Developing for {brandname} 10.0

Table of Contents

Chapter 1. The Cache API

1.1. The Cache interface

{brandname}'s Caches are manipulated through the Cache interface.

A Cache exposes simple methods for adding, retrieving and removing entries, including atomic mechanisms exposed by the JDK's ConcurrentMap interface. Based on the cache mode used, invoking these methods will trigger a number of things to happen, potentially even including replicating an entry to a remote node or looking up an entry from a remote node, or potentially a cache store.

For simple usage, using the Cache API should be no different from using the JDK Map API, and hence migrating from simple in-memory caches based on a Map to {brandname}'s Cache should be trivial.

1.1.1. Performance Concerns of Certain Map Methods

Certain methods exposed in Map have certain performance consequences when used with {brandname}, such as size() , values() , keySet() and entrySet() . Specific methods on the `keySet`, `values` and `entrySet` are fine for use please see their Javadoc for further details.

Attempting to perform these operations globally would have large performance impact as well as become a scalability bottleneck. As such, these methods should only be used for informational or debugging purposes only.

It should be noted that using certain flags with the withFlags method can mitigate some of these concerns, please check each method's documentation for more details.

1.1.2. Mortal and Immortal Data

Further to simply storing entries, {brandname}'s cache API allows you to attach mortality information to data. For example, simply using put(key, value) would create an *immortal* entry, i.e., an entry that lives in the cache forever, until it is removed (or evicted from memory to prevent running out of memory). If, however, you put data in the cache using put(key, value, lifespan, timeunit) , this creates a *mortal* entry, i.e., an entry that has a fixed lifespan and expires after that lifespan.

In addition to *lifespan* , {brandname} also supports *maxIdle* as an additional metric with which to determine expiration. Any combination of lifespans or maxIdles can be used.

1.1.3. Expiration and Mortal Data

See expiration for more information about using mortal data with {brandname}.

1.1.4. putForExternalRead operation

{brandname}'s Cache class contains a different 'put' operation called putForExternalRead . This operation is particularly useful when {brandname} is used as a temporary cache for data that is persisted elsewhere. Under heavy read scenarios, contention in the cache should not delay the real transactions at hand, since caching should just be an optimization and not something that gets in the way.

To achieve this, putForExternalRead acts as a put call that only operates if the key is not present in the cache, and fails fast and silently if another thread is trying to store the same key at the same time. In this particular scenario, caching data is a way to optimise the system and it's not desirable that a failure in caching affects the on-going transaction, hence why failure is handled differently. putForExternalRead is considered to be a fast operation because regardless of whether it's successful or not, it doesn't wait for any locks, and so returns to the caller promptly.

To understand how to use this operation, let's look at basic example. Imagine a cache of Person instances, each keyed by a PersonId , whose data originates in a separate data store. The following code shows the most common pattern of using putForExternalRead within the context of this example:

```
// Id of the person to look up, provided by the application
PersonId id = ...;

// Get a reference to the cache where person instances will be stored
Cache<PersonId, Person> cache = ...;

// First, check whether the cache contains the person instance
// associated with with the given id
Person cachedPerson = cache.get(id);

if (cachedPerson == null) {
   // The person is not cached yet, so query the data store with the id
   Person person = dataStore.lookup(id);

   // Cache the person along with the id so that future requests can
   // retrieve it from memory rather than going to the data store
   cache.putForExternalRead(id, person);
} else {
   // The person was found in the cache, so return it to the application
   return cachedPerson;
}
```

Please note that putForExternalRead should never be used as a mechanism to update the cache with a new Person instance originating from application execution (i.e. from a transaction that modifies a Person's address). When updating cached values, please use the standard put operation, otherwise the possibility of caching corrupt data is likely.

1.2. The AdvancedCache interface

In addition to the simple Cache interface, {brandname} offers an AdvancedCache interface, geared towards extension authors. The AdvancedCache offers the ability to inject custom interceptors, access certain internal components and to apply flags to alter the default behavior of certain cache methods. The following code snippet depicts how an AdvancedCache can be obtained:

```
AdvancedCache advancedCache = cache.getAdvancedCache();
```

1.2.1. Flags

Flags are applied to regular cache methods to alter the behavior of certain methods. For a list of all available flags, and their effects, see the Flag enumeration. Flags are applied using AdvancedCache.withFlags() . This builder method can be used to apply any number of flags to a cache invocation, for example:

```
advancedCache.withFlags(Flag.CACHE_MODE_LOCAL, Flag.SKIP_LOCKING)
   .withFlags(Flag.FORCE_SYNCHRONOUS)
   .put("hello", "world");
```

1.2.2. Custom Interceptors

The AdvancedCache interface also offers advanced developers a mechanism with which to attach custom interceptors. Custom interceptors allow developers to alter the behavior of the cache API methods, and the AdvancedCache interface allows developers to attach these interceptors programmatically, at run-time. See the AdvancedCache Javadocs for more details.

1.3. Listeners and Notifications

{brandname} offers a listener API, where clients can register for and get notified when events take place. This annotation-driven API applies to 2 different levels: cache level events and cache manager level events.

Events trigger a notification which is dispatched to listeners. Listeners are simple POJO s annotated with @Listener and registered using the methods defined in the Listenable interface.

Both Cache and CacheManager implement Listenable, which means you can attach listeners to either a cache or a cache manager, to receive either cache-level or cache manager-level notifications.

For example, the following class defines a listener to print out some information every time a new entry is added to the cache, in a non blocking fashion:

```
@Listener
public class PrintWhenAdded {
  Queue<CacheEntryCreatedEvent> events = new ConcurrentLinkedQueue<>();

  @CacheEntryCreated
  public CompletionStage<Void> print(CacheEntryCreatedEvent event) {
    events.add(event);
    return null;
  }

}
```

For more comprehensive examples, please see the Javadocs for @Listener.

1.3.1. Cache-level notifications

Cache-level events occur on a per-cache basis, and by default are only raised on nodes where the events occur. Note in a distributed cache these events are only raised on the owners of data being affected. Examples of cache-level events are entries being added, removed, modified, etc. These events trigger notifications to listeners registered to a specific cache.

Please see the Javadocs on the org.infinispan.notifications.cachelistener.annotation package for a comprehensive list of all cache-level notifications, and their respective method-level annotations.

Please refer to the Javadocs on the org.infinispan.notifications.cachelistener.annotation package for the list of cache-level notifications available in {brandname}.

Cluster Listeners

The cluster listeners should be used when it is desirable to listen to the cache events on a single node.

To do so all that is required is set to annotate your listener as being clustered.

```
@Listener (clustered = true)
public class MyClusterListener { .... }
```

There are some limitations to cluster listeners from a non clustered listener.

1. A cluster listener can only listen to `@CacheEntryModified`, `@CacheEntryCreated`, `@CacheEntryRemoved` and `@CacheEntryExpired` events. Note this means any other type of event will not be listened to for this listener.
2. Only the post event is sent to a cluster listener, the pre event is ignored.

Event filtering and conversion

All applicable events on the node where the listener is installed will be raised to the listener. It is possible to dynamically filter what events are raised by using a KeyFilter (only allows filtering on keys) or CacheEventFilter (used to filter for keys, old value, old metadata, new value, new metadata, whether command was retried, if the event is before the event (ie. isPre) and also the command type).

The example here shows a simple `KeyFilter` that will only allow events to be raised when an event modified the entry for the key `Only Me`.

```
public class SpecificKeyFilter implements KeyFilter<String> {
    private final String keyToAccept;

    public SpecificKeyFilter(String keyToAccept) {
      if (keyToAccept == null) {
        throw new NullPointerException();
      }
      this.keyToAccept = keyToAccept;
    }

    boolean accept(String key) {
      return keyToAccept.equals(key);
    }
}

...
cache.addListener(listener, new SpecificKeyFilter("Only Me"));
...
```

This can be useful when you want to limit what events you receive in a more efficient manner.

There is also a CacheEventConverter that can be supplied that allows for converting a value to another before raising the event. This can be nice to modularize any code that does value conversions.

The mentioned filters and converters are especially beneficial when used in conjunction with a Cluster Listener. This is because the filtering and conversion is done on the node where the event originated and not on the node where event is listened to. This can provide benefits of not having to replicate events across the cluster (filter) or even have reduced payloads (converter).

Initial State Events

When a listener is installed it will only be notified of events after it is fully installed.

It may be desirable to get the current state of the cache contents upon first registration of listener by having an event generated of type `@CacheEntryCreated` for each element in the cache. Any additionally generated events during this initial phase will be queued until appropriate events have

been raised.

This only works for clustered listeners at this time. ISPN-4608 covers adding this for non clustered listeners.

Duplicate Events

It is possible in a non transactional cache to receive duplicate events. This is possible when the primary owner of a key goes down while trying to perform a write operation such as a put.

{brandname} internally will rectify the put operation by sending it to the new primary owner for the given key automatically, however there are no guarantees in regards to if the write was first replicated to backups. Thus more than 1 of the following write events (`CacheEntryCreatedEvent`, `CacheEntryModifiedEvent` & `CacheEntryRemovedEvent`) may be sent on a single operation.

If more than one event is generated {brandname} will mark the event that it was generated by a retried command to help the user to know when this occurs without having to pay attention to view changes.

```
@Listener
public class MyRetryListener {
  @CacheEntryModified
  public void entryModified(CacheEntryModifiedEvent event) {
    if (event.isCommandRetried()) {
      // Do something
    }
  }
}
```

Also when using a `CacheEventFilter` or `CacheEventConverter` the EventType contains a method `isRetry` to tell if the event was generated due to retry.

1.3.2. Cache manager-level notifications

Cache manager-level events occur on a cache manager. These too are global and cluster-wide, but involve events that affect all caches created by a single cache manager. Examples of cache manager-level events are nodes joining or leaving a cluster, or caches starting or stopping.

Please see the Javadocs on the org.infinispan.notifications.cachemanagerlistener.annotation package for a comprehensive list of all cache manager-level notifications, and their respective method-level annotations.

1.3.3. Synchronicity of events

By default, all async notifications are dispatched in the notification thread pool. Sync notifications will delay the operation from continuing until the listener method completes or the CompletionStage completes (the former causing the thread to block). Alternatively, you could annotate your listener as *asynchronous* in which case the operation will continue immediately,

while the notification is completed asynchronously on the notification thread pool. To do this, simply annotate your listener such:

Asynchronous Listener

```
@Listener (sync = false)
public class MyAsyncListener {
   @CacheEntryCreated
   void listen(CacheEntryCreatedEvent event) { }
}
```

Blocking Synchronous Listener

```
@Listener
public class MySyncListener {
   @CacheEntryCreated
   void listen(CacheEntryCreatedEvent event) { }
}
```

Non-Blocking Listener

```
@Listener
public class MyNonBlockingListener {
   @CacheEntryCreated
   CompletionStage<Void> listen(CacheEntryCreatedEvent event) { }
}
```

Asynchronous thread pool

To tune the thread pool used to dispatch such asynchronous notifications, use the `<listener-executor />` XML element in your configuration file.

1.4. Asynchronous API

In addition to synchronous API methods like Cache.put() , Cache.remove() , etc., {brandname} also has an asynchronous, non-blocking API where you can achieve the same results in a non-blocking fashion.

These methods are named in a similar fashion to their blocking counterparts, with "Async" appended. E.g., Cache.putAsync() , Cache.removeAsync() , etc. These asynchronous counterparts return a Future containing the actual result of the operation.

For example, in a cache parameterized as `Cache<String, String>`, `Cache.put(String key, String value)` returns a `String`. `Cache.putAsync(String key, String value)` would return a `Future<String>`.

1.4.1. Why use such an API?

Non-blocking APIs are powerful in that they provide all of the guarantees of synchronous communications - with the ability to handle communication failures and exceptions - with the ease of not having to block until a call completes. This allows you to better harness parallelism in your system. For example:

```
Set<Future<?>> futures = new HashSet<Future<?>>();
futures.add(cache.putAsync(key1, value1)); // does not block
futures.add(cache.putAsync(key2, value2)); // does not block
futures.add(cache.putAsync(key3, value3)); // does not block

// the remote calls for the 3 puts will effectively be executed
// in parallel, particularly useful if running in distributed mode
// and the 3 keys would typically be pushed to 3 different nodes
// in the cluster

// check that the puts completed successfully
for (Future<?> f: futures) f.get();
```

1.4.2. Which processes actually happen asynchronously?

There are 4 things in {brandname} that can be considered to be on the critical path of a typical write operation. These are, in order of cost:

- network calls
- marshalling
- writing to a cache store (optional)
- locking

As of {brandname} 4.0, using the async methods will take the network calls and marshalling off the critical path. For various technical reasons, writing to a cache store and acquiring locks, however, still happens in the caller's thread. In future, we plan to take these offline as well. See this developer mail list thread about this topic.

1.4.3. Notifying futures

Strictly, these methods do not return JDK Futures, but rather a sub-interface known as a NotifyingFuture . The main difference is that you can attach a listener to a NotifyingFuture such that you could be notified when the future completes. Here is an example of making use of a notifying future:

```
FutureListener futureListener = new FutureListener() {

   public void futureDone(Future future) {
      try {
         future.get();
      } catch (Exception e) {
         // Future did not complete successfully
         System.out.println("Help!");
      }
   }
};

cache.putAsync("key", "value").attachListener(futureListener);
```

1.4.4. Further reading

The Javadocs on the Cache interface has some examples on using the asynchronous API, as does this article by Manik Surtani introducing the API.

