Terracotta Server Administration Guide

Version 10.1

October 2017

TERRACOTTA

This document applies to Terracotta DB and Terracotta Ehcache Version 10.1 and to all subsequent releases.

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

Document ID: TC-SRV-AG-101-20190415

Table of Contents

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 can be broken down into one or more logically independent stripes. The total storage capacity of the TSA can be increased with the addition of more stripes.

A stripe can be broken down into one or more servers. Each stripe contains a single *active server* and zero or more *passive servers*. These stripe members are all configured by a single Terracotta configuration file. Refer to the section "The Terracotta Configuration File" on page 53 for more details.

For more information on clients, active servers and passive servers, see the sections "Clients in a Cluster" on page 11 and "Active and Passive Servers" on page 7.

TSA Topologies

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

TSA Topology	Description
Single-server cluster	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.
High-availability cluster	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 will 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 to possess high-availability.
Multi-stripe cluster	Multi-stripe refers to a TSA that consists of multiple independent stripes. This scheme offers the ability for

TSA Topology	Description
	increased storage, 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 13.

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 15.

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 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 9 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 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.

4 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 15.

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

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 such 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), then concrete concepts (such as server up-time), ending in more arbitrary concepts designed to break edge-case ties (such as a randomly generated number).

5 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 5).

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.

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.

6 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.

The usage of script is as follows:

```
start-tc-server.sh [-f /path/to/tc-config.xml] [-n server_name]
```

Options to the script

While it is possible to run the script without any arguments, this will result in using empty defaults for the configuration, which is generally not useful for anything other than verifying that the Terracotta Server is able to run.

Specific arguments which should be used are:

- `[-f /path/to/tc-config.xml]` - This is the path to the `tc-config.xml` file for the stripe this server is expected to join. Note that all servers in the same stripe are expected to use the same configuration file.

  The file `tc-config.xml` describes per-server details such as listening TCP port and log directory.

- `[-n server_name]` - This is the name the server should use for itself, which determines which server stanza described within the `tc-config.xml` should be used.

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 stop all servers in the cluster by using the cluster tool. See the section "The Cluster Tool" on page 31 for details.

7 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 "The Cluster Tool" on page 31 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.

8 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.

These server resources are configured in the `plugins` section of the Terracotta configuration file. For related information, see the section "The Terracotta Configuration File" on page 53.

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 configuration file you define one ore 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.

Example Offheap Resource Configuration

```
<plugins> <!--1-->
  <config>
    <ohr:offheap-resources
      xmlns:ohr="http://www.terracotta.org/config/offheap-resource">
      <ohr:resource name="primary-server-resource"
          unit="MB">384</ohr:resource> <!--2-->
      <ohr:resource name="secondary-server-resource"
          unit="MB">256</ohr:resource> <!--3-->
    </ohr:offheap-resources>
  </config>
</plugins>
```

1	The `plugins` element is a direct child of the `tc-config` element
2	Defines `primary-server-resource` with size 384MB
3	Defines `secondary-server-resource` with size 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 "Platform Persistence" on page 23 below. See also the sections *Fast Restartability* and *Creating a Restartable Cache Manager* in the *Ehcache API Developer Guide*.

Sample Data Directories Configuration

```
<config xmlns:data="http://www.terracottatech.com/config/data-roots">
  <data:data-directories>
    <data:directory name="someData">    <!-- 1 -->
      /mnt1/data                        <!-- 2 -->
    </data:directory>
    <data:directory name="otherData">   <!-- 3 -->
      %(logs.path)/data                 <!-- 4 -->
    </data:directory>
  </data:data-directories>
</config>
```

1	`someData` is a data directory name,
2	`/mnt1/data` is the disk location to which `someData` is mapped to.
3	`otherData` is another data directory name mapped to a different disk location,
4	`%(logs.path)/data` is the data directory which `otherData` is mapped to. Note the use of `%(logs.path)`, which gets substituted with the logs path property

See the description of parameter substitution in the section "The Terracotta Configuration File" on page 53 to check the complete list of available parameter substitutions.

General Notes on Configuring Data Directories

- A data directory specified in a stripe configuration file must be specified in all the configurations of all stripes of 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.

Platform Persistence

The Terracotta server saves its internal state on a disk which enables server restarts without losing data. Platform persistence leverages data directories to store required data, so at least one data directory must be configured in the server configuration.

Note: Platform persistence is mandatory and a Terracotta server will refuse to start if there are no data directories defined.

Care must be taken to avoid losing data when restarting the stripe. Refer to the section "Restarting a Stripe" on page 51 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 7 for more details.

