication success:- IP: `203.0.113.1`, User: ~sanitized j|U+00fc|rgen~
```

Note that there may be more than four digits in the hex codepoint due to supplementary unicode characters.

Cluster Security

Connection encryption

You can enable connection encryption using the `ssl-tls` property. When `<ssl-tls>` is specified, you must also supply:

- `security-dir` with the path to the security root directory.
- `authc` with the authentication scheme to use. See " Authentication" on page 161 for more information.

With SSL/TLS configured, the specified security root directory must contain a valid truststore in the `trusted-authority` subdirectory and a valid keystore in the `identity` subdirectory. The identity certificate is required even if file-based or LDAP-based authentication is chosen. The choice of authentication scheme may require presence of additional files in the security root directory.

On the client, the security root directory must contain a trusted authority certificate in the `trusted-authority` subdirectory. Again, the choice of authentication scheme may mean other files, including an identity certificate, are also required in the security root directory.

Authentication

To configure authentication in the cluster, use `authc` with the appropriate authentication scheme. Supported authentication schemes are `file`, `ldap`, and `certificate`. When authentication is configured, a `security-dir`, with the path to the security root directory, must also be specified.

Important:
It is highly recommended that if you configure an authentication scheme, you also configure encrypted connections using the `ssl-tls` property, otherwise an attacker could acquire credentials by eavesdropping on the unencrypted connection.

Certificate-based authentication

To configure certificate-based authentication, use the `certificate` authentication scheme in the `authc` property, and set `ssl-tls` to `true`. Also, clients must have an appropriate identity certificate keystore in their security root directory.

File-based authentication

To configure file-based authentication, use the `file` authentication scheme in the `authc` property.

The server's security root directory must contain a `users.xml` file, which is a list of all valid users with a password hash for each user. The only password hashing algorithm currently supported is bcrypt.

Example of a `users.xml` file for authentication:

```
<users>
  <user>
    <username>alex</username>
    <password>
      <algorithm>bcrypt</algorithm>
      <hash>$2a$10$UoM85/5I4SnIbrOQuFZ43ekffuQKSxZmL93bR9VMcdr2URmPyjyX2</hash>
    </password>
  </user>
  <user>
    <username>beth</username>
    <password>
      <algorithm>bcrypt</algorithm>
      <hash>$2a$10$6D6c79lE0k/0SxrEtnfhGe2Yr.ygG0rFP1QzeyD9qshIMRrpUMOAS</hash>
    </password>
  </user>
</users>
```

Note:
Hashes should start with bcrypt version `$2a$`.

Generating bcrypt password hashes

The command-line utility for generating bcrypt password hashes is located in `tools/security/bin` under the product installation directory as `bcrypt.bat` for Windows platforms, and as `bcrypt.sh` for Unix/Linux.

When running the bcrypt script, you must specify the number of rounds. The number depends on the performance of the server hardware and how you decide to trade off security with speed. A higher number creates hashes that are harder for attackers to crack, but are also harder for your servers to verify. Increasing the number of rounds by 1 doubles the difficulty. You may find that a value between 10 and 13 is suitable.

The bcrypt script can read the input from the console:

```
bcrypt.sh -n 10
```

or directly from the command line:

```
bcrypt.sh -n 10 pa$$w0rd
```

The console flavor of the command should be preferred to prevent the shell from saving the input password in its history.

LDAP-based authentication

To configure LDAP-based authentication, use the `ldap` authentication scheme in the `authc` property.

The server's security root directory must contain an `ldap.properties` file. The properties in the `ldap.properties` file are the same properties that are used to configure LDAP integration in other Software AG products. See “ LDAP Properties” on page 172 for a full list of supported properties.

Example of an `ldap.properties` file for authentication:

```
url=ldap://ldapserver.example.com:389
userrootdn=ou=People,dc=example,dc=com
uidprop=uid
personobjclass=person
```

Auditing

To configure security event auditing, use the `audit-log-dir` property along with `security-dir` and at least one form of security (i.e. one of `ssl-tls`, `authc`, or `whitelist`). The directory specified in `audit-log-dir` must exist already, and must be appropriately access controlled to prevent illegitimate access to audit logs.

IP Whitelisting

Introduction

The IP whitelisting feature enables you as the cluster administrator to ensure that only clients from known IP addresses can access the TSA. You can use this feature to prevent malicious clients from establishing connections to the TSA.

Note:
It should be understood that usage of this feature in itself does not provide a strong level of security for the TSA. The ideal way to enforce connection restrictions based on IP addresses would be to use host-level firewalls.

The whitelist file

A whitelist file is a plain-text file containing a list of IPs. Only clients configured with these IPs are allowed to access the TSA. The server IPs specified in the config file, and the localhost IPs of the server are always whitelisted. The whitelist file must be named `whitelist.txt` and placed in the security root directory.

Note:
An empty whitelist file has the semantics of blacklisting all IPs, except the ones fetched from the config file, and those corresponding to `localhost`.

The whitelist file follows these parsing rules:

1. The entries can be IP addresses, or CIDR notations (to represent IP ranges). Any entry that is not a valid IP address or a valid CIDR is ignored.
2. Each line in the file can contain either a single IP address, or a comma-separated list of IP addresses.
3. Lines beginning with `#` are considered as comments, and are ignored during parsing.
4. Blank lines are ignored.

The following is an example of a valid whitelist file:

```
# Caching clients
192.168.5.28, 192.168.5.29, 192.168.5.30
10.60.98.0/28
# Other clients
192.168.10.0/24
```

Usage

To configure IP whitelisting, use `whitelist` along with the `security-dir` property.

If the `whitelist.txt` file is not found in the security root directory, or there is an error reading the file, the server startup will fail with an appropriate error message.

If hostnames are used in the config file, the server attempts to resolve these hostnames to IPs. If the resolution fails, the server startup fails with an appropriate error message. Note that hostname resolution is done for the config file only, and any hostnames present inside the `whitelist.txt` file are ignored.

A multi-stripe cluster should be started with the same `whitelist.txt` file contents. Any updates to this file should be performed on all the stripes, as described in the following section.

Dynamic updates

After a cluster is started with whitelisting enabled, entries can be dynamically added to or removed from the whitelist without the need for server restarts. To perform a dynamic update, edit the `whitelist.txt` file contained in the server security root directories, and run the `ipwhitelist-reload` command to notify the servers in the cluster to reload the `whitelist.txt` file. Refer to the “The "ipwhitelist-reload" Command” on page 111 section for more details.

Errors during whitelist reload, if any, are logged in respective server logs. Thus, after every update operation, server logs should be checked to verify that the updates took effect on all the servers.

If a cluster is not yet activated and the whitelist file needs to be reloaded on the servers, the server-level `ipwhitelist-reload` command can be used. It may also be helpful when the machine from where the cluster tool is to be used is itself not whitelisted initially. In this scenario, adding this machine's IP to the whitelist, and running the server-level `ipwhitelist-reload` command ensures that cluster tool can configure the cluster later.

Note:
If any failures happen while reading the `whitelist.txt` file during a dynamic update, the update is ignored and the server continues with the current whitelist. No partial updates are applied.

Connection Behaviour

When a client connects to a server, the server accepts the socket connection, and verifies the IP of the incoming client connection against the whitelist. If it finds that the client IP is not whitelisted, it closes the socket connection.

If a whitelisted client is removed from the whitelist via a dynamic update, it remains connected to the cluster as long as there is no network disconnection or explicit connection closure from the client. Subsequent connection attempts from the client to cluster will fail.

Full example of secure server configuration

This example illustrates the security configuration section in the configuration file of a single stripe, two node cluster (with node names **node1** and **node2**), that configures auditing, encrypted connections, LDAP authentication and IP whitelisting:

```
authc=ldap
ssl-tls=true
whitelist=true
node1:audit-log-dir=/path/to/audit-directory-1
node1:security-dir=/path/to/security-dir-1
node2:audit-log-dir=/path/to/audit-directory-2
node2:security-dir=/path/to/security-dir-2
```

With the configuration in the above example, the server's security root directory would contain a truststore in the `trusted-authority` subdirectory, a keystore and a `credentials.properties` in the `identity` directory, an `ldap.properties` in the `access-control` subdirectory and a `whitelist.txt` file.

The `credentials.properties` file is required so that the server can connect to other servers in the stripe.

Client configuration

To configure a client to connect to a secured cluster, you need to give the client a path to the client's security root directory. This should contain, for example, the credentials that the client needs to connect to the cluster.

Command line tools

To enable command-line tools to connect to a secure cluster, a command must be prefixed with `-security-dir`.

The following example shows the use of the config tool `get` command with the `-security-dir` option specified:

```
> config-tool.sh -security-dir /path/to/security-dir get -connect-to localhost -setting
 data-dirs
node:node1:data-dirs=main:%H/terracotta/user-data/main
```

Attempting to connect to a secure cluster without the `-security-dir` option will fail. Commands without this option retain their behavior.

Ehcache Client

An Ehcache client can define either an XML or a programmatic configuration, both of which support security configuration. The following are the examples of usages of each:

1. API Example

```
PersistentCacheManager cacheManager = CacheManagerBuilder
    .newCacheManagerBuilder()
    .with(EnterpriseClusteringServiceConfigurationBuilder
        .enterpriseSecureCluster(connectionURI,
            securityRootDirectoryPath)   // 1
        .autoCreate())
    .build(true);
```

1	`EnterpriseClusteringServiceConfigurationBuilder enterpriseSecureCluster(URI, Path)` lets you create a `CacheManager` using a secure connection. The first argument is the `URI` of the Terracotta cluster, appended with the `CacheManager` name. The second argument is the `Path` to the client's security root directory. `EnterpriseClusteringServiceConfigurationBuilder enterpriseSecureCluster(Iterable<InetSocketAddress>, String, Path)` serves the same function, with the added support for IPv6 addresses.

Note:
If the `URI` or `Iterable<InetSocketAddress>` contains host names, make sure that they match the host names specified in the server certificates.

2. XML Example

```
<ehcache:config
    xmlns:ehcache="http://www.ehcache.org/v3"
    xmlns:tc="http://www.terracottatech.com/v3/terracotta/ehcache">
  <ehcache:service>
    <tc:cluster>
      <tc:connection url="${cluster-uri}/CM"
security-root-directory="${security-dir}"/>  //1
      <tc:server-side-config auto-create="true"/>
    </tc:cluster>
  </ehcache:service>
</ehcache:config>
```

1	`security-root-directory` lets you specify the path to the client's security root directory. Not passing this option retains the behavior of communicating with an unsecured cluster.

TC Store Client

1. API Example

```
DatasetManager datasetManager = DatasetManager.secureClustered(
    connectionURI, securityRootDirectoryPath)  // 1
    .build();
```

1	`DatasetManager.secureClustered(URI, Path)` lets you create a `DatasetManager` using a secure connection. The first argument is the `URI` of the Terracotta cluster. The second argument is the `Path` to the security root directory which is to be used for the connection. `DatasetManager.secureClustered(Iterable<InetSocketAddress>, Path)` serves the same function, with the added support for IPv6 addresses.

Note:
If the `URI` or `Iterable<InetSocketAddress>` contains host names, make sure that they match the host names specified in the server certificates.

2. XML Example

```
<clustered xmlns=
   "http://www.terracottatech.com/v3/store/clustered"> <!--1-->
  <cluster-connection>
    <server host="localhost" port="9410"/>
    <security-root-directory>/path/to/security-dir
    </security-root-directory>                        <!--2-->
  </cluster-connection>
</clustered>
```

1	Declares a clustered `DatasetManager` configuration.
2	`security-root-directory` lets you specify the path to the client's security root directory. Not specifying this element retains the behavior of communicating with an unsecured cluster.

TMS Security

TMS security is configured using the `tmc.properties` file.

Setting the security root directory

If you need to set the security root directory to use for connections between a browser and the TMS, then you should set the `tms.security.root.directory` property to the path for the security root directory.

Windows platforms

Note that Windows paths often contain backslashes. The Java properties format requires backslashes to be escaped, so `C:\tcdb\security-root-directory` would be configured using:

```
tms.security.root.directory=C:\\tcdb\\security-root-directory
```

Alternatively, you can use forward slashes:

```
tms.security.root.directory=C:/tcdb/security-root-directory
```

Encrypted connections between the browser and the TMS

You can enable HTTPS by setting:

```
tms.security.https.enabled=true
```

When HTTPS is enabled, you must also set the `tms.security.root.directory` property to the path for the security root directory. See "Setting the security root directory" on page 167.

When HTTPS is configured, the specified security root directory must contain a valid truststore in the `trusted-authority` subdirectory and a valid keystore in the `identity` subdirectory.

When HTTPS is configured, the TMS is only accessible over HTTPS. Thus, any URL used to access the TMS must start with `https://` .

Browser warnings

If you are using self-signed certificates, you may see a security warning in your browser when connecting to the TMS. This is because the certificate was not signed by a certificate authority registered with the browser as a trusted certificate authority. We suggest that you check the certificate and verify its authenticity.

Furthermore, you should add the certificate to your browser's list of trusted root certificate authorities.

Note:
Each cookie entry is associated with a certain domain (not including port), and some browsers may remember the protocol under which the cookie has been set. So if you switch between http and https, or between localhost and 127.0.0.1, you should clear related cookies before reloading TMC in the browser.

Authentication

To configure TMS to require users to log in, you should set the `tms.security.authentication.scheme` property. There are two options: `file` and `ldap`, for file-based authentication and LDAP-based authentication respectively.

When authentication is configured, you must also set the following properties:

- `tms.security.root.directory` - see "Setting the security root directory" on page 167.
- `tms.security.authorization.scheme` - see "Authorization" on page 168.

File-based authentication requires a `users.xml` file to be added to the security root directory; LDAP-based authentication requires an `ldap.properties` file. See "Authentication" on page 161 for more details.

Authorization

To control which access to TMS operations, you should set the `tms.security.authorization.scheme` property. There are three options: `authenticated`, `file` and `ldap`. The `authenticated` option allows access to all operations to all authenticated users. The `file` and `ldap` options correspond to file-based authorization and LDAP-based authorization respectively.

When authorization is configured, you must also set the following properties:

- `tms.security.root.directory` - see "Setting the security root directory" on page 167.
- `tms.security.authentication.scheme` - see "Authentication" on page 168.

TMS supports one role: `admin`. Users with no role can use most of the TMS functionality, but some operations, such as shutting down a server, require the `admin` role.

File-based authorization

File-based authorization requires a `users.xml` file to be added to the security root directory. See "Authentication" on page 161 for more details.

For file-based authorization, the `users.xml` file must contain role information.

Example of a `users.xml` file for authorization:

```
<users>
  <user>
    <username>alex</username>
    <roles/>
  </user>
  <user>
    <username>beth</username>
    <roles>
      <role>admin</role>
    </roles>
  </user>
</users>
```

In this example `users.xml` file, `beth` has the `admin` role whereas `alex` does not.

Note that you can use a mixture of authentication and authorization schemes. For example, you could use LDAP-based authentication with file-based authorization. As in the example above, the `users.xml` file need not contain password hash information, if it is not being used for authentication. However, if you use specify `file` for both the authentication and authorization schemes, then you must specify both password hashes and role information.

Example of a `users.xml` file for both authentication and authorization:

```
<users>
  <user>
    <username>alex</username>
    <password>
      <algorithm>bcrypt</algorithm>
      <hash>$2a$10$UoM85/5I4SnIbrOQuFZ43ekffuQKSxZmL93bR9VMcdr2URmPyjyX2</hash>
    </password>
    <roles/>
  </user>
  <user>
    <username>beth</username>
    <password>
      <algorithm>bcrypt</algorithm>
      <hash>$2a$10$6D6c79lE0k/0SxrEtnfhGe2Yr.ygG0rFP1QzeyD9qshIMRrpUMOAS</hash>
    </password>
    <roles>
      <role>admin</role>
    </roles>
  </user>
</users>
```

LDAP-based authorization

LDAP-based authorization requires an `ldap.properties` file. See "Authentication" on page 161 for more details.

The properties in the `ldap.properties` file are the same properties that are used to configure LDAP integration in other Software AG products. See "LDAP Properties" on page 172 for a full list of supported properties.

Example of an `ldap.properties` file for authorization:

```
url=ldap://ldapserver.example.com:389
userrootdn=ou=People,dc=example,dc=com
uidprop=uid
personobjclass=person
memberinfoingroups=true
mattr=uniqueMember
grouprootdn=ou=Group,dc=example,dc=com
gidprop=gid
groupobjclass=group
```

You may wish to map the role defined in the LDAP server to the TMS `admin` role, in which case, you can add the `tcdb.roleMap` property to the `ldap.properties` file. For example:

```
tcdb.roleMap=terracottaTmsAdmin=admin
```

would map the `terracottaTmsAdmin` group defined in the LDAP server onto the TMS `admin` role.

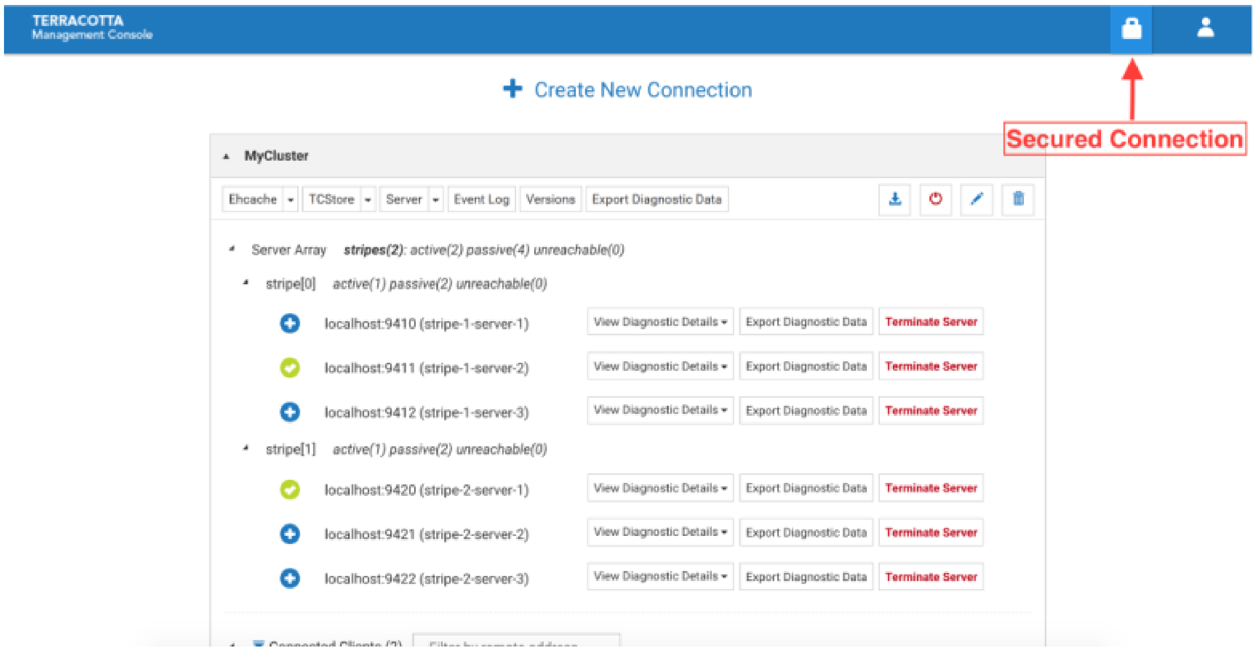
Auditing

To configure auditing for the TMS, set the `tms.security.audit.directory` property to the audit directory.

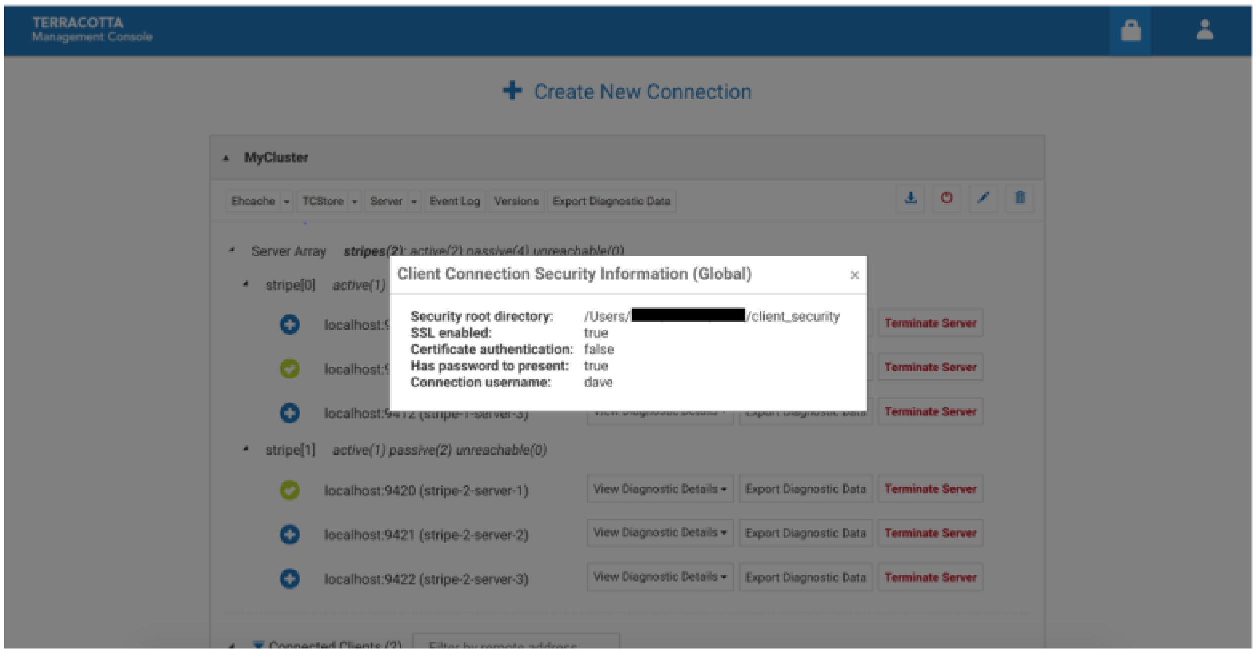
Connecting the TMS to secured clusters

To allow the TMS to connect to a secure cluster, you should set the `tms.security.root.directory.connection.default` property to the path for the security root directory containing, for example, credentials to connect to the secured cluster.

After configuring a secured cluster and setting up the TMS client security root directory, you should see a padlock icon in TMC:


TERRACOTTA
Management Console
Create New Connection
Secured Connection
MyCluster
Ehcache
TCStore
Server
Event Log
Versions
Export Diagnostic Data
Server Array stripes(2): active(2) passive(4) unreachable(0)
stripe[0] active(1) passive(2) unreachable(0)
localhost:9410 (stripe-1-server-1)
localhost:9411 (stripe-1-server-2)
localhost:9412 (stripe-1-server-3)
stripe[1] active(1) passive(2) unreachable(0)
localhost:9420 (stripe-2-server-1)
localhost:9421 (stripe-2-server-2)
localhost:9422 (stripe-2-server-3)
View Diagnostic Details
Export Diagnostic Data
Terminate Server

When you click on the padlock, a pop-up window shows some security related information:



Client Connection Security Information (Global)

Security root directory:	/Users/[redacted]/client_security
SSL enabled:	true
Certificate authentication:	false
Has password to present:	true
Connection username:	dave

Note:
This security root directory and its configuration relates to the connection between the TMS server and the secured cluster. It is independent of the connection security between browser and TMS server.

`SSL enabled`	Connection is SSL enabled - i.e. the security root directory has a trusted authority certificate.

`Certificate authentication`	Connection is 2-way SSL enabled - i.e. the security root directory has an identity certificate.
`Has password to present`	The TMS/TMC client has a `credentials.properties` file.

Windows platforms

Note that Windows paths often contain backslashes. The Java properties format requires backslashes to be escaped, so `C:\tcdb\audit-directory` would be configured using:

```
tms.security.audit.directory=C:\\tcdb\\audit-directory
```

Alternatively, you can use forward slashes:

```
tms.security.audit.directory=C:/tcdb/audit-directory
```

LDAP Properties

The minimum set of properties to specify is:

```
url
userrootdn
uidprop
personobjclass
```

If you are using LDAP for authorization, you must also specify:

```
memberinfoingroups
mattr
```

and, if you set `memberinfoingroups` to `true`, you must also specify:

```
grouprootdn
gidprop
groupobjclass
```

Most other properties need only be used if you have specific requirements.

Connection related properties

`url` - The URL of the LDAP server (e.g `ldap://ldapserver:389` or `ldaps://ldapserver:636`).

`keystoreUrl` - The URL from which a keystore can be retrieved (e.g. `file:///usr/local/ldap/keystore.jks`) - used to authenticate to the LDAP server.

`keystoreType` - The store type of the keystore (e.g. `JKS`).

`keystorePassword` - The password to verify the integrity of the keystore.

`keyAlias` - The alias in the keystore where the certificate and key are stored.

`keyPassword` - The password to allow access to the specified alias. Defaults to store password.

`truststoreUrl` - The URL from which a truststore can be retrieved. Used to to validate the certificate presented by the LDAP server during an SSL/TLS handshake.

`truststoreType` - The store type of the truststore (e.g. `JKS`).

`truststorePassword` - The password to verify the integrity of the truststore.

`noPrinIsAnonymous` - Set to true for LDAP servers that allow anonymous connections.

`prin` - The username to use to authenticate to the LDAP server.

`cred` - The password to use to authenticate to the LDAP server.

Note:
If `prin` and `cred` are not specified and `noPrinIsAnonymous` is not set to `true`, then the username and password of the user attempting to authenticate to the cluster / TMS will be used to authenticate to the LDAP server.

`watt.server.ldap.ignore.serverCertificateValidity` - If `true`, then invalid certificates presented by the LDAP server are ignored.

`watt.server.ldap.extendedProps` - Extra properties to add to the LDAP context. Format: `key1=value1;key2=value2`

`watt.server.ldap.retryCount` - How many times to retry a connection if it fails.

`watt.server.ldap.retryWait` - How many milliseconds to wait between connection retries.

Properties related to how to interact with the LDAP server

`timeout` - LDAP query timeout in milliseconds.

`watt.server.ldap.DNescapeChars` - A list of characters that should be escaped.

`watt.server.ldap.DNescapePairs` - A list of characters that should not be re-escaped.

`watt.server.ldap.DNstripQuotes` - If `false`, then quotes that get added when escaping are not striped from DNs.

`watt.server.ldap.DNescapeURL` - If true, then the start of a DN is escaped. This is useful for referrals when DNs can start with a URL.

`watt.server.jndi.searchresult.maxlimit` - The maximum number of results to return from an LDAP search. Zero means unlimited.

Properties related to the schema of a user

`userrootdn` - The DN under which users can be found (e.g. ou=People,dc=example,dc=com).

`uidprop` - The attribute on a user which contains the primary ID of the user (e.g. uid).

`personobjclass` - The LDAP schema class for users (e.g. person).

`useaf` - If true, then the `dnprefix` and `dnsuffix` properties should be used.

`dnprefix` - A string added to the beginning of a username for the LDAP lookup.

`dnsuffix` - A string added to the end of a username for the LDAP lookup.

Properties related to the schema of a group

`grouprootdn` - The DN under which groups can be found (e.g. ou=Group,dc=example,dc=com).

`gidprop` - The attribute on a group which contains the primary ID of the group (e.g. gid).

`groupobjclass` - The LDAP schema class for groups (e.g. group).

Properties related to how the schema connects users and groups

`group` - A role automatically given to every user.

`memberinfoingroups` - If true, then group membership is in the group definitions under the `grouprootdn`. If false, then group membership is in the user definitions under the `userrootdn`.

`mattr` - The attribute on a user that specifies a group to which the user belongs OR the attribute on a group that specifies a user is a member. The semantics depends on the choice of `memberinfoingroups`.

`recursiveSearchDepth` - How deep to search for groups that are members of other groups.

Properties that Terracotta supports in addition to other Software AG products

`tcdb.roleMap` - A mapping from group names on the LDAP server to roles used in Terracotta. Format: `group1=tcdbRole1;group2=tcdbRole2`

Note:
Multiple LDAP groups can map to the same role.

SSL / TLS Troubleshooting guide

This document provides a list of the most commonly seen problems related to "Cluster Security" on page 161, and their solutions:

Problem category: Host fails to start

This section describes the most commonly seen problems related to a host (server or a client) startup.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
configured security-dir /path/to/security-dir does not exist
```

Diagnosis

The specified security root directory does not exist.

Action

Make sure that the directory exists and contains `identity` and `trusted-authority` directories with valid keystores and truststores in them.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
identity directory doesn't exist in configured
security-dir /path/to/security-dir
```

Diagnosis

The specified security root directory does not contain an `identity` directory.

Action

Make sure that the directory exists inside the security root directory and contains valid keystores.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
trusted-authority directory doesn't exist in configured
security-dir /path/to/security-dir
```

Diagnosis

The specified security root directory does not contain a `trusted-authority` directory.