1.5. Invocation Flags

An important aspect of getting the most of {brandname} is the use of per-invocation flags in order to provide specific behaviour to each particular cache call. By doing this, some important optimizations can be implemented potentially saving precious time and network resources. One of the most popular usages of flags can be found right in Cache API, underneath the putForExternalRead() method which is used to load an {brandname} cache with data read from an external resource. In order to make this call efficient, {brandname} basically calls a normal put operation passing the following flags: FAIL_SILENTLY , FORCE_ASYNCHRONOUS , ZERO_LOCK_ACQUISITION_TIMEOUT

What {brandname} is doing here is effectively saying that when putting data read from external read, it will use an almost-zero lock acquisition time and that if the locks cannot be acquired, it will fail silently without throwing any exception related to lock acquisition. It also specifies that regardless of the cache mode, if the cache is clustered, it will replicate asynchronously and so won't wait for responses from other nodes. The combination of all these flags make this kind of operation very efficient, and the efficiency comes from the fact this type of *putForExternalRead* calls are used with the knowledge that client can always head back to a persistent store of some sorts to retrieve the data that should be stored in memory. So, any attempt to store the data is just a best effort and if not possible, the client should try again if there's a cache miss.

1.5.1. Examples

If you want to use these or any other flags available, which by the way are described in detail the Flag enumeration , you simply need to get hold of the advanced cache and add the flags you need via the withFlags() method call. For example:

```
Cache cache = ...
cache.getAdvancedCache()
   .withFlags(Flag.SKIP_CACHE_STORE, Flag.CACHE_MODE_LOCAL)
   .put("local", "only");
```

It's worth noting that these flags are only active for the duration of the cache operation. If the same flags need to be used in several invocations, even if they're in the same transaction, withFlags() needs to be called repeatedly. Clearly, if the cache operation is to be replicated in another node, the flags are carried over to the remote nodes as well.

Suppressing return values from a put() or remove()

Another very important use case is when you want a write operation such as put() to *not* return the previous value. To do that, you need to use two flags to make sure that in a distributed environment, no remote lookup is done to potentially get previous value, and if the cache is configured with a cache loader, to avoid loading the previous value from the cache store. You can see these two flags in action in the following example:

```
Cache cache = ...
cache.getAdvancedCache()
   .withFlags(Flag.SKIP_REMOTE_LOOKUP, Flag.SKIP_CACHE_LOAD)
   .put("local", "only")
```

For more information, please check the Flag enumeration javadoc.

Chapter 2. Functional Map API

{brandname} 8 introduces a new experimental API for interacting with your data which takes advantage of the functional programming additions and improved asynchronous programming capabilities available in Java 8.

{brandname}'s Functional Map API is a distilled map-like asynchronous API which uses functions to interact with data.

2.1. Asynchronous and Lazy

Being an asynchronous API, all methods that return a single result, return a CompletableFuture which wraps the result, so you can use the resources of your system more efficiently by having the possibility to receive callbacks when the `CompletableFuture` has completed, or you can chain or compose them with other CompletableFuture.

For those operations that return multiple results, the API returns instances of a `Traversable` interface which offers a lazy pull-style API for working with multiple results. `Traversable` , being a lazy pull-style API, can still be asynchronous underneath since the user can decide to work on the traversable at a later stage, and the `Traversable` implementation itself can decide when to compute those results.

2.2. Function transparency

Since the content of the functions is transparent to {brandname}, the API has been split into 3 interfaces for read-only (`ReadOnlyMap`), read-write (`ReadWriteMap`) and write-only (`WriteOnlyMap`) operations respectively, in order to provide hints to the {brandname} internals on the type of work needed to support functions.

2.3. Constructing Functional Maps

To construct any of the read-only, write-only or read-write map instances, an {brandname} `AdvancedCache` is required, which is retrieved from the Cache Manager, and using the `AdvancedCache` , static method factory methods are used to create `ReadOnlyMap` , `ReadWriteMap` or `WriteOnlyMap`

```
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.functional.impl.*;

AdvancedCache<String, String> cache = ...

FunctionalMapImpl<String, String> functionalMap = FunctionalMapImpl.create(cache);
ReadOnlyMap<String, String> readOnlyMap = ReadOnlyMapImpl.create(functionalMap);
WriteOnlyMap<String, String> writeOnlyMap = WriteOnlyMapImpl.create(functionalMap);
ReadWriteMap<String, String> readWriteMap = ReadWriteMapImpl.create(functionalMap);
```

At this stage, the Functional Map API is experimental and hence the way FunctionalMap, ReadOnlyMap, WriteOnlyMap and ReadWriteMap are constructed is temporary.

2.4. Read-Only Map API

Read-only operations have the advantage that no locks are acquired for the duration of the operation. Here's an example on how to the equivalent operation for Map.get(K):

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

ReadOnlyMap<String, String> readOnlyMap = ...
CompletableFuture<Optional<String>> readFuture = readOnlyMap.eval("key1",
ReadEntryView::find);
readFuture.thenAccept(System.out::println);
```

Read-only map also exposes operations to retrieve multiple keys in one go:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Traversable;

ReadOnlyMap<String, String> readOnlyMap = ...

Set<String> keys = new HashSet<>(Arrays.asList("key1", "key2"));
Traversable<String> values = readOnlyMap.evalMany(keys, ReadEntryView::get);
values.forEach(System.out::println);
```

Finally, read-only map also exposes methods to read all existing keys as well as entries, which include both key and value information.

2.4.1. Read-Only Entry View

The function parameters for read-only maps provide the user with a read-only entry view to interact with the data in the cache, which include these operations:

- `key()` method returns the key for which this function is being executed.
- `find()` returns a Java 8 `Optional` wrapping the value if present, otherwise it returns an empty optional. Unless the value is guaranteed to be associated with the key, it's recommended to use `find()` to verify whether there's a value associated with the key.
- `get()` returns the value associated with the key. If the key has no value associated with it, calling `get()` throws a `NoSuchElementException`. `get()` can be considered as a shortcut of `ReadEntryView.find().get()` which should be used only when the caller has guarantees that there's definitely a value associated with the key.

- `findMetaParam(Class<T> type)` allows metadata parameter information associated with the cache entry to be looked up, for example: entry lifespan, last accessed time...etc. See Metadata Parameter Handling to find out more.

2.5. Write-Only Map API

Write-only operations include operations that insert or update data in the cache and also removals. Crucially, a write-only operation does not attempt to read any previous value associated with the key. This is an important optimization since that means neither the cluster nor any persistence stores will be looked up to retrieve previous values. In the main {brandname} Cache, this kind of optimization was achieved using a local-only per-invocation flag, but the use case is so common that in this new functional API, this optimization is provided as a first-class citizen.

Using write-only map API , an operation equivalent to `javax.cache.Cache (JCache)` 's void returning `put` can be achieved this way, followed by an attempt to read the stored value using the read-only map API:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", "value1",
   (v, view) -> view.set(v));
CompletableFuture<String> readFuture = writeFuture.thenCompose(r ->
   readOnlyMap.eval("key1", ReadEntryView::get));
readFuture.thenAccept(System.out::println);
```

Multiple key/value pairs can be stored in one go using `evalMany` API:

```
WriteOnlyMap<String, String> writeOnlyMap = ...

Map<K, String> data = new HashMap<>();
data.put("key1", "value1");
data.put("key2", "value2");
CompletableFuture<Void> writerAllFuture = writeOnlyMap.evalMany(data, (v, view) ->
view.set(v));
writerAllFuture.thenAccept(x -> "Write completed");
```

To remove all contents of the cache, there are two possibilities with different semantics. If using `evalAll` each cached entry is iterated over and the function is called with that entry's information. Using this method also results in listeners being invoked. See functional listeners for more information.

```
WriteOnlyMap<String, String> writeOnlyMap = ...

CompletableFuture<Void> removeAllFuture = writeOnlyMap.evalAll(WriteEntryView::remove
);
removeAllFuture.thenAccept(x -> "All entries removed");
```

The alternative way to remove all entries is to call `truncate` operation which clears the entire cache contents in one go without invoking any listeners and is best-effort:

```
WriteOnlyMap<String, String> writeOnlyMap = ...

CompletableFuture<Void> truncateFuture = writeOnlyMap.truncate();
truncateFuture.thenAccept(x -> "Cache contents cleared");
```

2.5.1. Write-Only Entry View

The function parameters for write-only maps provide the user with a write-only entry view to modify the data in the cache, which include these operations:

- `set(V, MetaParam.Writable…)` method allows for a new value to be associated with the cache entry for which this function is executed, and it optionally takes zero or more metadata parameters to be stored along with the value. See Metadata Parameter Handling for more information.
- `remove()` method removes the cache entry, including both value and metadata parameters associated with this key.

2.6. Read-Write Map API

The final type of operations we have are readwrite operations, and within this category CAS-like (CompareAndSwap) operations can be found. This type of operations require previous value associated with the key to be read and for locks to be acquired before executing the function. The vast majority of operations within `ConcurrentMap` and `JCache` APIs fall within this category, and they can easily be implemented using the read-write map API . Moreover, with read-write map API , you can make CASlike comparisons not only based on value equality but based on metadata parameter equality such as version information, and you can send back previous value or boolean instances to signal whether the CASlike comparison succeeded.

Implementing a write operation that returns the previous value associated with the cache entry is easy to achieve with the read-write map API:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

ReadWriteMap<String, String> readWriteMap = ...

CompletableFuture<Optional<String>> readWriteFuture = readWriteMap.eval("key1",
"value1",
   (v, view) -> {
      Optional<V> prev = rw.find();
      view.set(v);
      return prev;
   });
readWriteFuture.thenAccept(System.out::println);
```

`ConcurrentMap.replace(K, V, V)` is a replace function that compares the value present in the map and if it's equals to the value passed in as first parameter, the second value is stored, returning a boolean indicating whether the replace was successfully completed. This operation can easily be implemented using the read-write map API:

```
ReadWriteMap<String, String> readWriteMap = ...

String oldValue = "old-value";
CompletableFuture<Boolean> replaceFuture = readWriteMap.eval("key1", "value1", (v,
view) -> {
   return view.find().map(prev -> {
      if (prev.equals(oldValue)) {
         rw.set(v);
         return true; // previous value present and equals to the expected one
      }
      return false; // previous value associated with key does not match
   }).orElse(false); // no value associated with this key
});
replaceFuture.thenAccept(replaced -> System.out.printf("Value was replaced? %s%n",
replaced));
```

The function in the example above captures `oldValue` which is an external value to the function which is valid use case.

Read-write map API contains `evalMany` and `evalAll` operations which behave similar to the write-only map offerings, except that they enable previous value and metadata parameters to be read.

2.6.1. Read-Write Entry View

The function parameters for read-write maps provide the user with the possibility to query the information associated with the key, including value and metadata parameters, and the user can also use this read-write entry view to modify the data in the cache.

The operations are exposed by read-write entry views are a union of the operations exposed by read-only entry views and write-only entry views.

2.7. Metadata Parameter Handling

Metadata parameters provide extra information about the cache entry, such as version information, lifespan, last accessed/used time...etc. Some of these can be provided by the user, e.g. version, lifespan...etc, but some others are computed internally and can only be queried, e.g. last accessed/used time.

The functional map API provides a flexible way to store metadata parameters along with an cache entry. To be able to store a metadata parameter, it must extend `MetaParam.Writable` interface, and implement the methods to allow the internal logic to extra the data. Storing is done via the `set(V, MetaParam.Writable…)` method in the write-only entry view or read-write entry view function parameters.

Querying metadata parameters is available via the `findMetaParam(Class)` method available via read-write entry view or read-only entry views or function parameters.

Here is an example showing how to store metadata parameters and how to query them:

```
import java.time.Duration;
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.MetaParam.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", "value1",
   (v, view) -> view.set(v, new MetaLifespan(Duration.ofHours(1).toMillis())));
CompletableFuture<MetaLifespan> readFuture = writeFuture.thenCompose(r ->
   readOnlyMap.eval("key1", view -> view.findMetaParam(MetaLifespan.class).get()));
readFuture.thenAccept(System.out::println);
```

If the metadata parameter is generic, for example `MetaEntryVersion<T>` , retrieving the metadata parameter along with a specific type can be tricky if using `.class` static helper in a class because it does not return a `Class<T>` but only `Class`, and hence any generic information in the class is lost:

```
ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<String> readFuture = readOnlyMap.eval("key1", view -> {
   // If caller depends on the typed information, this is not an ideal way to retrieve
it
   // If the caller does not depend on the specific type, this works just fine.
   Optional<MetaEntryVersion> version = view.findMetaParam(MetaEntryVersion.class);
   return view.get();
});
```

When generic information is important the user can define a static helper method that coerces the static class retrieval to the type requested, and then use that helper method in the call to `findMetaParam`:

```
class MetaEntryVersion<T> implements MetaParam.Writable<EntryVersion<T>> {
   ...
   public static <T> T type() { return (T) MetaEntryVersion.class; }
   ...
}

ReadOnlyMap<String, String> readOnlyMap = ...

CompletableFuture<String> readFuture = readOnlyMap.eval("key1", view -> {
   // The caller wants guarantees that the metadata parameter for version is numeric
   // e.g. to query the actual version information
   Optional<MetaEntryVersion<Long>> version = view.findMetaParam(MetaEntryVersion.
type());
   return view.get();
});
```

Finally, users are free to create new instances of metadata parameters to suit their needs. They are stored and retrieved in the very same way as done for the metadata parameters already provided by the functional map API.