Changing the disk location of the data directory used for platform persistence before restarting a server and not copying the data will result in the server starting as if it was a new server without any data.

Platform Persistence Configuration

By default, if a single data directory is defined, it will be used for platform persistence. If more than one data directory is defined, one of them must have the `use-for-platform` attribute set to `true`.

If the server cannot resolve a data directory to be used for platform persistence, it will fail to start.

Sample Server Configuration with Platform Persistence

```
<tc-config xmlns="http://www.terracotta.org/config"
  xmlns:data="http://www.terracottatech.com/config/data-roots"
  xmlns:persistence="http://www.terracottatech.com/config/platform-persistence">
  <plugins>
    <config>
      <data:data-directories>
        <data:directory name="platform"
            use-for-platform="true">/mnt/platform</data:directory>  <!-- 1 -->
      </data:data-directories>
      <data:data-directories>
        <data:directory name="data">/mnt/data</data:directory>
      </data:data-directories>
    </config>
  </plugins>
  <servers>
    <server host="localhost" name="server1">
      <tsa-port>9410</tsa-port>
    </server>
  </servers>
</tc-config>
```

1	Indicates that the `platform` data directory is to be used for platform persistence

Relation to Fast Restartability

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

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

9 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.

10 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.

11 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.

When TCStore serves data from its client-side cache, the client checks its lease. If it detects that the lease has expired, it will not use potentially stale data held in the cache.

Lease Length

When selecting the length of lease, consider the range of possible client to server roundtrip 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 roundtrip 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 length of a lease.

The default value of leases is currently two and a half minutes.

Lease Configuration

To configure the lease length, add a connection leasing service plugin configuration to the tc-config file. For example:

```
<tc-config xmlns="http://www.terracotta.org/config"
  xmlns:lease="http://www.terracotta.org/service/lease">
  <plugins>
    <service>
      <lease:connection-leasing>
        <lease:lease-length unit="seconds">60</lease:lease-length>
      </lease:connection-leasing>
    </service>
```

```
  </plugins>
</tc-config>
```

which configures a lease length of sixty seconds.

Valid values for the `unit` attribute are: `milliseconds`, `seconds`, `minutes` and `hours`.

Any positive integer may be used for the value within the `lease-length` element as long as the length of time configured for the lease length is not more than Long.MAX_VALUE nanoseconds, which is approximately 290 years.

12 The 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:

- Configure or re-configure a cluster
- Obtain the status of running servers
- Dump the state of running servers
- Stop the running servers
- Take backups of running servers
- Configure IP white-list based security on running servers

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

Usage Flow

The following is a typical flow in a cluster setup and usage:

1. Create Terracotta configuration files for each stripe in the deployment. See the section "The Terracotta Configuration File" on page 53 for details.
2. Start up the servers in each stripe. See the section "Starting and Stopping the Terracotta Server" on page 17 for details.
3. Make sure the stripes are online and ready.
4. Configure the cluster using the `configure` command of the cluster tool. See the section *The "configure" Command* below for details.
5. Check the current status of the cluster or specific servers in the cluster using the `status` command. See the section "The "status" Command" on page 36 below for details.

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 `-h` (long option `--help`). Option `-v` (long option `--verbose`) gives you verbose output, and is useful to debug error conditions.

Each command has the option `-h` (long option `--help`), which can be used to display the usage for the command.

The following sections give a comprehensive explanation of the available commands.

The "configure" Command

The `configure` command creates a cluster from Terracotta configuration files and a license. No functionality is available on the server until a valid license is installed. See the section " Licensing" on page 41 for details.

Note: All servers in any given stripe should be started with the same configuration file. The `configure` command configures the cluster based on the configuration(s) of the currently known active server(s) only. If there is a configuration mismatch among the active and passive server(s) within the same stripe, this command can configure the cluster while taking down any passive server(s) with configuration mismatches. This validation also happens upon server restart and changes will prevent the server from starting. See the section on the *reconfigure* command for more information on how to update server configurations.

The command will fail if any of the following checks do not pass:

1. License checks
   a. The license is valid.
   b. The provided configuration files do not violate the license.
2. Configuration checks
   - The provided configuration files are consistent across all the stripes.

The following items are validated within the `plugins` section of the configuration files:

1. `config`:
   a. `offheap-resources`: Offheap resources present in one configuration file must be present in all the files with the same sizes.
   b. `data-directories`: Data directory identifiers present in one configuration file must be present in all the files. However, the data directories they map to can differ.
2. `service`
   - `security`
     - `white-list`: If this element is present in one configuration file, it must be present in all the files.
   - `backup-restore`: If this element is present in one configuration file, it must be present in all the files.