Action

Make sure that the directory exists inside the security root directory and contains valid truststores.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
No acceptable keystore files found in identity directory
/path/to/security-dir/identity
```

Diagnosis

Either of:

- `identity` directory does not contain any keystores.
- `identity` directory contains keystores, but their file names are not in the format `${common name}-${yyyyMMddThhmmss}.jks` (e.g. `com.organization.host-20180223T102319.jks`).

Action

Make sure that `identity` directory contains keystores which follow the keystore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
No acceptable truststore files found in trusted-authority directory
/path/to/security-root-directory/trusted-authority
```

Diagnosis

Either of:

- `trusted-authority` directory does not contain any truststores.
- `trusted-authority` directory contains truststores, but their file names are not in the format `${common name}-${yyyyMMddThhmmss}.jks` (e.g. `trusted-authority-20180223T102319.jks`).

Action

Make sure that `trusted-authority` directory contains truststores which follow the truststore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
Tried to use the password terracotta_security_password to load the
keystore file
/path/to/security-dir/identity/com.organization.host-20180131T120830.jks
but that failed
```

Diagnosis

Latest keystore file does not have `terracotta_security_password` as its password, where *latest keystore file* is the keystore file with the latest timestamp string in the filename (e.g., `host-20180131T120830.jks` is considered newer than both `host-20170131T120830.jks` and `host-20180131T120822.jks`).

Action

Make sure the keystores follow the keystore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
```

```
Tried to use the password terracotta_security_password to read the
keystore entry with alias terracotta_security_alias in the keystore
file
/path/to/security-dir/identity/com.organization.host-20180131T120830.jks
but that failed
```

Diagnosis

Latest keystore file does not have `terracotta_security_password` as `terracotta_security_alias` entry password, where *latest keystore file* is the keystore file with the latest timestamp string in the filename (e.g., `host-20180131T120830.jks` is considered newer than both `host-20170131T120830.jks` and `host-20180131T120822.jks`).

Action

Make sure the keystores follow the follow the keystore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
Unable to find required private key/certificate chain entry using
alias terracotta_security_alias in keystore file
/path/to/security-dir/identity/com.organization.host-20180131T120830.jks
```

Diagnosis

Latest keystore file does not have `terracotta_security_alias` as certificate alias, where *latest keystore file* is the keystore file with the latest timestamp string in the filename (e.g., `host-20180131T120830.jks` is considered newer than both `host-20170131T120830.jks` and `host-20180131T120822.jks`).

Action

Make sure the keystores follow the keystore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
Certificate in keystore file
/path/to/security-dir/identity/com.organization.host-20180131T120830.jks
is expired
```

Diagnosis

Latest keystore file contains an expired certificate, where *latest keystore file* is the keystore file with the latest timestamp string in the filename (e.g., `host-20180131T120830.jks` is considered newer than both `host-20170131T120830.jks` and `host-20180131T120822.jks`).

Action

Make sure the keystores follow the keystore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
Certificate in keystore file
/path/to/security-dir/identity/com.organization.host-20180131T120830.jks
is not valid yet
```

Diagnosis

Latest keystore file contains a certificate with a future start date, where *latest keystore file* is the keystore file with the latest timestamp string in the filename (e.g., `host-20180131T120830.jks` is considered newer than both `host-20170131T120830.jks` and `host-20180131T120822.jks`).

Action

Make sure the keystores follow the keystore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
The common name org.host of the certificate that was loaded from
keystore file
/path/to/security-dir/identity/com.organization.host-20180131T120830.jks
doesn't match the common name com.organization.host in the filename
com.organization.host-20180131T120830.jks
```

Diagnosis

Common Name field in the Distinguished Name in the host's certificate does not match the common name fragment in the latest keystore filename.

Action

Make sure the keystores follow the keystore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
No valid trusted certificates found in trusted-authority directory
/path/to/security-dir/trusted-authority; Unable to find
required trusted certificate entry using alias
terracotta_security_alias in truststore file
/path/to/security-dir/trusted-authority/trusted-authority-20180131T120832.jks
```

Diagnosis

No valid truststores were found. The specific truststore reported in the Exception message contains a certificate which uses an alias other than `terracotta_security_alias`. Note that this Exception can be followed by one or more Suppressed Exceptions that can indicate why other truststores could not be used.

Action

Make sure the truststores follow the truststore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
No valid trusted certificates found in trusted-authority directory
/path/to/security-dir/trusted-authority; Certificate in
truststore file
/path/to/security-dir/trusted-authority/trusted-authority-20180131T120834.jks
is expired
```

Diagnosis

No valid truststores were found. The specific truststore reported in the Exception message contains an expired certificate. Note that this Exception can be followed by one or more Suppressed Exceptions that can indicate why other truststores could not be used.

Action

Make sure the truststores follow the truststore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
No valid trusted certificates found in trusted-authority directory
/path/to/security-dir/trusted-authority; Certificate in
truststore file
/path/to/security-dir/trusted-authority/trusted-authority-20180131T120834.jks
is not valid yet
```

Diagnosis

No valid truststores were found. The specific truststore reported in the Exception message contains a certificate with a future start date. Note that this Exception can be followed by one or more Suppressed Exceptions that can indicate why other truststores could not be used.

Action

Make sure the truststores follow the truststore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
No valid trusted certificates found in trusted-authority directory
security-dir\trusted-authority; Tried to use the password
terracotta_security_password to load the truststore file
security-dir\trusted-authority\trusted-authority-20180131T120832.jks
but that failed
```

Diagnosis

No valid truststores were found. The specific truststore reported in the Exception message does not have `terracotta_security_password` as its password. Note that this Exception can be followed by one or more Suppressed Exceptions that can indicate why other truststores could not be used.

Action

Make sure the truststores follow the truststore creation rules as described in the section "Certificates" on page 157.

Symptom

Host fails to start with an Exception message similar to:

```
java.lang.RuntimeException:
com.terracottatech.security.common.exception.SecurityConfigurationException:
Unable to validate certificate chain with alias
terracotta_security_alias in keystore file
/path/to/security-dir/identity/com.organization.host-20180131T120830.jks
using truststore file(s)
```

Diagnosis

Host certificate in latest keystore file is not signed by any of the known trusted authorities, and thus cannot be validated by any of the truststore files. *Latest keystore file* here is the keystore file with the latest timestamp string in the filename (e.g., `host-20180131T120830.jks` is considered newer than both `host-20170131T120830.jks` and `host-20180131T120822.jks`).