2.8. Invocation Parameter

Per-invocation parameters are applied to regular functional map API calls to alter the behaviour of certain aspects. Adding per invocation parameters is done using the `withParams(Param<?>…)` method.

`Param.FutureMode` tweaks whether a method returning a `CompletableFuture` will span a thread to invoke the method, or instead will use the caller thread. By default, whenever a call is made to a method returning a `CompletableFuture` , a separate thread will be span to execute the method asynchronously. However, if the caller will immediately block waiting for the `CompletableFuture` to complete, spanning a different thread is wasteful, and hence `Param.FutureMode.COMPLETED` can be passed as per-invocation parameter to avoid creating that extra thread. Example:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Param.*;

ReadOnlyMap<String, String> readOnlyMap = ...
ReadOnlyMap<String, String> readOnlyMapCompleted = readOnlyMap.withParams(FutureMode
.COMPLETED);
Optional<String> readFuture = readOnlyMapCompleted.eval("key1", ReadEntryView::find)
.get();
```

Param.PersistenceMode controls whether a write operation will be propagated to a persistence store. The default behaviour is for all write-operations to be propagated to the persistence store if the cache is configured with a persistence store. By passing PersistenceMode.SKIP as parameter, the write operation skips the persistence store and its effects are only seen in the in-memory contents of the cache. PersistenceMode.SKIP can be used to implement an `Cache.evict()` method which removes data from memory but leaves the persistence store untouched:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Param.*;

WriteOnlyMap<String, String> writeOnlyMap = ...
WriteOnlyMap<String, String> skiPersistMap = writeOnlyMap.withParams(PersistenceMode
.SKIP);
CompletableFuture<Void> removeFuture = skiPersistMap.eval("key1", WriteEntryView:
:remove);
```

Note that there's no need for another PersistenceMode option to skip reading from the persistence store, because a write operation can skip reading previous value from the store by calling a write-only operation via the WriteOnlyMap.

Finally, new Param implementations are normally provided by the functional map API since they tweak how the internal logic works. So, for the most part of users, they should limit themselves to using the Param instances exposed by the API. The exception to this rule would be advanced users who decide to add new interceptors to the internal stack. These users have the ability to query these parameters within the interceptors.

2.9. Functional Listeners

The functional map offers a listener API, where clients can register for and get notified when events take place. These notifications are post-event, so that means the events are received after the event has happened.

The listeners that can be registered are split into two categories: write listeners and read-write listeners.

2.9.1. Write Listeners

Write listeners enable user to register listeners for any cache entry write events that happen in either a read-write or write-only functional map.

Listeners for write events cannot distinguish between cache entry created and cache entry modify/update events because they don't have access to the previous value. All they know is that a new non-null entry has been written.

However, write event listeners can distinguish between entry removals and cache entry create/modify-update events because they can query what the new entry's value via `ReadEntryView.find()` method.

Adding a write listener is done via the WriteListeners interface which is accessible via both `ReadWriteMap.listeners()` and `WriteOnlyMap.listeners()` method.

A write listener implementation can be defined either passing a function to `onWrite(Consumer<ReadEntryView<K, V>>)` method, or passing a WriteListener implementation to `add(WriteListener<K, V>)` method. Either way, all these methods return an AutoCloseable instance that can be used to de-register the function listener:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.api.functional.Listeners.WriteListeners.WriteListener;

WriteOnlyMap<String, String> woMap = ...

AutoCloseable writeFunctionCloseHandler = woMap.listeners().onWrite(written -> {
   // `written` is a ReadEntryView of the written entry
   System.out.printf("Written: %s%n", written.get());
});
AutoCloseable writeCloseHanlder = woMap.listeners().add(new WriteListener<String,
String>() {
   @Override
   public void onWrite(ReadEntryView<K, V> written) {
      System.out.printf("Written: %s%n", written.get());
   }
});

// Either wrap handler in a try section to have it auto close...
try(writeFunctionCloseHandler) {
   // Write entries using read-write or write-only functional map API
   ...
}
// Or close manually
writeCloseHanlder.close();
```

2.9.2. Read-Write Listeners

Read-write listeners enable users to register listeners for cache entry created, modified and removed events, and also register listeners for any cache entry write events.

Entry created, modified and removed events can only be fired when these originate on a read-write functional map, since this is the only one that guarantees that the previous value has been read, and hence the differentiation between create, modified and removed can be fully guaranteed.

Adding a read-write listener is done via the `ReadWriteListeners` interface which is accessible via `ReadWriteMap.listeners()` method.

If interested in only one of the event types, the simplest way to add a listener is to pass a function to either `onCreate` , `onModify` or `onRemove` methods. All these methods return an AutoCloseable instance that can be used to de-register the function listener:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

ReadWriteMap<String, String> rwMap = ...
AutoCloseable createClose = rwMap.listeners().onCreate(created -> {
   // `created` is a ReadEntryView of the created entry
   System.out.printf("Created: %s%n", created.get());
});
AutoCloseable modifyClose = rwMap.listeners().onModify((before, after) -> {
   // `before` is a ReadEntryView of the entry before update
   // `after` is a ReadEntryView of the entry after update
   System.out.printf("Before: %s%n", before.get());
   System.out.printf("After: %s%n", after.get());
});
AutoCloseable removeClose = rwMap.listeners().onRemove(removed -> {
   // `removed` is a ReadEntryView of the removed entry
   System.out.printf("Removed: %s%n", removed.get());
});
AutoCloseable writeClose = woMap.listeners().onWrite(written -> {
   // `written` is a ReadEntryView of the written entry
   System.out.printf("Written: %s%n", written.get());
});
...
// Either wrap handler in a try section to have it auto close...
try(createClose) {
   // Create entries using read-write functional map API
   ...
}
// Or close manually
modifyClose.close();
```

If listening for two or more event types, it's better to pass in an implementation of `ReadWriteListener` interface via the `ReadWriteListeners.add()` method. `ReadWriteListener` offers the same `onCreate`/`onModify`/`onRemove` callbacks with default method implementations that are empty:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import
org.infinispan.commons.api.functional.Listeners.ReadWriteListeners.ReadWriteListener;

ReadWriteMap<String, String> rwMap = ...
AutoCloseable readWriteClose = rwMap.listeners.add(new ReadWriteListener<String,
String>() {
   @Override
   public void onCreate(ReadEntryView<String, String> created) {
      System.out.printf("Created: %s%n", created.get());
   }

   @Override
   public void onModify(ReadEntryView<String, String> before, ReadEntryView<String,
String> after) {
      System.out.printf("Before: %s%n", before.get());
      System.out.printf("After: %s%n", after.get());
   }

   @Override
   public void onRemove(ReadEntryView<String, String> removed) {
      System.out.printf("Removed: %s%n", removed.get());
   }
);
AutoCloseable writeClose = rwMap.listeners.add(new WriteListener<String, String>() {
   @Override
   public void onWrite(ReadEntryView<K, V> written) {
      System.out.printf("Written: %s%n", written.get());
   }
);

// Either wrap handler in a try section to have it auto close...
try(readWriteClose) {
   // Create/update/remove entries using read-write functional map API
   ...
}
// Or close manually
writeClose.close();
```

2.10. Marshalling of Functions

Running functional map in a cluster of nodes involves marshalling and replication of the operation parameters under certain circumstances.

To be more precise, when write operations are executed in a cluster, regardless of read-write or write-only operations, all the parameters to the method and the functions are replicated to other nodes.

There are multiple ways in which a function can be marshalled. The simplest way, which is also the most costly option in terms of payload size, is to mark the function as `Serializable` :

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;

WriteOnlyMap<String, String> writeOnlyMap = ...

// Force a function to be Serializable
Consumer<WriteEntryView<String>> function =
   (Consumer<WriteEntryView<String>> & Serializable) wv -> wv.set("one");

CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", function);
```

{brandname} provides overloads for all functional methods that make lambdas passed directly to the API serializable by default; the compiler automatically selects this overload if that's possible. Therefore you can call

```
WriteOnlyMap<String, String> writeOnlyMap = ...
CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", wv -> wv.set("one"));
```

without doing the cast described above.

A more economical way to marshall a function is to provide an {brandname} `Externalizer` for it:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.marshall.Externalizer;
import org.infinispan.commons.marshall.SerializeFunctionWith;

WriteOnlyMap<String, String> writeOnlyMap = ...

// Force a function to be Serializable
Consumer<WriteEntryView<String>> function = new SetStringConstant<>();
CompletableFuture<Void> writeFuture = writeOnlyMap.eval("key1", function);

@SerializeFunctionWith(value = SetStringConstant.Externalizer0.class)
class SetStringConstant implements Consumer<WriteEntryView<String>> {
   @Override
   public void accept(WriteEntryView<String> view) {
      view.set("value1");
   }

   public static final class Externalizer0 implements Externalizer<Object> {
      public void writeObject(ObjectOutput oo, Object o) {
         // No-op
      }
      public Object readObject(ObjectInput input) {
         return new SetStringConstant<>();
      }
   }
}
```

To help users take advantage of the tiny payloads generated by `Externalizer`-based functions, the functional API comes with a helper class called `org.infinispan.commons.marshall.MarshallableFunctions` which provides marshallable functions for some of the most commonly user functions.

In fact, all the functions required to implement `ConcurrentMap` and `JCache` using the functional map API have been defined in `MarshallableFunctions`. For example, here is an implementation of JCache's `boolean putIfAbsent(K, V)` using functional map API which can be run in a cluster:

```
import org.infinispan.commons.api.functional.EntryView.*;
import org.infinispan.commons.api.functional.FunctionalMap.*;
import org.infinispan.commons.marshall.MarshallableFunctions;

ReadWriteMap<String, String> readWriteMap = ...

CompletableFuture<Boolean> future = readWriteMap.eval("key1,
   MarshallableFunctions.setValueIfAbsentReturnBoolean());
future.thenAccept(stored -> System.out.printf("Value was put? %s%n", stored));
```

2.11. Use Cases for Functional API

This new API is meant to complement existing Key/Value {brandname} API offerings, so you'll still be able to use `ConcurrentMap` or `JCache` standard APIs if that's what suits your use case best.

The target audience for this new API is either:

- Distributed or persistent caching/inmemorydatagrid users that want to benefit from CompletableFuture and/or Traversable for async/lazy data grid or caching data manipulation. The clear advantage here is that threads do not need to be idle waiting for remote operations to complete, but instead these can be notified when remote operations complete and then chain them with other subsequent operations.
- Users who want to go beyond the standard operations exposed by `ConcurrentMap` and `JCache`, for example, if you want to do a replace operation using metadata parameter equality instead of value equality, or if you want to retrieve metadata information from values and so on.

Chapter 3. Encoding

Encoding is the data conversion operation done by {brandname} caches before storing data, and when reading back from storage.

3.1. Overview

Encoding allows dealing with a certain data format during API calls (map, listeners, stream, etc) while the format effectively stored is different.

The data conversions are handled by instances of *org.infinispan.commons.dataconversion.Encoder* :

```
public interface Encoder {

   /**
    * Convert data in the read/write format to the storage format.
    *
    * @param content data to be converted, never null.
    * @return Object in the storage format.
    */
   Object toStorage(Object content);

   /**
    * Convert from storage format to the read/write format.
    *
    * @param content data as stored in the cache, never null.
    * @return data in the read/write format
    */
   Object fromStorage(Object content);

   /**
     * Returns the {@link MediaType} produced by this encoder or null if the storage
format is not known.
     */
   MediaType getStorageFormat();
}
```

3.2. Default encoders

{brandname} automatically picks the Encoder depending on the cache configuration. The table below shows which internal Encoder is used for several configurations:

Mode	Configuration	Encoder	Description
Embedded/Server	Default	IdentityEncoder	Passthrough encoder, no conversion done

Mode	Configuration	Encoder	Description
Embedded	StorageType.OFF_HEAP	GlobalMarshallerEncoder	Use the {brandname} internal marshaller to convert to byte[]. May delegate to the configured marshaller in the cache manager.
Embedded	StorageType.BINARY	BinaryEncoder	Use the {brandname} internal marshaller to convert to byte[], except for primitives and String.
Server	StorageType.OFF_HEAP	IdentityEncoder	Store byte[]s directly as received by remote clients

3.3. Overriding programmatically

Is is possible to override programmatically the encoding used for both keys and values, by calling the *.withEncoding()* method variants from *AdvancedCache*.

Example, consider the following cache configured as OFF_HEAP:

```
// Read and write POJO, storage will be byte[] since for
// OFF_HEAP the GlobalMarshallerEncoder is used internally:
cache.put(1, new Pojo())
Pojo value = cache.get(1)

// Get the content in its stored format by overriding
// the internal encoder with a no-op encoder (IdentityEncoder)
Cache<?,?> rawContent = cache.getAdvancedCache().withValueEncoding(IdentityEncoder
.class)
byte[] marshalled = rawContent.get(1)
```

The override can be useful if any operation in the cache does not require decoding, such as counting number of entries, or calculating the size of byte[] of an OFF_HEAP cache.

3.4. Defining custom Encoders

A custom encoder can be registered in the *EncoderRegistry*.

Ensure that the registration is done in every node of the cluster, before starting the caches.

Consider a custom encoder used to compress/decompress with gzip:

```
```

```
public class GzipEncoder implements Encoder {

   @Override
   public Object toStorage(Object content) {
      assert content instanceof String;
      return compress(content.toString());
   }

   @Override
   public Object fromStorage(Object content) {
      assert content instanceof byte[];
      return decompress((byte[]) content);
   }

   private byte[] compress(String str) {
      try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
           GZIPOutputStream gis = new GZIPOutputStream(baos)) {
         gis.write(str.getBytes("UTF-8"));
         gis.close();
         return baos.toByteArray();
      } catch (IOException e) {
         throw new RuntimeException("Unabled to compress", e);
      }
   }

   private String decompress(byte[] compressed) {
      try (GZIPInputStream gis = new GZIPInputStream(new ByteArrayInputStream
(compressed));
           BufferedReader bf = new BufferedReader(new InputStreamReader(gis, "UTF-8")
)) {
         StringBuilder result = new StringBuilder();
         String line;
         while ((line = bf.readLine()) != null) {
            result.append(line);
         }
         return result.toString();
      } catch (IOException e) {
         throw new RuntimeException("Unable to decompress", e);
      }
   }

   @Override
   public MediaType getStorageFormat() {
      return MediaType.parse("application/gzip");
   }

   @Override
   public boolean isStorageFormatFilterable() {
      return false;
   }
```

```
   @Override
   public short id() {
      return 10000;
   }
}
```

It can be registered by:

```
GlobalComponentRegistry registry = cacheManager.getGlobalComponentRegistry();
EncoderRegistry encoderRegistry = registry.getComponent(EncoderRegistry.class);
encoderRegistry.registerEncoder(new GzipEncoder());
```

And then be used to write and read data from a cache:

```
AdvancedCache<String, String> cache = ...

// Decorate cache with the newly registered encoder, without encoding keys
(IdentityEncoder)
// but compressing values
AdvancedCache<String, String> compressingCache = (AdvancedCache<String, String>)
cache.withEncoding(IdentityEncoder.class, GzipEncoder.class);

// All values will be stored compressed...
compressingCache.put("297931749", "0412c789a37f5086f743255cfa693dd5");

// ... but API calls deals with String
String value = compressingCache.get("297931749");

// Bypassing the value encoder to obtain the value as it is stored
Object value = compressingCache.withEncoding(IdentityEncoder.class).get("297931749");

// value is a byte[] which is the compressed value
```

3.5. MediaType

A Cache can optionally be configured with a `org.infinispan.commons.dataconversion.MediaType` for keys and values. By describing the data format of the cache, {brandname} is able to convert data on the fly during cache operations.

The MediaType configuration is more suitable when storing binary data. When using server mode, it's common to have a MediaType configured and clients such as REST or Hot Rod reading and writing in different formats.

The data conversion between MediaType formats are handled by instances of `org.infinispan.commons.dataconversion.Transcoder`

```
public interface Transcoder {

   /**
    * Transcodes content between two different {@link MediaType}.
    *
    * @param content         Content to transcode.
    * @param contentType     The {@link MediaType} of the content.
    * @param destinationType The target {@link MediaType} to convert.
    * @return the transcoded content.
    */
   Object transcode(Object content, MediaType contentType, MediaType destinationType);

   /**
    * @return all the {@link MediaType} handled by this Transcoder.
    */
   Set<MediaType> getSupportedMediaTypes();
}
```

3.5.1. Configuration

Declarative:

```
<cache>
   <encoding>
      <key media-type="application/x-java-object; type=java.lang.Integer"/>
      <value media-type="application/xml; charset=UTF-8"/>
   </encoding>
</cache>
```

Programmatic:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();

cfg.encoding().key().mediaType("text/plain");
cfg.encoding().value().mediaType("application/json");
```

3.5.2. Overriding the MediaType Programmatically

It's possible to decorate the Cache with a different MediaType, allowing cache operations to be executed sending and receiving different data formats.

Example:

```
DefaultCacheManager cacheManager = new DefaultCacheManager();

// The cache will store POJO for keys and values
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-java-object");
cfg.encoding().value().mediaType("application/x-java-object");

cacheManager.defineConfiguration("mycache", cfg.build());

Cache<Integer, Person> cache = cacheManager.getCache("mycache");

cache.put(1, new Person("John","Doe"));

// Wraps cache using 'application/x-java-object' for keys but JSON for values
Cache<Integer, byte[]> jsonValuesCache = (Cache<Integer, byte[]>) cache
.getAdvancedCache().withMediaType("application/x-java-object", "application/json");

byte[] json = jsonValuesCache.get(1);
```

Will return the value in JSON format:

```
{
   "_type":"org.infinispan.sample.Person",
   "name":"John",
   "surname":"Doe"
}
```

Most Transcoders are installed when server mode is used; when using library mode, an extra dependency, *org.infinispan:infinispan-server-core* should be added to the project.

3.5.3. Transcoders and Encoders

Usually there will be none or only one data conversion involved in a cache operation:

- No conversion by default on caches using in embedded or server mode;
- *Encoder* based conversion for embedded caches without MediaType configured, but using OFF_HEAP or BINARY;
- *Transcoder* based conversion for caches used in server mode with multiple REST and Hot Rod clients sending and receiving data in different formats. Those caches will have MediaType configured describing the storage.

But it's possible to have both encoders and transcoders being used simultaneously for advanced use cases.

Consider an example, a cache that stores marshalled objects (with jboss marshaller) content but for security reasons a transparent encryption layer should be added in order to avoid storing "plain"

data to an external store. Clients should be able to read and write data in multiple formats.

This can be achieved by configuring the cache with the the MediaType that describes the storage regardless of the encoding layer:

```
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
cfg.encoding().key().mediaType("application/x-jboss-marshalling");
```

The transparent encryption can be added by decorating the cache with a special *Encoder* that encrypts/decrypts with storing/retrieving, for example:

```
public class Scrambler implements Encoder {

   Object toStorage(Object content) {
      // Encrypt data
   }

   Object fromStorage(Object content) {
      // Decrypt data
   }

   MediaType getStorageFormat() {
      return "application/scrambled";
   }

}
```

To make sure all data written to the cache will be stored encrypted, it's necessary to decorate the cache with the Encoder above and perform all cache operations in this decorated cache:

```
Cache<?,?> secureStorageCache = cache.getAdvancedCache().withEncoding(Scrambler.class
).put(k,v);
```

The capability of reading data in multiple formats can be added by decorating the cache with the desired MediaType:

```
// Obtain a stream of values in XML format from the secure cache
secureStorageCache.getAdvancedCache().withMediaType("application/xml","application/xml
").values().stream();
```

Internally, {brandname} will first apply the encoder *fromStorage* operation to obtain the entries, that will be in "application/x-jboss-marshalling" format and then apply a successive conversion to "application/xml" by using the adequate Transcoder.

Chapter 4. The Embedded CacheManager

The CacheManager is {brandname}'s main entry point. You use a CacheManager to

- configure and obtain caches
- manage and monitor your nodes
- execute code across a cluster
- more…

Depending on whether you are embedding {brandname} in your application or you are using it remotely, you will be dealing with either an `EmbeddedCacheManager` or a `RemoteCacheManager`. While they share some methods and properties, be aware that there are semantic differences between them. The following chapters focus mostly on the *embedded* implementation.

CacheManagers are heavyweight objects, and we foresee no more than one CacheManager being used per JVM (unless specific setups require more than one; but either way, this would be a minimal and finite number of instances).

The simplest way to create a CacheManager is:

```
EmbeddedCacheManager manager = new DefaultCacheManager();
```

which starts the most basic, local mode, non-clustered cache manager with no caches. CacheManagers have a lifecycle and the default constructors also call start(). Overloaded versions of the constructors are available, that do not start the CacheManager, although keep in mind that CacheManagers need to be started before they can be used to create Cache instances.

Once constructed, CacheManagers should be made available to any component that require to interact with it via some form of application-wide scope such as JNDI, a ServletContext or via some other mechanism such as an IoC container.

When you are done with a CacheManager, you must stop it so that it can release its resources:

```
manager.stop();
```

This will ensure all caches within its scope are properly stopped, thread pools are shutdown. If the CacheManager was clustered it will also leave the cluster gracefully.

4.1. Obtaining caches

After you configure the `CacheManager`, you can obtain and control caches.

Invoke the `getCache()` method to obtain caches, as follows:

```
Cache<String, String> myCache = manager.getCache("myCache");
```

The preceding operation creates a cache named `myCache`, if it does not already exist, and returns it.

Using the `getCache()` method creates the cache only on the node where you invoke the method. In other words, it performs a local operation that must be invoked on each node across the cluster. Typically, applications deployed across multiple nodes obtain caches during initialization to ensure that caches are *symmetric* and exist on each node.

Invoke the `createCache()` method to create caches dynamically across the entire cluster, as follows:

```
Cache<String, String> myCache = manager.administration().createCache("myCache",
"myTemplate");
```

The preceding operation also automatically creates caches on any nodes that subsequently join the cluster.

Caches that you create with the `createCache()` method are ephemeral by default. If the entire cluster shuts down, the cache is not automatically created again when it restarts.

Use the PERMANENT flag to ensure that caches can survive restarts, as follows:

```
Cache<String, String> myCache = manager.administration().withFlags(AdminFlag.
PERMANENT).createCache("myCache", "myTemplate");
```

For the PERMANENT flag to take effect, you must enable global state and set a configuration storage provider.

For more information about configuration storage providers, see GlobalStateConfigurationBuilder#configurationStorage().

4.2. Clustering Information

The `EmbeddedCacheManager` has quite a few methods to provide information as to how the cluster is operating. The following methods only really make sense when being used in a clustered environment (that is when a Transport is configured).

4.3. Member Information

When you are using a cluster it is very important to be able to find information about membership in the cluster including who is the owner of the cluster.

getMembers()

The getMembers() method returns all of the nodes in the current cluster.

getCoordinator()

The getCoordinator() method will tell you which one of the members is the coordinator of the cluster. For most intents you shouldn't need to care who the coordinator is. You can use isCoordinator() method directly to see if the local node is the coordinator as well.

4.4. Other methods

getTransport()

This method provides you access to the underlying Transport that is used to send messages to other nodes. In most cases a user wouldn't ever need to go to this level, but if you want to get Transport specific information (in this case JGroups) you can use this mechanism.

getStats()

The stats provided here are coalesced from all of the active caches in this manager. These stats can be useful to see if there is something wrong going on with your cluster overall.

Chapter 5. Locking and Concurrency

{brandname} makes use of multi-versioned concurrency control (MVCC) - a concurrency scheme popular with relational databases and other data stores. MVCC offers many advantages over coarse-grained Java synchronization and even JDK Locks for access to shared data, including:

- allowing concurrent readers and writers
- readers and writers do not block one another
- write skews can be detected and handled
- internal locks can be striped

5.1. Locking implementation details

{brandname}'s MVCC implementation makes use of minimal locks and synchronizations, leaning heavily towards lock-free techniques such as compare-and-swap and lock-free data structures wherever possible, which helps optimize for multi-CPU and multi-core environments.

In particular, {brandname}'s MVCC implementation is heavily optimized for readers. Reader threads do not acquire explicit locks for entries, and instead directly read the entry in question.

Writers, on the other hand, need to acquire a write lock. This ensures only one concurrent writer per entry, causing concurrent writers to queue up to change an entry. To allow concurrent reads, writers make a copy of the entry they intend to modify, by wrapping the entry in an `MVCCEntry`. This copy isolates concurrent readers from seeing partially modified state. Once a write has completed, `MVCCEntry.commit()` will flush changes to the data container and subsequent readers will see the changes written.

5.1.1. How does it work in clustered caches?

In clustered caches, each key has a node responsible to lock the key. This node is called primary owner.

Non Transactional caches

1. The write operation is sent to the primary owner of the key.
2. The primary owner tries to lock the key.
   a. If it succeeds, it forwards the operation to the other owners;
   b. Otherwise, an exception is thrown.

If the operation is conditional and it fails on the primary owner, it is not forwarded to the other owners.

If the operation is executed locally in the primary owner, the first step is skipped.

5.1.2. Transactional caches

The transactional cache supports optimistic and pessimistic locking mode. Refer to Transaction Locking for more information.

5.1.3. Isolation levels

Isolation level affects what transactions can read when running concurrently with other transaction. Refer to Isolation Levels for more information.