Refer to the section "The Terracotta Configuration File" on page 53 for more information on these `plugin` elements.

The `servers` section of the configuration files is also validated. Note that it is not validated between stripes but rather against the configuration used to start the servers themselves.

- `server`
  - `host`: It must be a strict match
  - `name`: It must be a strict match
  - `tsa-port`: It must be a strict match

Note: Once a cluster is configured, a similar validation will take place upon server restart. It will cause the server to fail to start if there are differences.

Usage:

```
configure -n CLUSTER-NAME [-l LICENSE-FILE] TC-CONFIG [TC-CONFIG...]
configure -n CLUSTER-NAME [-l LICENSE-FILE] -s HOST[:PORT] [-s HOST[:PORT]]... ]
```

Parameters:

- `-n CLUSTER-NAME`: A name that is to be assigned to the cluster.
- `-l LICENSE-FILE`: The path to the license file. If you omit this option, the cluster tool looks for a license file named `license.xml` in the location `tools/cluster-tool/conf` under the product installation directory.
- `TC-CONFIG [TC-CONFIG ...]`: A whitespace-separated list of configuration files (minimum 1) that describes the stripes to be added to the cluster.
- `-s HOST[:PORT] [-s HOST[:PORT]...]`: The host:port(s) or host(s) (default port being 9410) of running servers, each specified using the `-s` option . Any one server from each stripe can be provided. However, multiple servers from the same stripe will work as well. The cluster will be configured with the configurations which were originally used to start the servers.

Note: The command configures the cluster only once. To update the configuration of an already configured cluster, the `reconfigure` command should be used.

Examples

- The example below shows a successful execution for a two stripe configuration and a valid license.

```
./cluster-tool.sh configure -l ~/license.xml -n tc-cluster
    ~/tc-config-stripe-1.xml ~/tc-config-stripe-2.xml
Command completed successfully.
```

- The example below shows a failed execution because of an invalid license.

```
./cluster-tool.sh configure -l ~/license.xml
   -n tc-cluster ~/tc-config-stripe-1.xml ~/tc-config-stripe-2.xml
Configure a cluster
Usage:
    configure -n CLUSTER-NAME [-l LICENSE-FILE] TC-CONFIG [TC-CONFIG...]
    configure -n CLUSTER-NAME [-l LICENSE-FILE] -s HOST[:PORT] [-s HOST[:PORT]]... ]
Options:
    -h, --help
        Help
    -l
        License file
    -n (required)
```

```
        Cluster name
    -s
        List of server host:port(s), default port(s) being optional
Error (BAD_REQUEST): com.terracottatech.LicenseException: Invalid license
```

- The example below shows a failed execution with two stripe configurations mismatching in their offheap resource sizes.

```
./cluster-tool.sh configure -n tc-cluster -l
   ~/license.xml ~/tc-config-stripe-1.xml ~/tc-config-stripe-2.xml
Configure a cluster
Usage:
    configure -n CLUSTER-NAME [-l LICENSE-FILE] TC-CONFIG [TC-CONFIG...]
    configure -n CLUSTER-NAME [-l LICENSE-FILE] -s HOST[:PORT] [-s HOST[:PORT]]... ]
Options:
    -h, --help
        Help
    -l
        License file
    -n (required)
        Cluster name
    -s
        List of server host:port(s), default port(s) being optional
Error (BAD_REQUEST): Mismatched off-heap resources in provided config files
 (in order): [[primary-server-resource: 64M], [primary-server-resource: 128M]]
```

The "reconfigure" Command

The `reconfigure` command updates the configuration of a cluster which was configured using the `configure` command. With `reconfigure`, it is possible to:

1. Update the license on the cluster.
2. Add new offheap resources, or grow existing ones.
3. Add new data directories.
4. Change server host, name and/or ports.

The command will fail if any of the following checks do not pass:

1. License checks
   a. The new license is valid.
   b. The new configuration files do not violate the license.
2. Configuration checks
   a. Stripe consistency checks

      The new configuration files are consistent across all the stripes. Refer to the above description of the `plugins` section of the configuration files for details.

   b. Offheap checks

      The new configuration has all the previously configured offheap resources, and the new sizes are not smaller than the old sizes.

   c. Data directories checks

The new configuration has all the previously configured data directory names.

Usage:

```
reconfigure -n CLUSTER-NAME TC-CONFIG [TC-CONFIG...]
reconfigure -n CLUSTER-NAME -l LICENSE-FILE [ -s HOST[:PORT] | TC-CONFIG [TC-CONFIG...] ]
```

Parameters:

- `-n CLUSTER-NAME`: The name of the configured cluster.
- `TC-CONFIG [TC-CONFIG ...]`: A whitespace-separated list of configuration files (minimum 1) that describe the new configurations for the stripes.
- `-l LICENSE-FILE`: The path to the new license file.
- `-s HOST[:PORT]`: A single `host:port` or `host` (default port being 9410) of any running server in the cluster.

