DSO Migration Plan

Introduction

Since DSO is planned to enter maintenance phase with TC 4.0, this document provides guidelines to existing DSO customers to migrate their application usage of DSO to Ehcache. Information in this document addresses technical aspect of the migration on the basis of past customer experiences and in-house expertise, highlighting some of the most common DSO use cases, how we successfully migrated them to Ehcache.

Why move off DSO

Ehcache BigMemory – manage huge cached data sets:

- L2 BigMemory
- L1 BigMemory
  Terracotta In Memory Data Stores brings with it a wealth of performance and simplification benefits. Standard Terracotta installations can take advantage of multi tiered data stores, which in combination with BigMemory adds tremendous capacity, efficiency, and speed to in memory data.

Accessible APIs: All Terracotta products have simple to use, defacto industry standard data management APIs. Use of the standard APIs makes applications easier to build and maintain than with DSO. Along with Terracotta products are embedded with common and advanced data management behavior, which results into quicker adoption

No extensive JVM heap tuning – with the addition of BigMemory in Terracotta product line, developers are now free from spending efforts in tuning heaps to keep application pauses to low. Terracotta BigMemory is smart in moving the right data pieces close to application on demand and manage the capacity of available tiers based on the hot set of data.

Cache lookup through Search – an Ehcache feature that enables an application to look for the cached data by executing arbitrarily complex queries against caches with pre-build indexes.

No explicit locking – no more explicit lock management – built in eventual/strong consistency meets majority of use case requirements.

Toolkit (new) features – Comprehensive support for low level Java collection interfaces. Allows designing of comprehensive feature rich application when combined with the power of BigMemory/Ehcache.

No class instrumentation: DSO requires all shared classes to be instrumented, but Terracotta Ehcache installation requires no class instrumentation. Objects can automatically be shared through Ehcache BigMemory, Terracotta Web Sessions, or using a Clustered Map with the Terracotta Toolkit. The class-portability issues of DSO are no longer a concern.

Integration Without TIMs: since classes do not need to be instrumented in a standard Terracotta installation, explicitly specifying TIMs in Ehcache configuration is not necessary. The DSO errors caused by missing or out-of-date TIMs are no longer an issue. Any frameworks, containers, or other technologies that could not be used with Terracotta due to the lack of a TIM in DSO world can now be integrated with Ehcache with little effort.

Even though Ehcache BigMemory use serialization, their built-in efficiency and substantial advances in streamlined functionality, BigMemory clusters have significantly lower implementation costs, higher performance and improved stability.

Non-Stop cache: Uninterrupted cluster operations. A nonstop cache allows certain cache operations to proceed on clients that have become disconnected from the cluster or if a cache operation cannot complete by the nonstop timeout value – not available in DSO. So if a client node is disconnected from TSA, the application will not stall and keep on working as normal. Disconnect clients automatically connects to cluster as soon as cluster gets back to normal.

Fundamental different between DSO and Express Mode

DSO is primarily adheres to JVM memory model and ensures that JVM semantics are maintained as is when object model is shared across the cluster with other nodes, which means

1) DSO API is Java, it is raw and limited to the behavior and capability provided by Java core classes. This doesn't suffice even for basic enterprise application data management needs. Lot of behavior like data sizing, synchronization with other sources of data, querying, ability to grow beyond java heap capacity, transactions, rollback, handle cluster disconnections will have to built by the application developers, which gets quite complex and error prone even at a very small scale.

2) DSO doesn't distinguish between data stores/containers and the application model. All are treated equal. Since DSO operates on the principal of object identity, any changes done inside the data model are immediately visible to all the entities in the cluster.

In express mode, data store and data model are managed differently. Data stores (BigMemory, Ehcache, Quartz, sessions, Toolkit) are exposes well-understood standard data management APIs. On the other hand application data model doesn't adhere to object identity semantics. Any data stored inside these stores is automatically serialized into byte[] before it is stored and distributed in the cluster. Similarly when application accesses the data from these stores, corresponding byte[] is automatically deserialized into application data model.

Since deserialized copies are detached from underlying clustered byte[], any change made to the data model does not automatically reflect in the deserialized instances application is working against. To ensure that application is working against the latest copy of data, application should always retrieve a new copy from data from store before working on it.

Migration plan for common use cases

To smooth out the DSO migration to express products, there are multiple options –

1) Terracotta Toolkit low level data structures for widely used java interfaces, like Map, Set, List, Queues, ReentrantLock, Barrier etc. Checkout http://terracotta.org/documentation/3.6/terracotta-toolkit/toolkit-usage for details of toolkit APIs. Most of the toolkit containers comes with built in locking semantics and do not require any additional lock management by the application, unless application wants to control the default behavior through explicit locking. To migrate DSO application to Toolkit, choose the appropriate toolkit container and use the corresponding Java interface to interact with the container. Additionally All application objects stored inside toolkit should be marked serialized.

2) Our recommend approach is to use BigMemory/Ehcache for migration, which is one of the most advanced in memory data store. Ehcache is defacto in memory data management standard and provides simple, familiar APIs. Remaining section outlines the details of DSO to Ehcache API migration for most of the common DSO usage –

- **Locks (sync block, re-entrant R/W lock)** – Ehcache consistency types can safely replace DSO locks and sync blocks.