5.1.4. The LockManager

The `LockManager` is a component that is responsible for locking an entry for writing. The `LockManager` makes use of a `LockContainer` to locate/hold/create locks. `LockContainers` come in two broad flavours, with support for lock striping and with support for one lock per entry.

5.1.5. Lock striping

Lock striping entails the use of a fixed-size, shared collection of locks for the entire cache, with locks being allocated to entries based on the entry's key's hash code. Similar to the way the JDK's `ConcurrentHashMap` allocates locks, this allows for a highly scalable, fixed-overhead locking mechanism in exchange for potentially unrelated entries being blocked by the same lock.

The alternative is to disable lock striping - which would mean a *new* lock is created per entry. This approach *may* give you greater concurrent throughput, but it will be at the cost of additional memory usage, garbage collection churn, etc.

Default lock striping settings

lock striping is disabled by default, due to potential deadlocks that can happen if locks for different keys end up in the same lock stripe.

The size of the shared lock collection used by lock striping can be tuned using the `concurrencyLevel` attribute of the `<locking /> configuration element.

Configuration example:

```
<locking striping="false|true"/>
```

Or

```
new ConfigurationBuilder().locking().useLockStriping(false|true);
```

5.1.6. Concurrency levels

In addition to determining the size of the striped lock container, this concurrency level is also used to tune any JDK `ConcurrentHashMap` based collections where related, such as internal to `DataContainer`s. Please refer to the JDK `ConcurrentHashMap` Javadocs for a detailed discussion of

concurrency levels, as this parameter is used in exactly the same way in {brandname}.

Configuration example:

```
<locking concurrency-level="32"/>
```

Or

```
new ConfigurationBuilder().locking().concurrencyLevel(32);
```

5.1.7. Lock timeout

The lock timeout specifies the amount of time, in milliseconds, to wait for a contented lock.

Configuration example:

```
<locking acquire-timeout="10000"/>
```

Or

```
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10000);
//alternatively
new ConfigurationBuilder().locking().lockAcquisitionTimeout(10, TimeUnit.SECONDS);
```

5.1.8. Consistency

The fact that a single owner is locked (as opposed to all owners being locked) does not break the following consistency guarantee: if key `K` is hashed to nodes `{A, B}` and transaction `TX1` acquires a lock for `K`, let's say on `A`. If another transaction, `TX2`, is started on `B` (or any other node) and `TX2` tries to lock `K` then it will fail with a timeout as the lock is already held by `TX1`. The reason for this is the that the lock for a key `K` is always, deterministically, acquired on the same node of the cluster, regardless of where the transaction originates.

5.2. Data Versioning

{brandname} supports two forms of data versioning: simple and external. The simple versioning is used in transactional caches for write skew check. Refer to Write Skews for more information.

The external versioning is used to encapsulate an external source of data versioning within {brandname}, such as when using {brandname} with Hibernate which in turn gets its data version information directly from a database.

In this scheme, a mechanism to pass in the version becomes necessary, and overloaded versions of `put()` and `putForExternalRead()` will be provided in `AdvancedCache` to take in an external data version. This is then stored on the `InvocationContext` and applied to the entry at commit time.

Write skew checks cannot and will not be performed in the case of external data versioning.

Chapter 6. Clustered Lock

A *clustered lock* is a lock which is distributed and shared among all nodes in the {brandname} cluster and currently provides a way to execute code that will be synchronized between the nodes in a given cluster.

6.1. Installation

In order to start using the clustered locks, you needs to add the dependency in your Maven `pom.xml` file:

=om.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-lock</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

6.2. ClusteredLock Configuration

Currently there is a single type of `ClusteredLock` supported : non reentrant, NODE ownership lock.

6.2.1. Ownership

- `NODE` When a `ClusteredLock` is defined, this lock can be used from all the nodes in the {brandname} cluster. When the ownership is NODE type, this means that the owner of the lock is the {brandname} node that acquired the lock at a given time. This means that each time we get a `ClusteredLock` instance with the `ClusteredCacheManager`, this instance will be the same instance for each {brandname} node. This lock can be used to synchronize code between {brandname} nodes. The advantage of this lock is that any thread in the node can release the lock at a given time.
- `INSTANCE` - not yet supported

When a `ClusteredLock` is defined, this lock can be used from all the nodes in the {brandname} cluster. When the ownership is INSTANCE type, this means that the owner of the lock is the actual instance we acquired when `ClusteredLockManager.get("lockName")` is called.

This means that each time we get a `ClusteredLock` instance with the `ClusteredCacheManager`, this instance will be a new instance. This lock can be used to synchronize code between {brandname} nodes and inside each {brandname} node. The advantage of this lock is that only the instance that called 'lock' can release the lock.

6.2.2. Reentrancy

When a `ClusteredLock` is configured reentrant, the owner of the lock can reacquire the lock as many consecutive times as it wants while holding the lock.

Currently, only non reentrant locks are supported. This means that when two consecutive `lock` calls are sent for the same owner, the first call will acquire the lock if it's available, and the second call will block.

6.3. ClusteredLockManager Interface

The `ClusteredLockManager` interface, **marked as experimental**, is the entry point to define, retrieve and remove a lock. It automatically listen to the creation of `EmbeddedCacheManager` and proceeds with the registration of an instance of it per `EmbeddedCacheManager`. It starts the internal caches needed to store the lock state.

Retrieving the `ClusteredLockManager` is as simple as invoking the `EmbeddedClusteredLockManagerFactory.from(EmbeddedCacheManager)` as shown in the example below:

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager manager = ...;

// retrieve the ClusteredLockManager
ClusteredLockManager clusteredLockManager = EmbeddedClusteredLockManagerFactory.from
(manager);
```

```
@Experimental
public interface ClusteredLockManager {

   boolean defineLock(String name);

   boolean defineLock(String name, ClusteredLockConfiguration configuration);

   ClusteredLock get(String name);

   ClusteredLockConfiguration getConfiguration(String name);

   boolean isDefined(String name);

   CompletableFuture<Boolean> remove(String name);

   CompletableFuture<Boolean> forceRelease(String name);
}
```

- `defineLock` : Defines a lock with the specified name and the default `ClusteredLockConfiguration`. It does not overwrite existing configurations.
- `defineLock(String name, ClusteredLockConfiguration configuration)` : Defines a lock with the

specified name and `ClusteredLockConfiguration`. It does not overwrite existing configurations.

- `ClusteredLock get(String name)` : Get's a `ClusteredLock` by it's name. A call of `defineLock` must be done at least once in the cluster. See ownership level section to understand the implications of `get` method call.

Currently, the only ownership level supported is **NODE**.

- `ClusteredLockConfiguration getConfiguration(String name)` :

Returns the configuration of a `ClusteredLock`, if such exists.

- `boolean isDefined(String name)` : Checks if a lock is already defined.
- `CompletableFuture<Boolean> remove(String name)` : Removes a `ClusteredLock` if such exists.
- `CompletableFuture<Boolean> forceRelease(String name)` : Releases - or unlocks - a `ClusteredLock`, if such exists, no matter who is holding it at a given time. Calling this method may cause concurrency issues and has to be used in **exceptional situations**.

6.4. ClusteredLock Interface

`ClusteredLock` interface, **marked as experimental**, is the interface that implements the clustered locks.

```
@Experimental
public interface ClusteredLock {

   CompletableFuture<Void> lock();

   CompletableFuture<Boolean> tryLock();

   CompletableFuture<Boolean> tryLock(long time, TimeUnit unit);

   CompletableFuture<Void> unlock();

   CompletableFuture<Boolean> isLocked();

   CompletableFuture<Boolean> isLockedByMe();
}
```

- `lock` : Acquires the lock. If the lock is not available then call blocks until the lock is acquired. Currently, **there is no maximum time specified for a lock request to fail**, so this could cause thread starvation.
- `tryLock` Acquires the lock only if it is free at the time of invocation, and returns `true` in that case. This method does not block (or wait) for any lock acquisition.
- `tryLock(long time, TimeUnit unit)` If the lock is available this method returns immediately with `true`. If the lock is not available then the call waits until :
  - The lock is acquired

- The specified waiting time elapses

If the time is less than or equal to zero, the method will not wait at all.

- `unlock`

Releases the lock. Only the holder of the lock may release the lock.

- `isLocked` Returns `true` when the lock is locked and `false` when the lock is released.
- `isLockedByMe` Returns `true` when the lock is owned by the caller and `false` when the lock is owned by someone else or it's released.

6.4.1. Usage Examples

```
EmbeddedCache cm = ...;
ClusteredLockManager cclm = EmbeddedClusteredLockManagerFactory.from(cm);

lock.tryLock()
  .thenCompose(result -> {
     if (result) {
      try {
          // manipulate protected state
          } finally {
             return lock.unlock();
          }
     } else {
        // Do something else
     }
  });
}
```

6.4.2. ClusteredLockManager Configuration

You can configure `ClusteredLockManager` to use different strategies for locks, either declaratively or programmatically, with the following attributes:

`num-owners`
: Defines the total number of nodes in each cluster that store the states of clustered locks. The default value is `-1`, which replicates the value to all nodes.

`reliability`
: Controls how clustered locks behave when clusters split into partitions or multiple nodes leave a cluster. You can set the following values:

  - `AVAILABLE`: Nodes in any partition can concurrently operate on locks.
  - `CONSISTENT`: Only nodes that belong to the majority partition can operate on locks. This is the default value.

The following is an example declarative configuration for `ClusteredLockManager`:

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="urn:infinispan:config:${infinispan.core.schema.version}
http://www.infinispan.org/schemas/infinispan-config-10.0.xsd"
        xmlns="urn:infinispan:config:10.0">
    ...
    <cache-container default-cache="default">
        <transport/>
        <local-cache name="default">
            <locking concurrency-level="100" acquire-timeout="1000"/>
        </local-cache>

        <clustered-locks xmlns="urn:infinispan:config:clustered-locks:10.0"
                         num-owners = "3"
                         reliability="AVAILABLE">
            <clustered-lock name="lock1" />
            <clustered-lock name="lock2" />
        </clustered-locks>
    </cache-container>
    ...
</infinispan>
```

Chapter 7. Clustered Counters

Clustered counters are counters which are distributed and shared among all nodes in the {brandname} cluster. Counters can have different consistency levels: strong and weak.

Although a strong/weak consistent counter has separate interfaces, both support updating its value, return the current value and they provide events when its value is updated. Details are provided below in this document to help you choose which one fits best your uses-case.

7.1. Installation and Configuration

In order to start using the counters, you needs to add the dependency in your Maven `pom.xml` file:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-clustered-counter</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

The counters can be configured {brandname} configuration file or on-demand via the `CounterManager` interface detailed later in this document. A counters configured in {brandname} configuration file is created at boot time when the `EmbeddedCacheManager` is starting. Theses counters are started eagerly and they are available in all the cluster's nodes.

configuration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan>
    <cache-container ...>
        <!-- if needed to persist counter, global state needs to be configured -->
        <global-state>
            ...
        </global-state>
        <!-- your caches configuration goes here -->
         <counters xmlns="urn:infinispan:config:counters:9.2" num-owners="3"
reliability="CONSISTENT">
            <strong-counter name="c1" initial-value="1" storage="PERSISTENT"/>
            <strong-counter name="c2" initial-value="2" storage="VOLATILE">
                <lower-bound value="0"/>
            </strong-counter>
            <strong-counter name="c3" initial-value="3" storage="PERSISTENT">
                <upper-bound value="5"/>
            </strong-counter>
            <strong-counter name="c4" initial-value="4" storage="VOLATILE">
                <lower-bound value="0"/>
                <upper-bound value="10"/>
            </strong-counter>
            <weak-counter name="c5" initial-value="5" storage="PERSISTENT"
concurrency-level="1"/>
         </counters>
    </cache-container>
</infinispan>
```

or programmatically, in the `GlobalConfigurationBuilder`:

```
GlobalConfigurationBuilder globalConfigurationBuilder = ...;
CounterManagerConfigurationBuilder builder = globalConfigurationBuilder.addModule
(CounterManagerConfigurationBuilder.class);
builder.numOwner(3).reliability(Reliability.CONSISTENT);
builder.addStrongCounter().name("c1").initialValue(1).storage(Storage.PERSISTENT);
builder.addStrongCounter().name("c2").initialValue(2).lowerBound(0).storage(Storage.VO
LATILE);
builder.addStrongCounter().name("c3").initialValue(3).upperBound(5).storage(Storage.PE
RSISTENT);
builder.addStrongCounter().name("c4").initialValue(4).lowerBound(0).upperBound(10).sto
rage(Storage.VOLATILE);
builder.addWeakCounter().name("c5").initialValue(5).concurrencyLevel(1).storage(Storag
e.PERSISTENT);
```

On other hand, the counters can be configured on-demand, at any time after the `EmbeddedCacheManager` is initialized.