`reconfigure` **command usage scenarios:**

1. **License update**

   When it is required to update the license, most likely because the existing license has expired, the following `reconfigure` command syntax should be used:

   ```
   reconfigure -n CLUSTER-NAME -l LICENSE-FILE -s HOST[:PORT]
   ```

   Note: A license update does not require the servers to be restarted.

2. **Configuration update**

   When it is required to update the cluster configuration, the following `reconfigure` command syntax should be used:

   ```
   reconfigure -n CLUSTER-NAME TC-CONFIG [TC-CONFIG...]
   ```

   The steps below should be followed in order:

   a. Update the Terracotta configuration files with the new configuration, ensuring that it meets the reconfiguration criteria mentioned above.

   b. Run the `reconfigure` command with the new configuration files.

   c. Restart the servers for the new configuration to take effect.

3. **License and configuration update at once**

   In the rare event that it is desirable to update the license and the cluster configuration in one go, the following `reconfigure` command syntax should be used:

   ```
   cluster-tool.sh reconfigure -n
       CLUSTER-NAME -l LICENSE-FILE TC-CONFIG [TC-CONFIG...]
   ```

   The steps to be followed here are the same as those mentioned in the *Configuration update* section above.

Examples

- The example below shows a successful re-configuration of a two stripe cluster `tc-cluster` with new stripe configurations.

```
./cluster-tool.sh reconfigure -n tc-cluster
    ~/tc-config-stripe-1.xml ~/tc-config-stripe-2.xml
Command completed successfully.
```

- The example below shows a failed re-configuration because of a license violation.

```
./cluster-tool.sh reconfigure -n tc-cluster
    -l ~/license.xml -s localhost:9410
Prepares the configuration update of a cluster -
requires restart of server(s) with updated configuration(s)
Usage:
    reconfigure -n CLUSTER-NAME TC-CONFIG [TC-CONFIG...]
    reconfigure -n CLUSTER-NAME -l LICENSE-FILE [ -s HOST[:PORT] | TC-CONFIG [TC-CONFIG...] ]
Options:
    -h, --help
        Help
    -l
        License file
    -n (required)
        Cluster name
    -s
        Server host:port, default port being optional
Error (BAD_REQUEST): Cluster offheap resource is not within the limit of the license.
Provided: 2000 MB, but license allows: 1000 MB only
```

- The example below shows a failed re-configuration of a two stripe cluster with new stripe configurations having fewer data directories than existing configuration.

```
./cluster-tool.sh reconfigure -n tc-cluster
    ~/tc-config-stripe-1.xml ~/tc-config-stripe-2.xml
Error (CONFLICT): Mismatched data directories. Provided: [use-for-platform, myData],
but previously configured: [use-for-platform, data, myData]
```

The "status" Command

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

Usage:

```
status -n CLUSTER-NAME HOST[:PORT]
status HOST[:PORT] [HOST[:PORT]...]
```

Parameters:

- `-n CLUSTER-NAME HOST[:PORT]`: The name of the cluster and the `host:port` or `host` (default port being 9410) of a running server in the cluster.
- `HOST[:PORT] [HOST[:PORT]…]`: A list of server `host:port`(s) or `host`(s) in the same or different clusters, the minimum being one.

Examples

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

```
./cluster-tool.sh status -n tc-cluster localhost:9410
------------------------------| CLUSTER: tc-cluster |------------------------------
| STRIPE 1 |
Server Name                    Host:Port                      Status
-----------                    ---------                      ------
```

```
server-1                        localhost:9410                  ACTIVE-COORDINATOR
server-2                        localhost:9610                  PASSIVE-STANDBY
| STRIPE 2 |
Server Name                     Host:Port                       Status
-----------                     ---------                       ------
server-3                        localhost:9710                  ACTIVE-COORDINATOR
server-4                        localhost:9910                  PASSIVE-STANDBY
```

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

```
./cluster-tool.sh status localhost:9410 localhost:9510 localhost:9910
Host:Port                       Status
---------                       ------
localhost:9410                  ACTIVE-COORDINATOR
localhost:9510                  UNKNOWN
localhost:9910                  PASSIVE-STANDBY
```

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.