Action

Make sure that the latest keystore file contained in the identity directory is signed by the truststores in the `trusted-authority` directory.

Problem category: Connection fails to establish

Symptom

```
org.terracotta.connection.ConnectionException:
com.terracottatech.connection.ProbableSecurityConfigurationException:
Handshake with server failed when this client tried to initiate a
non-secure connection. Possible reason: server is running with
security enabled.
```

Diagnosis

The client which tried to establish a connection to a server running with SSL/TLS configuration is not using an SSL/TLS configuration.

Action

Make sure the client uses a correct SSL/TLS configuration (via a secure API, an XML config, command parameters etc.) so that it can establish a secure connection with an SSL/TLS security-enabled server.

Symptom

```
org.terracotta.connection.ConnectionException:
com.terracottatech.connection.ProbableSecurityConfigurationException:
Handshake with server failed when this client tried to initiate a
secure connection. Possible reasons: client security configuration
is not valid, or server is not running with security
enabled.
```

Diagnosis

Either of:

1. The client is using an SSL/TLS security configuration but the server is not.
2. The client cannot validate the server because the server's CA certificate is not present in the client's trusted certificates.
3. The server cannot validate the client because the client's CA certificate is not present in the server's trusted certificates.

Action

Make sure that:

1. The client uses an unsecured configuration (via an unsecured API, an XML config, command parameters etc.) if the server is running with an unsecured configuration.
2. The client and the server certificates are signed by the same CA and their `trusted-authority` directories contain the same truststores.

28 Terracotta Server Migration from BigMemory to Terracotta

The new generation of Terracotta Server and platform has very significant changes from BigMemory with respect to handling of cluster topology, data storage formats, and various other aspects.

Because of this, there is no possibility of direct migration of data and configuration from a BigMemory installation to a Terracotta installation.

If you intend to reuse the same host systems in Terracotta that you were using in BigMemory, the Terracotta Server configuration file(s) from an existing BigMemory installation may be a useful reference when you create your new Terracotta configuration file(s). The BigMemory configuration file(s) contain the host names, addresses, etc. that you have been using so far.

29 Using Command Central to Manage Terracotta

Software AG Command Central is a tool that release managers, infrastructure engineers, system administrators, and operators can use to perform administrative tasks from a centralized location. It assists with configuration, management, and monitoring tasks in a simple and flexible manner.

Terracotta server instances can be managed from Command Central like other Software AG products. Both the Command Line and Web Interfaces of Command Central are supported.

Supported Commands

Terracotta supports the following Command Central CLI (Command Line Interface) commands:

1. **Inventory**
   - `sagcc list inventory components` : Lists information about run-time components.
   - `sagcc get inventory components` : Retrieves information about a specified run-time component.
2. **Lifecycle**
   - `sagcc exec lifecycle` : Executes a lifecycle action against run-time components. See " Lifecycle Actions for Terracotta" on page 186 for Terracotta-specific information about Lifecycle Actions.
3. **Monitoring**
   - `sagcc get monitoring state` : Retrieves the run-time status and run-time state of a run-time component.
   - `sagcc get monitoring alerts` : Lists the alerts for a specified run-time component.
   - `sagcc get monitoring runtimestatus` : Retrieves the run-time status of a run-time component.
4. **Configuration**
   - `sagcc get configuration data` : Retrieves data for a specified configuration instance that belongs to a specified run-time component.

- `sagcc list configuration types` : Lists information about configuration types for the specified run-time component. See " Supported Configuration Types" on page 186 for Terracotta-specific information about configuration types.
- sagcc list configuration instances : Retrieves information about a specific configuration instance that belongs to a specified run-time component.

5. **Diagnostics**

   - `sagcc list diagnostics logs` : Lists the log files that a specified run-time component supports.
   - `sagcc get diagnostics logs` : Retrieves log entries from a log file. Log information includes the date, time, and description of events that occurred with a specified run-time component.

For information about Command Central CLI commands, see the Command Central Help.

Supported Configuration Types

Terracotta supports creating instances of the following configuration types:

- `JVM-OPTIONS`: The JVM memory settings for the Terracotta Server instance in `JAVA_OPTS` environment variable format.

  Changes to this configuration will be effective upon a server restart.
- `TC-SERVER-NAME`: The name for the Terracotta Server instance. Editing this property is not allowed.

Lifecycle Actions for Terracotta

Terracotta supports the following lifecycle actions with the `sagcc exec lifecycle` CLI command and the Command Central Web Interface:

- `Start`: Start a server instance.
- `Restart`: Restart a running server instance.
- `Stop`: Stop a running server instance.

Runtime Monitoring Statuses for Terracotta

Terracotta can return the following statuses from `sagcc get monitoring runtimestatus` and `sagcc get monitoring state` CLI commands and the Command Central Web Interface:

- **Starting**: The server instance is starting. This is usually shown when:
  - The server was just started.
  - The server is a slave (Passive) synchronizing with its master (Active).
  - The server is recovering from an error condition.

- **Not Ready**: The server is not ready to accept client requests. To make it ready, follow the steps defined in the section Making server ready.
- **Online Master**: The server instance is running and is the master (Active) in its stripe.
- **Online Slave**: The server instance is running and is a slave (Passive) in its stripe.
- **Stopping**: The server instance is stopping.
- **Stopped**: The server instance is not running.
- **Failed**: The server instance was running, but crashed or was killed without the SPM plugin's knowledge. If the server had crashed, checking its logs may help uncover the reason.
- **Unresponsive**: The server instance is running, but is not responding.
- **Unknown**: The state of the server instance is not known. This is most likely because of an unexpected exception or error that occurred while trying to fetch the server status.

Directory structure

The Terracotta server and SPM related files can be found under `${INSTALL_ROOT}/TerracottaDB/server/SPM`. This directory contains the following:

1. `bin`

   Contains scripts to start and shut down the server.