```
CounterManager manager = ...;
manager.defineCounter("c1", CounterConfiguration.builder(CounterType.UNBOUNDED_STRONG
).initialValue(1).storage(Storage.PERSISTENT)build());
manager.defineCounter("c2", CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
.initialValue(2).lowerBound(0).storage(Storage.VOLATILE).build());
manager.defineCounter("c3", CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
.initialValue(3).upperBound(5).storage(Storage.PERSISTENT).build());
manager.defineCounter("c4", CounterConfiguration.builder(CounterType.BOUNDED_STRONG)
.initialValue(4).lowerBound(0).upperBound(10).storage(Storage.VOLATILE).build());
manager.defineCounter("c2", CounterConfiguration.builder(CounterType.WEAK)
.initialValue(5).concurrencyLevel(1).storage(Storage.PERSISTENT).build());
```

`CounterConfiguration` is immutable and can be reused.

The method `defineCounter()` will return `true` if the counter is successful configured or `false` otherwise. However, if the configuration is invalid, the method will throw a `CounterConfigurationException`. To find out if a counter is already defined, use the method `isDefined()`.

```
CounterManager manager = ...
if (!manager.isDefined("someCounter")) {
    manager.define("someCounter", ...);
}
```

Per cluster attributes:

- `num-owners`: Sets the number of counter's copies to keep cluster-wide. A smaller number will make update operations faster but will support a lower number of server crashes. It **must be positive** and its default value is `2`.
- `reliability`: Sets the counter's update behavior in a network partition. Default value is `AVAILABLE` and valid values are:
  - `AVAILABLE`: all partitions are able to read and update the counter's value.
  - `CONSISTENT`: only the primary partition (majority of nodes) will be able to read and update the counter's value. The remaining partitions can only read its value.

Per counter attributes:

- `initial-value` [common]: Sets the counter's initial value. Default is `0` (zero).
- `storage` [common]: Sets the counter's behavior when the cluster is shutdown and restarted. Default value is `VOLATILE` and valid values are:
  - `VOLATILE`: the counter's value is only available in memory. The value will be lost when a cluster is shutdown.
  - `PERSISTENT`: the counter's value is stored in a private and local persistent store. The value is kept when the cluster is shutdown and restored after a restart.

On-demand and `VOLATILE` counters will lose its value and configuration after a cluster shutdown. They must be defined again after the restart.

- `lower-bound` [strong]: Sets the strong consistent counter's lower bound. Default value is `Long.MIN_VALUE`.
- `upper-bound` [strong]: Sets the strong consistent counter's upper bound. Default value is `Long.MAX_VALUE`.

If neither the `lower-bound` or `upper-bound` are configured, the strong counter is set as unbounded.

The `initial-value` must be between `lower-bound` and `upper-bound` inclusive.

- `concurrency-level` [weak]: Sets the number of concurrent updates. Its value **must be positive** and the default value is `16`.

7.1.1. List counter names

To list all the counters defined, the method `CounterManager.getCounterNames()` returns a collection of all counter names created cluster-wide.

7.2. The `CounterManager` interface.

The `CounterManager` interface is the entry point to define, retrieve and remove a counter. It automatically listen to the creation of `EmbeddedCacheManager` and proceeds with the registration of an instance of it per `EmbeddedCacheManager`. It starts the caches needed to store the counter state and configures the default counters.

Retrieving the `CounterManager` is as simple as invoke the `EmbeddedCounterManagerFactory.asCounterManager(EmbeddedCacheManager)` as shown in the example below:

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager = EmbeddedCounterManagerFactory.asCounterManager(
manager);
```

For Hot Rod client, the `CounterManager` is registered in the RemoteCacheManager and it can be retrieved like:

```
// create or obtain your RemoteCacheManager
RemoteCacheManager manager = ...;

// retrieve the CounterManager
CounterManager counterManager = RemoteCounterManagerFactory.asCounterManager(manager);
```

7.2.1. Remove a counter via CounterManager

use with caution.

There is a difference between remove a counter via the `Strong/WeakCounter` interfaces and the `CounterManager`. The `CounterManager.remove(String)` removes the counter value from the cluster and removes all the listeners registered in the counter in the local counter instance. In addition, the counter instance is no longer reusable and it may return an invalid results.

On the other side, the `Strong/WeakCounter` removal only removes the counter value. The instance can still be reused and the listeners still works.

The counter is re-created if it is accessed after a removal.

7.3. The Counter

A counter can be strong (`StrongCounter`) or weakly consistent (`WeakCounter`) and both is identified by a name. They have a specific interface but they share some logic, namely, both of them are asynchronous (a `CompletableFuture` is returned by each operation), provide an update event and can be reset to its initial value.

If you don't want to use the async API, it is possible to return a synchronous counter via `sync()` method. The API is the same but without the `CompletableFuture` return value.

The following methods are common to both interfaces:

```
String getName();
CompletableFuture<Long> getValue();
CompletableFuture<Void> reset();
<T extends CounterListener> Handle<T> addListener(T listener);
CounterConfiguration getConfiguration();
CompletableFuture<Void> remove();
SyncStrongCounter sync(); //SyncWeakCounter for WeakCounter
```

- `getName()` returns the counter name (identifier).
- `getValue()` returns the current counter's value.
- `reset()` allows to reset the counter's value to its initial value.
- `addListener()` register a listener to receive update events. More details about it in the Notification and Events section.

- `getConfiguration()` returns the configuration used by the counter.
- `remove()` removes the counter value from the cluster. The instance can still be used and the listeners are kept.
- `sync()` creates a synchronous counter.

The counter is re-created if it is accessed after a removal.

7.3.1. The `StrongCounter` interface: when the consistency or bounds matters.

The strong counter provides uses a single key stored in {brandname} cache to provide the consistency needed. All the updates are performed under the key lock to updates its values. On other hand, the reads don't acquire any locks and reads the current value. Also, with this scheme, it allows to bound the counter value and provide atomic operations like compare-and-set/swap.

A `StrongCounter` can be retrieved from the `CounterManager` by using the `getStrongCounter()` method. As an example:

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getStrongCounter("my-counter);
```

Since every operation will hit a single key, the `StrongCounter` has a higher contention rate.

The `StrongCounter` interface adds the following method:

```
default CompletableFuture<Long> incrementAndGet() {
   return addAndGet(1L);
}

default CompletableFuture<Long> decrementAndGet() {
   return addAndGet(-1L);
}

CompletableFuture<Long> addAndGet(long delta);

CompletableFuture<Boolean> compareAndSet(long expect, long update);

CompletableFuture<Long> compareAndSwap(long expect, long update);
```

- `incrementAndGet()` increments the counter by one and returns the new value.
- `decrementAndGet()` decrements the counter by one and returns the new value.
- `addAndGet()` adds a delta to the counter's value and returns the new value.
- `compareAndSet()` and `compareAndSwap()` atomically set the counter's value if the current value is the expected.

A operation is considered completed when the `CompletableFuture` is completed.

The difference between compare-and-set and compare-and-swap is that the former returns true if the operation succeeds while the later returns the previous value. The compare-and-swap is successful if the return value is the same as the expected.

Bounded `StrongCounter`

When bounded, all the update method above will throw a `CounterOutOfBoundsException` when they reached the lower or upper bound. The exception has the following methods to check which side bound has been reached:

```
public boolean isUpperBoundReached();
public boolean isLowerBoundReached();
```

Uses cases

The strong counter fits better in the following uses cases:

- When counter's value is needed after each update (example, cluster-wise ids generator or sequences)
- When a bounded counter is needed (example, rate limiter)

Usage Examples

```
StrongCounter counter = counterManager.getStrongCounter("unbounded_coutner");

// incrementing the counter
System.out.println("new value is " + counter.incrementAndGet().get());

// decrement the counter's value by 100 using the functional API
counter.addAndGet(-100).thenApply(v -> {
   System.out.println("new value is " + v);
   return null;
}).get

// alternative, you can do some work while the counter is updated
CompletableFuture<Long> f = counter.addAndGet(10);
// ... do some work ...
System.out.println("new value is " + f.get());

// and then, check the current value
System.out.println("current value is " + counter.getValue().get());

// finally, reset to initial value
counter.reset().get();
System.out.println("current value is " + counter.getValue().get());

// or set to a new value if zero
System.out.println("compare and set succeeded? " + counter.compareAndSet(0, 1));
```

And below, there is another example using a bounded counter:

```
StrongCounter counter = counterManager.getStrongCounter("bounded_counter");

// incrementing the counter
try {
    System.out.println("new value is " + counter.addAndGet(100).get());
} catch (ExecutionException e) {
    Throwable cause = e.getCause();
    if (cause instanceof CounterOutOfBoundsException) {
       if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
          System.out.println("ops, upper bound reached.");
       } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
          System.out.println("ops, lower bound reached.");
       }
    }
}

// now using the functional API
counter.addAndGet(-100).handle((v, throwable) -> {
   if (throwable != null) {
      Throwable cause = throwable.getCause();
      if (cause instanceof CounterOutOfBoundsException) {
         if (((CounterOutOfBoundsException) cause).isUpperBoundReached()) {
            System.out.println("ops, upper bound reached.");
         } else if (((CounterOutOfBoundsException) cause).isLowerBoundReached()) {
            System.out.println("ops, lower bound reached.");
         }
      }
      return null;
   }
   System.out.println("new value is " + v);
   return null;
}).get();
```

Compare-and-set vs Compare-and-swap examples:

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue, newValue;
do {
   oldValue = counter.getValue().get();
   newValue = someLogic(oldValue);
} while (!counter.compareAndSet(oldValue, newValue).get());
```

With compare-and-swap, it saves one invocation counter invocation (`counter.getValue()`)

```
StrongCounter counter = counterManager.getStrongCounter("my-counter");
long oldValue = counter.getValue().get();
long currentValue, newValue;
do {
   currentValue = oldValue;
   newValue = someLogic(oldValue);
} while ((oldValue = counter.compareAndSwap(oldValue, newValue).get()) !=
currentValue);
```

7.3.2. The `WeakCounter` interface: when speed is needed

The `WeakCounter` stores the counter's value in multiple keys in {brandname} cache. The number of keys created is configured by the `concurrency-level` attribute. Each key stores a partial state of the counter's value and it can be updated concurrently. It main advantage over the `StrongCounter` is the lower contention in the cache. On other hand, the read of its value is more expensive and bounds are not allowed.

The reset operation should be handled with caution. It is **not** atomic and it produces intermediates values. These value may be seen by a read operation and by any listener registered.

A `WeakCounter` can be retrieved from the `CounterManager` by using the `getWeakCounter()` method. As an example:

```
CounterManager counterManager = ...
StrongCounter aCounter = counterManager.getWeakCounter("my-counter);
```

Weak Counter Interface

The `WeakCounter` adds the following methods:

```
default CompletableFuture<Void> increment() {
   return add(1L);
}

default CompletableFuture<Void> decrement() {
   return add(-1L);
}

CompletableFuture<Void> add(long delta);
```

They are similar to the `StrongCounter's methods but they don't return the new value.

Uses cases

The weak counter fits best in uses cases where the result of the update operation is not needed or

the counter's value is not required too often. Collecting statistics is a good example of such an use case.

Examples

Below, there is an example of the weak counter usage.

```
WeakCounter counter = counterManager.getWeakCounter("my_counter");

// increment the counter and check its result
counter.increment().get();
System.out.println("current value is " + counter.getValue().get());

CompletableFuture<Void> f = counter.add(-100);
//do some work
f.get(); //wait until finished
System.out.println("current value is " + counter.getValue().get());

//using the functional API
counter.reset().whenComplete((aVoid, throwable) -> System.out.println("Reset done " +
(throwable == null ? "successfully" : "unsuccessfully"))).get();
System.out.println("current value is " + counter.getValue().get());
```

7.4. Notifications and Events

Both strong and weak counter supports a listener to receive its updates events. The listener must implement `CounterListener` and it can be registerer by the following method:

```
<T extends CounterListener> Handle<T> addListener(T listener);
```

The `CounterLister` has the following interface:

```
public interface CounterListener {
   void onUpdate(CounterEvent entry);
}
```

The `Handle` object returned has the main goal to remove the `CounterListener` when it is not longer needed. Also, it allows to have access to the `CounterListener` instance that is it handling. It has the following interface:

```
public interface Handle<T extends CounterListener> {
   T getCounterListener();
   void remove();
}
```

Finally, the `CounterEvent` has the previous and current value and state. It has the following interface:

```
public interface CounterEvent {
   long getOldValue();
   State getOldState();
   long getNewValue();
   State getNewState();
}
```

The state is always `State.VALID` for unbounded strong counter and weak counter. `State.LOWER_BOUND_REACHED` and `State.UPPER_BOUND_REACHED` are only valid for bounded strong counters.

The weak counter `reset()` operation will trigger multiple notification with intermediate values.

Chapter 8. Protocol Interoperability

Clients exchange data with {brandname} through endpoints such as REST or Hot Rod.

Each endpoint uses a different protocol so that clients can read and write data in a suitable format. Because {brandname} can interoperate with multiple clients at the same time, it must convert data between client formats and the storage formats.

To configure {brandname} endpoint interoperability, you should define the MediaType that sets the format for data stored in the cache.

8.1. Considerations with Media Types and Endpoint Interoperability

Configuring {brandname} to store data with a specific media type affects client interoperability.

Although REST clients do support sending and receiving encoded binary data, they are better at handling text formats such as JSON, XML, or plain text.