Usage:

```
dump -n CLUSTER-NAME HOST[:PORT]
dump HOST[:PORT] [HOST[:PORT]...] ]
```

Parameters:

- `-n CLUSTER-NAME HOST[:PORT]`: The name of the cluster and the `host:port` or `host` (default port being 9410) of a running server in the cluster.
- `HOST[:PORT] [HOST[:PORT]…]`: A list of server `host:port`(s) or `host`(s) in the same or different clusters, the minimum being one.

Examples

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

```
./cluster-tool.sh dump -n tc-cluster localhost
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 localhost:9410 localhost:9510 localhost:9910
Dump successful for server at: localhost:9410
Connection refused from server at: localhost:9510
Dump successful for server at: localhost:9910
Error (PARTIAL_FAILURE): Command completed with errors.
```

The "stop" Command

The `stop` command stops the cluster, or particular server(s) in the same or different clusters.

Usage:

```
stop -n CLUSTER-NAME HOST[:PORT]
stop HOST[:PORT] [HOST[:PORT]...]
```

Parameters:

- `-n CLUSTER-NAME HOST[:PORT]`: The name of the cluster and the `host:port` or `host` (default port being 9410) of a running server in the cluster.
- `HOST[:PORT] [HOST[:PORT]…]`: A list of server `host:port`(s) or `host`(s) in the same or different clusters, the minimum being one.

Examples

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

```
./cluster-tool.sh stop -n tc-cluster localhost
Command completed successfully.
```

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

```
./cluster-tool.sh stop localhost:9410 localhost:9510 localhost:9910
Stop successful for server at: localhost:9410
Connection refused from server at: localhost:9510
Stop successful for server at: localhost:9910
Error (PARTIAL_FAILURE): Command completed with errors.
```

The "ipwhitelist-reload" Command

The `ipwhitelist-reload` command reloads the IP white-list on a cluster, or particular server(s) in the same or different clusters. See the section " IP white-listing" on page 57 for details.

Usage:

```
ipwhitelist-reload -n CLUSTER-NAME HOST[:PORT]
ipwhitelist-reload HOST[:PORT] [HOST[:PORT]...] ]
```

Parameters:

- `-n CLUSTER-NAME HOST[:PORT]`: The name of the cluster and the`host:port` or `host` (default port being 9410) of a running server in the cluster.
- `HOST[:PORT] [HOST[:PORT]…]`: A list of server `host:port`(s) or `host`(s) in the same or different clusters, the minimum being one.

Examples

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

```
./cluster-tool.sh ipwhitelist-reload -n tc-cluster localhost
IP white-list reload successful for server at: localhost:9410
IP white-list reload successful for server at: localhost:9610
IP white-list reload successful for server at: localhost:9710
IP white-list reload successful for server at: localhost:9910
Command completed successfully.
```

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

```
./cluster-tool.sh ipwhitelist-reload
    localhost:9410 localhost:9510 localhost:9910
```

```
IP white-list reload successful for server at: localhost:9410
Connection refused from server at: localhost:9510
IP white-list reload successful for server at: localhost:9910
Error (PARTIAL_FAILURE): Command completed with errors.
```

The "backup" Command

The `backup` command takes a backup of the running Terracotta cluster.

Usage:

```
cluster-tool.sh backup -n CLUSTER-NAME HOST[:PORT]
```

Parameters:

- `-n CLUSTER-NAME HOST[:PORT]`: The name of the cluster and the`host:port` or `host` (default port being `9410`) of a running server in the cluster.
- `HOST[:PORT]`: A single`host:port` or `host` (default being `9410`) of any running server in the cluster.

Note: If there is a problem accessing the given server (for example, a slow network connection or an unknown server), the command times out after 60 seconds.

Examples

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

```
./cluster-tool.sh backup -n tc-cluster localhost
PHASE 0: SETTING BACKUP NAME TO : 93cdb93d-ad7c-42aa-9479-6efbdd452302
localhost:9410: SUCCESS
localhost:19410: SUCCESS
localhost:9510: SUCCESS
localhost:19510: SUCCESS
PHASE (1/4): PREPARE_FOR_BACKUP
localhost:9410: SUCCESS
localhost:19410: SUCCESS
localhost:9510: NOOP
localhost:19510: NOOP
PHASE (2/4): ENTER_ONLINE_BACKUP_MODE
localhost:9410: SUCCESS
localhost:19410: SUCCESS
PHASE (3/4): START_BACKUP
localhost:9410: SUCCESS
localhost:19410: SUCCESS
PHASE (4/4): EXIT_ONLINE_BACKUP_MODE
localhost:9410: SUCCESS
localhost:19410: SUCCESS
Command completed successfully!
```