Ehcache primarily has two consistency modes – Eventual and Strong. When set to "eventual", allows reads without locks, which means the cache may temporarily return stale data in exchange for substantially improved performance. When set to "strong", guarantees that after any update is completed no local read can return a stale value, but at a potentially high cost of performance.

For more advanced use cases, Ehcache support JTA for commit and rollback of multiple updates in single transaction, bulk load for cache warm and Explicit locking APIs to have a fine grained control on Ehcache locking boundaries. Checkout following links for details of locking/consistency APIs

http://ehcache.org/documentation/get-started/consistency-options

http://ehcache.org/documentation/apis/transactions

http://ehcache.org/documentation/apis/bulk-loading

http://ehcache.org/documentation/apis/explicitlocking

- **Collections –**

1. Single level Map can be directly converted to Ehcache. Efforts involved are:

    a.Mark application objects as Serializable and replace the usage of Maps by Ehcache.

    b. Flatten the object graph – if serialized size of the object stored inside Ehcache is expected to be large (in the region of few hundred KBs or MBs), then it is advised to break the object model into multiple smaller chunks and store each chunk as a separate unit inside Ehcache.

    c. Instead of using Java Map, use BigMemory/Ehcache to store data, e.g. replace map.put(key, value) by cache.put(new Element(key, value)). Similarly, map.get(key) calls by cache.get(key). More info on Ehcache config and APIs:

    http://ehcache.org/documentation/2.5/configuration/configuration

    http://ehcache.org/apidocs/

2. Map of maps – In DSO, many applications use multi level Java Maps to store application date. E.g. in a typical web application where a session has to be created/maintained for a user, each individual user session maintains user state inside a Map, which intern is stored inside another Map.

In express mode, map of maps can be implemented in two ways:

- Use Terracotta Toolkit Map - obtain a clustered instance of Map from toolkit for every user session and store session data inside it, e.g.

    ClusteredMap sessionMap = Toolkit.getMap(SessionID);

    This essentially gets rid of first level map and creates a new instance of clustered map for each user session.

    More details on toolkit maps here: http://www.terracotta.org/documentation/bigmemorymax/terracotta-toolkit/toolkit-usage

- Use Ehcache - converting to Ehcache would depend on the number of elements in second level map and size of each element:

    - If serialized of second level map is not quite high (> few hundreds) then do the following:

        - Create one parent Ehcache
        - Directly store second level serialized Java Maps as it is inside parent Ehcache.

This essentially means that now there is only one level of mapping of elements in cache, child map is serialized and stored as a cache value object. This approach works well as long as cost of serialization/deserialization of second level maps is not large enough to impact the performance.

- If serialized size of second level map is relatively high then consider following approach -
    - Create one parent Ehcache
    - Flatten out each entry of second level maps into parent Ehcache. This can be done by storing each value of second level map using a combination parent key and second level map key.

  E.g. if first level map has 1000 maps, and each of the second level map has 10k elements, the total number of elements in the parent cache would become 1000 * 10k = 10 million.

Parent map:

parentKey1	childMap1
.	.
.	.
parentKey1000	childMap1000

And child map:

childKey1	childValue1
.	.
.	.
childKey10k	childValue10k

So the cache store becomes:

parentKey1childKey1	childValue1
.	.
.	.
parentKey1childKey10k	childValue10k
parentKey2childKey1	childValue1
.	.

.	.
parentKey2childKey10k	childValue10k
.	.

Queues – To migrate queues, choose one of the following strategy –

1. Use Toolkit BlockingQueue and push the message directly through queues
2. If Message size if relatively larger (in KBs or more), store the actual messages inside a Ehcache and only push the messages unique Ehcache identifier through Toolkit Queue. This approach is expected to scale better than directly using Toolkit for complete message transport.

- **Honor-transient**: In DSO, the "honor-transient" option, if set to "true," tells Terracotta to honor the transience of a field in the class as declared by the "transient" keyword. This will direct Terracotta to ignore fields marked as transient and not include those fields in the shared object graph. This is similar to how the transient keyword directs the Java serialization mechanism to leave transient fields out of the serialized object stream.

  In Ehcache, honor-transient configuration is not required in tc-config, instead declare the field in question as java transient.

- **DMI** – Any method of an object contained in a shared object graph can be distributed, meaning that an invocation of that method in one virtual machine will trigger the same method invocation on all the mirrored instances in other virtual machines. This was a useful mechanism for implementing a distributed listener model.

  Ehcache Events in Ehcache replace DMI – Cache events are fired for below mentioned cache operations and are captured by all application nodes using Ehcache:

    - Evictions - An eviction on a client generates an eviction event on that client. An eviction on a Terracotta server fires an event on a random client.
    - Puts - A put() on a client generates a put event on that client.
    - Updates - an update on a client generates a put event on that client.

  More details at: http://www.terracotta.org/documentation/enterprise-ehcache/configuration-guide#20075

Other configuration changes

- Simplify tc-config.xml by removing the entire configuration for Instruments and Locks.
- Remove explicit TIMs configurations
- Remove DSO annotations from Java code