Memcached text clients can handle String-based keys and byte[] values but cannot negotiate data types with the server. These clients do not offer much flexibility when handling data formats because of the protocol definition.

Java Hot Rod clients are suitable for handling Java objects that represent entities that reside in the cache. Java Hot Rod clients use marshalling operations to serialize and deserialize those objects into byte arrays.

Similarly, non-Java Hot Rod clients, such as the C++, C#, and Javascript clients, are suitable for handling objects in the respective languages. However, non-Java Hot Rod clients can interoperate with Java Hot Rod clients using platform independent data formats.

8.2. REST, Hot Rod, and Memcached Interoperability with Text-Based Storage

You can configure key and values with a text-based storage format.

For example, specify `text/plain; charset=UTF-8`, or any other character set, to set plain text as the media type. You can also specify a media type for other text-based formats such as JSON (`application/json`) or XML (`application/xml`) with an optional character set.

The following example configures the cache to store entries with the `text/plain; charset=UTF-8` media type:

```
<cache>
   <encoding>
      <key media-type="text/plain; charset=UTF-8"/>
      <value media-type="text/plain; charset=UTF-8"/>
   </encoding>
</cache>
```

To handle the exchange of data in a text-based format, you must configure Hot Rod clients with the `org.infinispan.commons.marshall.StringMarshaller` marshaller.

REST clients must also send the correct headers when writing and reading from the cache, as follows:

- Write: `Content-Type: text/plain; charset=UTF-8`
- Read: `Accept: text/plain; charset=UTF-8`

Memcached clients do not require any configuration to handle text-based formats.

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Memcached clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	No
Custom Java objects	No

8.3. REST, Hot Rod, and Memcached Interoperability with Custom Java Objects

If you store entries in the cache as marshalled, custom Java objects, you should configure the cache with the MediaType of the marshalled storage.

Java Hot Rod clients use the JBoss marshalling storage format as the default to store entries in the cache as custom Java objects.

The following example configures the cache to store entries with the `application/x-jboss-marshalling` media type:

```
<distributed-cache name="my-cache">
   <encoding>
      <key media-type="application/x-jboss-marshalling"/>
      <value media-type="application/x-jboss-marshalling"/>
   </encoding>
</distributed-cache>
```

If you use the Protostream marshaller, configure the MediaType as `application/x-protostream`. For UTF8Marshaller, configure the MediaType as `text/plain`.

If only Hot Rod clients interact with the cache, you do not need to configure the MediaType.

Because REST clients are most suitable for handling text formats, you should use primitives such as `java.lang.String` for keys. Otherwise, REST clients must handle keys as `bytes[]` using a supported binary encoding.

REST clients can read values for cache entries in XML or JSON format. However, the classes must be available in the server.

To read and write data from Memcached clients, you must use `java.lang.String` for keys. Values are stored and returned as marshalled objects.

Some Java Memcached clients allow data transformers that marshall and unmarshall objects. You can also configure the Memcached server module to encode responses in different formats, such as 'JSON' which is language neutral. This allows non-Java clients to interact with the data even if the storage format for the cache is Java-specific.

Storing Java objects in the cache requires you to deploy entity classes to {ProductName}. See Deploying Entity Classes.

This configuration is compatible with…	
REST clients	Yes
Java Hot Rod clients	Yes
Memcached clients	Yes
Non-Java Hot Rod clients	No
Querying and Indexing	No
Custom Java objects	Yes

8.4. Java and Non-Java Client Interoperability with Protobuf

Storing data in the cache as Protobuf encoded entries provides a platform independent configuration that enables Java and Non-Java clients to access and query the cache from any endpoint.

If indexing is configured for the cache, {brandname} automatically stores keys and values with the `application/x-protostream` media type.

If indexing is not configured for the cache, you can configure it to store entries with the `application/x-protostream` media type as follows:

```
<distributed-cache name="my-cache">
   <encoding>
      <key media-type="application/x-protostream"/>
      <value media-type="application/x-protostream"/>
   </encoding>
</distributed-cache>
```

{brandname} converts between `application/x-protostream` and `application/json`, which allows REST clients to read and write JSON formatted data. However REST clients must send the correct headers, as follows:

Read Header

```
Read: Accept: application/json
```

Write Header

```
Write: Content-Type: application/json
```

The `application/x-protostream` media type uses Protobuf encoding, which requires you to register a Protocol Buffers schema definition that describes the entities and marshallers that the clients use.

This configuration is compatible with...	
REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	Yes
Querying and Indexing	Yes
Custom Java objects	Yes

8.5. Custom Code Interoperability

You can deploy custom code with {brandname}. For example, you can deploy scripts, tasks, listeners, converters, and merge policies. Because your custom code can access data directly in the cache, it must interoperate with clients that access data in the cache through different endpoints.

For example, you might create a remote task to handle custom objects stored in the cache while other clients store data in binary format.

To handle interoperability with custom code you can either convert data on demand or store data as Plain Old Java Objects (POJOs).

8.5.1. Converting Data On Demand

If the cache is configured to store data in a binary format such as `application/x-protostream` or

`application/x-jboss-marshalling`, you can configure your deployed code to perform cache operations using Java objects as the media type. See Overriding the MediaType Programmatically.

This approach allows remote clients to use a binary format for storing cache entries, which is optimal. However, you must make entity classes available to the server so that it can convert between binary format and Java objects.

Additionally, if the cache uses Protobuf (`application/x-protostream`) as the binary format, you must deploy protostream marshallers so that {ProductName} can unmarshall data from your custom code.

8.5.2. Storing Data as POJOs

Storing unmarshalled Java objects in the server is not recommended. Doing so requires {brandname} to serialize data when remote clients read from the cache and then deserialize data when remote clients write to the cache.

The following example configures the cache to store entries with the `application/x-java-object` media type:

```
<distributed-cache name="my-cache">
   <encoding>
      <key media-type="application/x-java-object"/>
      <value media-type="application/x-java-object"/>
   </encoding>
</distributed-cache>
```

Hot Rod clients must use a supported marshaller when data is stored as POJOs in the cache, either the JBoss marshaller or the default Java serialization mechanism. You must also deploy the classes must be deployed in the server.

REST clients must use a storage format that {brandname} can convert to and from Java objects, currently JSON or XML.

Storing Java objects in the cache requires you to deploy entity classes to {brandname}. See Deploying Entity Classes.

Memcached clients must send and receive a serialized version of the stored POJO, which is a JBoss marshalled payload by default. However if you configure the client encoding in the appropriate Memcached connector, you change the storage format so that Memcached clients use a platform neutral format such as `JSON`.

This configuration is compatible with…	
REST clients	Yes
Java Hot Rod clients	Yes
Non-Java Hot Rod clients	No

This configuration is compatible with...	
Querying and Indexing	Yes. However, querying and indexing works with POJOs only if the entities are annotated.
Custom Java objects	Yes

8.6. Deploying Entity Classes

If you plan to store entries in the cache as custom Java objects or POJOs, you must deploy entity classes to {brandname}. Clients always exchange objects as `bytes[]`. The entity classes represent those custom objects so that {brandname} can serialize and deserialize them.

To make entity classes available to the server, do the following:

- Create a `JAR` file that contains the entities and dependencies.
- Stop {brandname} if it is running.

  {brandname} loads entity classes during boot. You cannot make entity classes available to {brandname} if the server is running.

- Copy the `JAR` file to the ***$INFINISPAN_HOME/standalone/deployments/*** directory.
- Specify the `JAR` file as a module in the cache manager configuration, as in the following example:

```
<cache-container name="local" default-cache="default">
   <modules>
     <module name="deployment.my-entities.jar"/>
   </modules>
   ...
</cache-container>
```

8.7. Trying the Interoperability Demo

Try the demo for protocol interoperability using the {brandname} Docker image at: https://github.com/infinispan-demos/endpoint-interop

Chapter 9. Marshalling

Marshalling is the process of converting Java POJOs into something that can be written in a format that can be transferred over the wire. Unmarshalling is the reverse process whereby data read from a wire format is transformed back into Java POJOs. {brandname} uses marshalling/unmarshalling in order to:

- Transform data so that it can be send over to other {brandname} nodes in a cluster.
- Transform data so that it can be stored in underlying cache stores.
- Store data in {brandname} in a wire format to provide lazy deserialization capabilities.

9.1. The Role Of JBoss Marshalling

Since performance is a very important factor in this process, {brandname} uses JBoss Marshalling framework instead of standard Java Serialization in order to marshall/unmarshall Java POJOs. Amongst other things, this framework enables {brandname} to provide highly efficient ways to marshall internal {brandname} Java POJOs that are constantly used. Apart from providing more efficient ways to marshall Java POJOs, including internal Java classes, JBoss Marshalling uses highly performant `java.io.ObjectOutput` and `java.io.ObjectInput` implementations compared to standard `java.io.ObjectOutputStream` and `java.io.ObjectInputStream`.

9.2. Support For Non-Serializable Objects

From a users perspective, a very common concern is whether {brandname} supports storing non-Serializable objects. In 4.0, an {brandname} cache instance can only store non-Serializable key or value objects if, and only if:

- cache is configured to be a local cache *and...*
- cache is not configured with lazy serialization *and...*
- cache is not configured with any write-behind cache store

If either of these options is true, key/value pairs in the cache will need to be marshalled and currently they require to either to extend `java.io.Serializable` or `java.io.Externalizable`.

Since {brandname} 5.0, marshalling non-Serializable key/value objects are supported as long as users can to provide meaningful Externalizer implementations for these non-Seralizable objects.

If you're unable to retrofit Serializable or Externalizable into the classes whose instances are stored in {brandname}, you could alternatively use something like XStream to convert your Non-Serializable objects into a String that can be stored into {brandname}. The one caveat about using XStream is that it slows down the process of storing key/value objects due to the XML transformation that it needs to do.

9.2.1. Store As Binary

Store as binary enables data to be stored in its serialized form. This can be useful to achieve lazy deserialization, which is the mechanism by which {brandname} by which serialization and deserialization of objects is deferred till the point in time in which they are used and needed. This typically means that any deserialization happens using the thread context class loader of the invocation that requires deserialization, and is an effective mechanism to provide classloader isolation. By default lazy deserialization is disabled but if you want to enable it, you can do it like this:

- Via XML at the Cache level, either under `<*-cache />` element:

```
<memory>
   <binary />
</memory>
```

- Programmatically:

```
ConfigurationBuilder builder = ...
builder.memory().storageType(StorageType.BINARY);
```

Equality Considerations

When using lazy deserialization/storing as binary, keys and values are wrapped as WrappedBytes. It is this wrapper class that transparently takes care of serialization and deserialization on demand, and internally may have a reference to the object itself being wrapped, or the serialized, byte array representation of this object.

This has a particular effect on the behavior of equality. The equals() method of this class will either compare binary representations (byte arrays) or delegate to the wrapped object instance's equals() method, depending on whether both instances being compared are in serialized or deserialized form at the time of comparison. If one of the instances being compared is in one form and the other in another form, then one instance is either serialized or deserialized.

This will affect the way keys stored in the cache will work, when `storeAsBinary` is used, since comparisons happen on the key which will be wrapped by a MarshalledValue. Implementers of equals() methods on their keys need to be aware of the behavior of equality comparison, when a key is wrapped as a MarshalledValue, as detailed above.

Store-by-value via defensive copying

The configuration `storeAsBinary` offers the possibility to enable defensive copying, which allows for store-by-value like behaviour.

{brandname} marshalls objects the moment they're stored, hence changes made to object references are not stored in the cache, not even for local caches. This provides store-by-value like behaviour. Enabling `storeAsBinary` can be achieved:

- Via XML at the Cache level, either under `<*-cache />` or `<default />` elements:

```
<store-as-binary keys="true" values="true"/>
```

- Programmatically:

```
ConfigurationBuilder builder = ...
builder.storeAsBinary().enable().storeKeysAsBinary(true).storeValuesAsBinary(true);
```

9.3. Advanced Configuration

Internally, {brandname} uses an implementation of this Marshaller interface in order to marshall/unmarshall Java objects so that they're sent other nodes in the grid, or so that they're stored in a cache store, or even so to transform them into byte arrays for lazy deserialization.

By default, {brandname} uses the GlobalMarshaller. Optionally, {brandname} users can provide their own marshaller, for example:

- Via XML at the CacheManager level, under `<cache-manager />` element:

```
<serialization marshaller="com.acme.MyMarshaller"/>
```

- Programmatically:

```
GlobalConfigurationBuilder builder = ...
builder.serialization().marshaller(myMarshaller); // needs an instance of the
marshaller
```

9.3.1. Troubleshooting

Sometimes it might happen that the {brandname} marshalling layer, and in particular JBoss Marshalling, might have issues marshalling/unmarshalling some user object. In {brandname} 4.0, marshalling exceptions will contain further information on the objects that were being marshalled. Example:

```
java.io.NotSerializableException: java.lang.Object
at org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:857)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.exts.ReplicableCommandExternalizer.writeObject(ReplicableComma
ndExternalizer.java:54)
at
org.infinispan.marshall.jboss.ConstantObjectTable$ExternalizerAdapter.writeObject(Cons
tantObjectTable.java:267)
at org.jboss.marshalling.river.RiverMarshaller.doWriteObject(RiverMarshaller.java:143)
at org.jboss.marshalling.AbstractMarshaller.writeObject(AbstractMarshaller.java:407)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectToObjectStream(JBossMarshaller.jav
a:167)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToBuffer(VersionAwareMarshaller.j
ava:92)
at
org.infinispan.marshall.VersionAwareMarshaller.objectToByteBuffer(VersionAwareMarshall
er.java:170)
at
org.infinispan.marshall.DefaultMarshallerTest.testNestedNonSerializable(VersionAwareMa
rshallerTest.java:415)
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
... Removed 22 stack frames
```

The way the "in object" messages are read is the same in which stacktraces are read. The highest "in object" being the most inner one and the lowest "in object" message being the most outer one. So, the above example indicates that a java.lang.Object instance contained in an instance of org.infinispan.commands.write.PutKeyValueCommand could not be serialized because java.lang.Object@b40ec4 is not serializable.