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

```
./cluster-tool.sh backup -n tc-cluster localhost
PHASE 0: SETTING BACKUP NAME TO : 93cdb93d-ad7c-42aa-9479-6efbdd452302
localhost:9410: SUCCESS
localhost:19410: SUCCESS
localhost:9510: SUCCESS
localhost:19510: SUCCESS
PHASE (1/4): PREPARE_FOR_BACKUP
localhost:9410: SUCCESS
localhost:19410: SUCCESS
```

```
localhost:9510: NOOP
localhost:19510: NOOP
PHASE (2/4): ENTER_ONLINE_BACKUP_MODE
localhost:9410: BACKUP_FAILURE
localhost:19410: SUCCESS
PHASE (CLEANUP): ABORT_BACKUP
localhost:9410: SUCCESS
localhost:19410: SUCCESS
Backup failed as some servers '[Server{name='server-1',
host='localhost', port=9410}, [Server{name='server-2',
host='localhost', port=19410}]]', failed to enter online backup mode.
```

13 Licensing

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

Installing a license

A Terracotta license is installed on a Terracotta cluster using the cluster tool `configure` command, thereby enabling cluster configuration and license installation in one go. The command ensures 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 configures a Terracotta cluster using the license file `license.xml`, the name `tc-cluster`, and the configuration file `tc-config.xml`.

```
cluster-tool.sh configure -l license.xml -n tc-cluster tc-config.xml
Command completed successfully.
```

See the section "The Cluster Tool" on page 31 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 cluster tool `reconfigure` command as follows:

```
cluster-tool.sh reconfigure -l license.xml -s localhost:9410
Command completed successfully.
```

See the section "The Cluster Tool" on page 31 for a detailed explanation of the command usage.

14 Backup, Restore and Data Migration

Overview

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 topic *Passive servers* in the section "Active and Passive Servers" on page 7 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 configuration file
2. `server-1` is the server name defined in the server configuration 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 allow the server to use the Backup feature, ensure that you have set up the `<backup-restore>` and `<backup-location>` elements in the server configuration file, as shown in the following snippet:

```
<tc-config>
  <plugins>
    ...
    <service>
      <backup-restore xmlns="http://www.terracottatech.com/config/backup-restore">
        <backup-location path="/path/to/backup/dir" />
      </backup-restore>
    </service>
    ...
  </plugins>
  ...
</tc-config>
```

The path in the `<backup-location>` element can be absolute, or relative to the directory where the configuration file is located. If the directory specified by the backup location path is not present, it will be created during backup.

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 "The Cluster Tool" on page 31 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 `-v` (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
    └── stripe0
        └── 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 configuration file
2. `7c868f83-5075-4b32-bef5-56f29fdcc6f0` is an ID created by the backup command to uniquely identify a backup instance
3. `stripe0` is the stripe sequence in the cluster

4. `server-1` is the server name defined in the server configuration file
5. `datadir` is the data directory name (for a given data directory) defined in the server configuration 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 46 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 configuration 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` attribute of that server in the configuration file. If there is no `name` attribute, skip step 4 mentioned below.
3. Create an empty directory for each path specified by the data directory in your configuration file. This will be the target directory for your restored data. Repeat this step for every data directory path specified in your configuration file.
4. If the `name` attribute for this server is specified, 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 `<stripe-id>/<server-name>/<data-directory>` to this newly created directory. For example, in the example given above, copy from `/tmp/backup1/<backup-name>/stripe0/server-1/data` to `/tmp/data1/server-2`.
6. Start the server with the newly created data directory with the configuration 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.

15 Restarting a Stripe

Restart behavior is closely related to *failover*, but the difference is that the interruption period is typically much longer. A restart waits for the server to return instead of waiting for a new active server to be elected.

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.

16 The Terracotta Configuration File

This document describes the elements of the Terracotta configuration file, which is an XML-formatted file named `tc-config.xml` by default.

This file serves to configure all the members of a single Terracotta Server Array (TSA) stripe. Refer to the section "Cluster Architecture" on page 5 for details on various different TSA topologies.

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

Explanation of Configuration Sections

The Terracotta configuration file is divided into several main, unordered sections. For general configuration purposes, the most relevant sections are:

Section	Description
Servers	The `servers` section defines all the servers which will make up this stripe of a cluster. High-availability is enabled by configuring and running at least 2 servers in this stripe. Note that there is no explicit configuration of which server takes an active or passive role, as that may change over the lifetime of the cluster. Each `server` element in the `servers` section is identified by a name given by the `name` attribute: `<server ... name="`*`ServerName`*`">` To start the server with the name "ServerName", you pass the option `-n ServerName` to the start script. Refer to the section "Starting and Stopping the Terracotta Server" on page 17 for more details.
Plugins	The `plugins` section extends the capabilities of the listed servers by (1) registering and configuring additional `services` made available to those servers, and (2) providing general configuration information made available to all services. The use of extension points is required for the server to work with your application end-points. Refer to the section "Configuring the Terracotta Server" on page 21 for concrete examples.