2. `conf`

   Contains the Terracotta config file `cluster.properties`. If any changes to the configuration are required, such as increasing the offheap, this file needs to be updated manually. See the "Updating the config file" on page 188 section for more details.

   Note:
   This is the only directory in which content can be changed.

3. `instance`

   Contains Terracotta SPM instance related metadata files.

4. `server-data`

   Contains data maintained by Terracotta server. The contents of this directory are useful for troubleshooting problems with the server. The table below summarizes the contents of this directory:

Directory	Description
`logs`	Contains Terracotta server logs
`metadata`	Contains Terracotta server metadata
`user-data`	Contains Terracotta client data

Directory	Description
`config`	Contains Terracotta server configuration repository

Updating the config file

A Terracotta configuration file is a Java properties file containing configuration and topology information of the entire Terracotta cluster. Any changes to the config file must be done manually.

To find out more, visit "The TerracottaConfiguration File" on page 45.

Making configuration changes

Configuration changes in the default configuration `cluster.properties` file can be done freely before cluster activation. It's advised to make all the important changes before activating the cluster. After the cluster has been activated, the configuration can still be changed, but that's subject to certain constraints.

To find out more, visit "Performing configuration changes" on page 63.

Making topology changes

Topology changes in the default configuration `cluster.properties` file can be done freely before cluster activation. If the configuration of a new node is added to this file, remember to use the same file across all installations. Also, the server name for the current installation must be updated in the `server-name` file (located next to `cluster.properties`).

The cluster topology can be changed dynamically as well.

To find out more, see "attaching nodes dynamically" on page 61 and "detaching nodes dynamically" on page 62 in the description of the config tool.

Making the server ready

When the Terracotta server is started for the first time from Command Central, it will be in the 'Not Ready' state. To make the server ready to accept client requests, it needs to be part of an activated cluster. The `activate` command of the config tool (located under `${INSTALL_ROOT}/TerracottaDB/tools/bin`) can be used to activate the cluster.

See the config tool section "Activate" on page 59 for more details.

30 Terracotta in Network Environments with Subnets

If Terracotta nodes reside in a given subnet (for example in a Kubernetes cluster) and clients in another (for example outside of the Kubernetes cluster), node addresses (host name or IP address) are not resolvable by clients. Administrators of the Terracotta cluster can use public addresses in this scenario to assign public names to Terracotta cluster nodes, using which clients can establish connections.

Configuring public addresses

A node's *public address* is defined by its `public-hostname` and `public-port` properties. Public addresses must be defined for all nodes in a cluster or none of them. When configured, each node's public address must be unique within the cluster.

Consider a 2x2 cluster (2-stripes, 2 nodes per stripe) with node names: **node1**, **node2**, **node3** and **node4**. Public addresses can be configured using one of the following methods:

Using command-line parameters during node startup

This example illustrates starting each of the four nodes by passing the same public hostname (**tc-cluster.public.com**) to the startup script but specifying a unique public-port for each node:

```
> start-tc-server.sh -name node1 public-hostname=tc-cluster.public.com public-port=1111
 ...
> start-tc-server.sh -name node2 public-hostname=tc-cluster.public.com public-port=2222
 ...
> start-tc-server.sh -name node3 public-hostname=tc-cluster.public.com public-port=3333
 ...
> start-tc-server.sh -name node4 public-hostname=tc-cluster.public.com public-port=4444
 ...
```

Using config file during node startup

Public addresses for an entire cluster can be saved in a Terracotta " configuration file" on page 45, which can later be used to start a cluster.

This example illustrates the configuration file entries required for specifying the same public hostname (**tc-cluster.public.com**) for all nodes but specifying a unique public-port for each node (as in the previous example):

```
public-hostname=tc-cluster.public.com
node1:port=1111
```

```
node2:port=2222
node3:port=3333
node4:port=4444
```

Using config tool "set" command

The config tool “set” on page 65 command can be used to dynamically configure public addresses on a cluster. NOTE: Public addresses can be updated without having to restart the cluster.

The following example illustrates how to set the same public hostname (**tc-cluster.public.com**) for all nodes but setting a unique public-port for each node:

```
> config-tool.sh set -connect-to tc-cluster.internal.com:9410
-setting public-hostname=tc-cluster.public.com
-setting node1:public-port=1111
-setting node2:public-port=2222
-setting node3:public-port=3333
-setting node4:public-port=4444
```

SSL/TLS considerations

The SSL/TLS certificates of the Terracotta nodes will need to include Subject Alternative Names (SANs) that match the public addresses.

Using public addresses

Once the public addresses have been set, all tools (e.g. config tool, cluster tool etc.) would use the public addresses by default.

The following example illustrates the execution of the cluster tool `status` command on the cluster with public addresses configured. Take note of how the internal to public address mapping is displayed in the `Host-Port` column:

```
> cluster-tool.sh status -cluster-name tc-cluster -connect-to tc-cluster.public.com:1111
| STRIPE: stripeA |
+----------------+-------------------------------------------------------------+------------+
|    Node Name    |                  Host-Port                                         |
 Status    |
+----------------+-------------------------------------------------------------+------------+
|     node1       |    tc-cluster.internal.com:9410=tc-cluster.public.com:1111   |
 ACTIVE    |
---------------------------------------------------------------------------------------------
|     node2       |    tc-cluster.internal.com:9420=tc-cluster.public.com:2222   |
 PASSIVE   |
+-------------------------------------------------------------------------------------------+
| STRIPE: stripeB |
+----------------+-------------------------------------------------------------+------------+
|   Node Name     |                  Host-Port                                         |
 Status    |
+----------------+-------------------------------------------------------------+------------+
|     node3       |    tc-cluster.internal.com:9430=tc-cluster.public.com:3333   |
 ACTIVE    |
---------------------------------------------------------------------------------------------
|     node4       |    tc-cluster.internal.com:9440=tc-cluster.public.com:4444   |
 PASSIVE   |
+----------------+-------------------------------------------------------------+------------+
```