Section	Description
Properties	The `tc-properties` section exposes a list of key-value pairs to further customize the behavior of the cluster. Note that this section is normally empty.

Simple Configuration Sample

This is an example of a very simple server configuration file.

```
<tc-config xmlns="http://www.terracotta.org/config">
  <servers>
    <server host="localhost" name="testServer0">
      <logs>terracotta-kit-test/testServer0/logs</logs>
      <tsa-port>26270</tsa-port>
      <tsa-group-port>26271</tsa-group-port>
    </server>
  </servers>
</tc-config>
```

This shows the key components of a standard configuration but describes only a single server with no extension points used.

Key points:

- Configuration namespace: "http://www.terracotta.org/config".
- Only a single server with name `testServer0`.
  - Server name is important as it is used when starting a server so it knows which server it is.
  - `localhost` should be replaced with the actual fully-qualified hostname or IP address of the server, in a real deployment.
- A relative path to a logs directory is given.
  - All relative paths are with respect to the location of the containing configuration file.
- The `tsa-port` is the port that clients will use when connecting to the server (default: 9410).
- The `tsa-group-port` is for inter-server communication among stripe members, even though there are no other servers in this case (default: 9530).
- Neither *restartability* nor *failover* would be possible with the above sample configuration, as restart support requires an extension point which provides that capability, and failover requires at least 2 servers in the cluster.

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 most commonly done for hostname, IP address and directory path substitutions.

The following predefined substitutions are available for use:

Parameter	Description
%h	the fully-qualified host name
%i	the IP address
%H	the user's home directory
%n	the username
%o	the operating system name
%a	the processor architecture
%v	the operating system version
%t	the temporary directory (on Linux or Solaris, e.g., `/tmp`)
%(property)	the Java system property of the JVM (e.g. `%(java.home)`, `%(logs.path)`)
%D	the time stamp (yyyyMMddHHmmssSSS)

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

- the `name`, `host` and `bind` attributes of the `<server>` element
- the `logs` child element of the `<server>` element
- data-roots

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.

17 Securing TSA Access using a Permitted IP List

Overview

The IP white-listing feature enables you as the cluster administrator to ensure that only clients from certain explicitly named IP addresses can access the TSA. You can use this feature, for example, to secure the TSA from malicious clients attempting to connect to the TSA. The term "clients" here refers to TSA Clients communicating using the TSA wire protocol.

The so-called white-list is a list of IPs, and clients running on these IPs are allowed to access the TSA; any client whose IP is not in the white-list will not be allowed to access the TSA. You maintain the white-list of known client IPs in a plain text file. CIDR notations can also be used to cover a range of IPs.

Note: It should be understood that usage of this feature (on its own) does not provide a strong level of security for the TSA. Additionally, the ideal way to enforce connection restrictions based on IP addresses would be to use host-level firewalls rather than this feature.

If you want to use white-listing, you need to enable it at server startup. Once the server has started with white-listing enabled, white-listing cannot be turned back off while the server is running. However, you can change the existing IP/CIDR entries in the white-list file while the server is running. Also you can add and delete entries in the white-list file while the server is running, in order to modify the set of clients that need access to the cluster.

If you do not switch on white-listing at server startup, you cannot switch on white-listing while the server is running.

In a multi-stripe cluster, you need to start up all servers (both actives and passives) with the same copy of the white-list file, and when there are updates to the white-list file, you need to ensure that the same changes are mirrored across all stripes. Note that the TSA does not do any cross-stripe validation on the contents of the white-list file, so it is your responsibility as the cluster administrator to make sure that this happens.

Usage

To allow the server to start up using white-listing, ensure that you have set up the `<security>` and `<white-list>` elements in the tc-config configuration file, as in the following snippet:

```
<tc-config ...>
  <plugins>
    ...
    <service>
      <security xmlns="http://www.terracottatech.com/config/security">
        <white-list path="/path/to/white-list-file" />
      </security>
    </service>
```

```
  </plugins> ...
</tc-config>
```

White-listing is enabled/disabled based on the presence of the `<security>` tag in tc-config.

If the <security> tag is not specified, white-listing is disabled. The path in the `<white-list>` element can be absolute, or relative to the folder where the tc-config configuration file is located.

If the file is not found at the specified location, the server startup will fail with an appropriate error message.

If the file is present but there is an error reading the file, the server startup will fail with an appropriate error message. The server IPs specified in the tc-config file of the server are always white-listed.

If hostnames are used in the tc-config entries instead of IPs, the server will attempt to resolve these hostnames to IPs. If the resolution fails, the server startup will fail with an appropriate error message.

So when this IP white-listing feature is used, it is recommended to have only IPs configured for servers in the tc-config file. Similarly, localhost IPs are always white-listed.

White-list file

The white-list file is a simple text file. You can choose any name for this file, for example `white-list.txt`. The entries can be raw IP addresses in IPv4 format or in CIDR notation to represent ranges. IPv6 entries are not supported. Each line in the file can contain either a single IP address or a comma-separated set of IP addresses. Any entry that is not a valid IPv4 address or a valid CIDR will be ignored. Lines starting with a # are skipped. Blank lines are also skipped. Here is a sample white-list file:

```
# The white-list for my cluster
# 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
```

The white-list file must be kept in a directory where you have write permissions.

Dynamic updates

Once a server is started with white-listing enabled, entries can be dynamically added/removed from the white-list file. The updates to the white-list file are processed by the server only when you signal the server to do so. When you have finished making the changes to the white-list file, you must use cluster-tool to notify the server about the change.

Refer to the section "The Cluster Tool" on page 31 for details of usage.

TSA will synchronously reload the white-listed IP/CIDRs from the white-list file when you run the cluster-tool, which prints the error message if an error occurs during reloading. The corresponding log statements can be found in the server log.

As already mentioned, every server node (active and passive) has its own copy of the white-list file. So you need to update the white-list file and execute the script on each node separately. Or you can signal the entire cluster (as configured by the cluster-tool `configure` command) to reload the white-list file on every server in the cluster.

When you run the cluster-tool, the server receives the update signal and reloads the white-list file, and the updated white-list entries are logged in the server log. Thus, after every update operation, you should check the server logs to verify if the updates took effect. Make sure that the updates took effect in all the servers in the cluster.

On every dynamic update, the server reads the contents of the white-list file and the tc-config to update its in-memory white-list. Reading the white-list file involves a disk IO, and reading a tc-config file with hostnames in it involves a DNS lookup for hostname resolution. In both cases, failures can occur. So if such a failure happens after a dynamic update, the updates will be ignored and the server will continue with the current white-list. No partial updates will be applied. The update won't be retried either until the user signals so by running the cluster-tool again.

Client Behaviours

This section details different client behaviors with white-listing enabled.

When a client connects to a server on the tsa-port, the server first accepts the socket connection, then verifies if the IP of the incoming client is white-listed and closes the socket connection if the client is not white-listed. In this case, the client will get an EOF on trying to read from the socket connection established with the server.

If a client was white-listed initially and was removed from the white-list on a dynamic update, it will not be removed immediately from the cluster. Instead, the client will remain connected to the cluster as long as there is no network disconnection between the client and server. The client will be rejected only on the next reconnect attempt.

18 Terracotta Server Migration from Terracotta BigMemory Max v4 to Terracotta DB v10

Terracotta DB 10.x is significantly different from Terracotta BigMemory Max 4.x and Terracotta 3.x in terms of handling of cluster topology, data storage formats, and other functionality. Because of this, you cannot migrate data and configuration for a Terracotta BigMemory Max server to a Terracotta DB server. If you install Terracotta DB on the same machines that host Terracotta BigMemory Max, however, you can find the host names, addresses, and so on that you used for the Terracotta DB BigMemory Max installation in the server configuration files.

19 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.

Disk location

The Terracotta related files can be found under `{installation_root}/TerracottaDB/server/SPM`. This directory contains the following:

1. **bin**: Contains scripts to start and shut down the server.
2. **conf**: Contains Terracotta specific configuration files, including tc-config.xml. If any changes to the configuration are required, such as increasing the offheap or changing the server name, the tc-config.xml file needs to be updated manually.
3. **logs**: Contains Terracotta server and Terracotta DB Platform Manager Plug-in logs.

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 64 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 64 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. The default is the hostname of the machine the server is running on. This name should match the server name in the Terracotta configuration file. Therefore, after changing it from Command Central, proceed to the conf directory mentioned in the " Disk location" on page 63 section above, and update the tc-config.xml with the same name.

  Changes to this configuration will be effective upon a server restart.

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:

- `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.
- `STARTING`: The server instance is starting. This is usually shown when the startup takes longer than expected because either it is a slave (Passive) synchronizing with its master (Active), or it is recovering from an error condition.
- `STOPPING`: The server instance is stopping.
- `STOPPED`: The server instance is not running.
- `FAILED`: The server instance was running, but crashed. Check the server logs to find out 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.