This is not all though! If you enable DEBUG or TRACE logging levels, marshalling exceptions will contain show the toString() representations of objects in the stacktrace. For example:

```
java.io.NotSerializableException: java.lang.Object
...
Caused by: an exception which occurred:
in object java.lang.Object@b40ec4
-> toString = java.lang.Object@b40ec4
in object org.infinispan.commands.write.PutKeyValueCommand@df661da7
-> toString = PutKeyValueCommand{key=k, value=java.lang.Object@b40ec4,
putIfAbsent=false, lifespanMillis=0, maxIdleTimeMillis=0}
```

With regards to unmarshalling exceptions, showing such level of information it's a lot more complicated but where possible. {brandname} will provide class type information. For example:

```
java.io.IOException: Injected failure!
at
org.infinispan.marshall.DefaultMarshallerTest$1.readExternal(VersionAwareMarshallerTes
t.java:426)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadNewObject(RiverUnmarshaller.java:1
172)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:273)
at
org.jboss.marshalling.river.RiverUnmarshaller.doReadObject(RiverUnmarshaller.java:210)
at org.jboss.marshalling.AbstractUnmarshaller.readObject(AbstractUnmarshaller.java:85)
at
org.infinispan.marshall.jboss.JBossMarshaller.objectFromObjectStream(JBossMarshaller.j
ava:210)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarsha
ller.java:104)
at
org.infinispan.marshall.VersionAwareMarshaller.objectFromByteBuffer(VersionAwareMarsha
ller.java:177)
at
org.infinispan.marshall.DefaultMarshallerTest.testErrorUnmarshalling(VersionAwareMarsh
allerTest.java:431)
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.DefaultMarshallerTest$1
```

In this example, an IOException was thrown when trying to unmarshall a instance of the inner class org.infinispan.marshall.DefaultMarshallerTest$1. In similar fashion to marshalling exceptions, when DEBUG or TRACE logging levels are enabled, classloader information of the class type is provided. For example:

```
java.io.IOException: Injected failure!
...
Caused by: an exception which occurred:
in object of type org.infinispan.marshall.DefaultMarshallerTest$1
-> classloader hierarchy:
-> type classloader = sun.misc.Launcher$AppClassLoader@198dfaf
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/eclipse
-testng.jar
->...file:/opt/eclipse/configuration/org.eclipse.osgi/bundles/285/1/.cp/lib/testng
-jdk15.jar
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/test-classes/
->...file:/home/galder/jboss/infinispan/code/trunk/core/target/classes/
->...file:/home/galder/.m2/repository/org/testng/testng/5.9/testng-5.9-jdk15.jar
->...file:/home/galder/.m2/repository/net/jcip/jcip-annotations/1.0/jcip-annotations
-1.0.jar
->...file:/home/galder/.m2/repository/org/easymock/easymockclassextension/2.4/easymock
classextension-2.4.jar
->...file:/home/galder/.m2/repository/org/easymock/easymock/2.4/easymock-2.4.jar
->...file:/home/galder/.m2/repository/cglib/cglib-nodep/2.1_3/cglib-nodep-2.1_3.jar
->...file:/home/galder/.m2/repository/javax/xml/bind/jaxb-api/2.1/jaxb-api-2.1.jar
->...file:/home/galder/.m2/repository/javax/xml/stream/stax-api/1.0-2/stax-api-1.0
-2.jar
->...file:/home/galder/.m2/repository/javax/activation/activation/1.1/activation
-1.1.jar
->...file:/home/galder/.m2/repository/jgroups/jgroups/2.8.0.CR1/jgroups-2.8.0.CR1.jar
->...file:/home/galder/.m2/repository/org/jboss/javaee/jboss-transaction
-api/1.0.1.GA/jboss-transaction-api-1.0.1.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/river/1.2.0.CR4
-SNAPSHOT/river-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/marshalling/marshalling-api/1.2.0.CR4
-SNAPSHOT/marshalling-api-1.2.0.CR4-SNAPSHOT.jar
->...file:/home/galder/.m2/repository/org/jboss/jboss-common-core/2.2.14.GA/jboss
-common-core-2.2.14.GA.jar
->...file:/home/galder/.m2/repository/org/jboss/logging/jboss-logging
-spi/2.0.5.GA/jboss-logging-spi-2.0.5.GA.jar
->...file:/home/galder/.m2/repository/log4j/log4j/1.2.14/log4j-1.2.14.jar
->...file:/home/galder/.m2/repository/com/thoughtworks/xstream/xstream/1.2/xstream
-1.2.jar
->...file:/home/galder/.m2/repository/xpp3/xpp3_min/1.1.3.4.O/xpp3_min-1.1.3.4.O.jar
->...file:/home/galder/.m2/repository/com/sun/xml/bind/jaxb-impl/2.1.3/jaxb-impl
-2.1.3.jar
-> parent classloader = sun.misc.Launcher$ExtClassLoader@1858610
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/localedata.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunpkcs11.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/sunjce_provider.jar
->...file:/usr/java/jdk1.5.0_19/jre/lib/ext/dnsns.jar
... Removed 22 stack frames
</code>
```

Finding the root cause of marshalling/unmarshalling exceptions can sometimes be really daunting

but we hope that the above improvements would help get to the bottom of those in a more quicker and efficient manner.

9.4. User Defined Externalizers

One of the key aspects of {brandname} is that it often needs to marshall/unmarshall objects in order to provide some of its functionality. For example, if it needs to store objects in a write-through or write-behind cache store, the stored objects need marshalling. If a cluster of {brandname} nodes is formed, objects shipped around need marshalling. Even if you enable lazy deserialization, objects need to be marshalled so that they can be lazily unmarshalled with the correct classloader.

Using standard JDK serialization is slow and produces payloads that are too big and can affect bandwidth usage. On top of that, JDK serialization does not work well with objects that are supposed to be immutable. In order to avoid these issues, {brandname} uses JBoss Marshalling for marshalling/unmarshalling objects. JBoss Marshalling is fast, produces very space efficient payloads, and on top of that during unmarshalling, it enables users to have full control over how to construct objects, hence allowing objects to carry on being immutable.

Starting with 5.0, users of {brandname} can now benefit from this marshalling framework as well, and they can provide their own externalizer implementations, but before finding out how to provide externalizers, let's look at the benefits they bring.

9.4.1. Benefits of Externalizers

The JDK provides a simple way to serialize objects which, in its simplest form, is just a matter of extending java.io.Serializable , but as it's well known, this is known to be slow and it generates payloads that are far too big. An alternative way to do serialization, still relying on JDK serialization, is for your objects to extend java.io.Externalizable . This allows for users to provide their own ways to marshall/unmarshall classes, but has some serious issues because, on top of relying on slow JDK serialization, it forces the class that you want to serialize to extend this interface, which has two side effects: The first is that you're forced to modify the source code of the class that you want to marshall/unmarshall which you might not be able to do because you either, don't own the source, or you don't even have it. Secondly, since Externalizable implementations do not control object creation, you're forced to add set methods in order to restore the state, hence potentially forcing your immutable objects to become mutable.

Instead of relying on JDK serialization, {brandname} uses JBoss Marshalling to serialize objects and requires any classes to be serialized to be associated with an Externalizer interface implementation that knows how to transform an object of a particular class into a serialized form and how to read an object of that class from a given input. {brandname} does not force the objects to be serialized to implement Externalizer. In fact, it is recommended that a separate class is used to implement the Externalizer interface because, contrary to JDK serialization, Externalizer implementations control how objects of a particular class are created when trying to read an object from a stream. This means that readObject() implementations are responsible of creating object instances of the target class, hence giving users a lot of flexibility on how to create these instances (whether direct instantiation, via factory or reflection), and more importantly, allows target classes to carry on being immutable. This type of externalizer architecture promotes good OOP designs principles, such as the principle of single responsibility .

It's quite common, and in general recommended, that Externalizer implementations are stored as inner static public classes within classes that they externalize. The advantages of doing this is that related code stays together, making it easier to maintain. In {brandname}, there are two ways in which {brandname} can be plugged with user defined externalizers:

9.4.2. User Friendly Externalizers

In the simplest possible form, users just need to provide an Externalizer implementation for the type that they want to marshall/unmarshall, and then annotate the marshalled type class with {@link SerializeWith} annotation indicating the externalizer class to use. For example:

```
import org.infinispan.commons.marshall.Externalizer;
import org.infinispan.commons.marshall.SerializeWith;

@SerializeWith(Person.PersonExternalizer.class)
public class Person {

   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public static class PersonExternalizer implements Externalizer<Person> {
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
         output.writeObject(person.name);
         output.writeInt(person.age);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), input.readInt());
      }
   }
}
```

At runtime JBoss Marshalling will inspect the object and discover that it is marshallable (thanks to the annotation) and so marshall it using the externalizer class passed. To make externalizer implementations easier to code and more typesafe, make sure you define type <T> as the type of object that's being marshalled/unmarshalled.

Even though this way of defining externalizers is very user friendly, it has some disadvantages:

- Due to several constraints of the model, such as support for different versions of the same class

or the need to marshall the Externalizer class, the payload sizes generated via this method are not the most efficient.

- This model requires that the marshalled class be annotated with link:https://docs.jboss.org/infinispan/10.0/apidocs/org/infinispan/commons/marshall/SerializeWith.html but a user might need to provide an Externalizer for a class for which source code is not available, or for any other constraints, it cannot be modified.
- The use of annotations by this model might be limiting for framework developers or service providers that try to abstract lower level details, such as the marshalling layer, away from the user.

If you're affected by any of these disadvantages, an alternative method to provide externalizers is available via more advanced externalizers:

9.4.3. Advanced Externalizers

AdvancedExternalizer provides an alternative way to provide externalizers for marshalling/unmarshalling user defined classes that overcome the deficiencies of the more user-friendly externalizer definition model explained in Externalizer. For example:

```
import org.infinispan.marshall.AdvancedExternalizer;

public class Person {

   final String name;
   final int age;

   public Person(String name, int age) {
      this.name = name;
      this.age = age;
   }

   public static class PersonExternalizer implements AdvancedExternalizer<Person> {
      @Override
      public void writeObject(ObjectOutput output, Person person)
            throws IOException {
         output.writeObject(person.name);
         output.writeInt(person.age);
      }

      @Override
      public Person readObject(ObjectInput input)
            throws IOException, ClassNotFoundException {
         return new Person((String) input.readObject(), input.readInt());
      }

      @Override
      public Set<Class<? extends Person>> getTypeClasses() {
         return Util.<Class<? extends Person>>asSet(Person.class);
      }

      @Override
      public Integer getId() {
         return 2345;
      }
   }
}
```

The first noticeable difference is that this method does not require user classes to be annotated in anyway, so it can be used with classes for which source code is not available or that cannot be modified. The bound between the externalizer and the classes that are marshalled/unmarshalled is set by providing an implementation for getTypeClasses() which should return the list of classes that this externalizer can marshall:

Linking Externalizers with Marshaller Classes

Once the Externalizer's readObject() and writeObject() methods have been implemented, it's time to link them up together with the type classes that they externalize. To do so, the Externalizer implementation must provide a getTypeClasses() implementation. For example:

```
import org.infinispan.commons.util.Util;
...
@Override
public Set<Class<? extends ReplicableCommand>> getTypeClasses() {
  return Util.asSet(LockControlCommand.class, RehashControlCommand.class,
      StateTransferControlCommand.class, GetKeyValueCommand.class,
      ClusteredGetCommand.class,
      SingleRpcCommand.class, CommitCommand.class,
      PrepareCommand.class, RollbackCommand.class,
      ClearCommand.class, EvictCommand.class,
      InvalidateCommand.class, InvalidateL1Command.class,
      PutKeyValueCommand.class, PutMapCommand.class,
      RemoveCommand.class, ReplaceCommand.class);
}
```

In the code above, ReplicableCommandExternalizer indicates that it can externalize several type of commands. In fact, it marshalls all commands that extend ReplicableCommand interface, but currently the framework only supports class equality comparison and so, it's not possible to indicate that the classes to marshalled are all children of a particular class/interface.

However there might sometimes when the classes to be externalized are private and hence it's not possible to reference the actual class instance. In this situations, users can attempt to look up the class with the given fully qualified class name and pass that back. For example:

```
@Override
public Set<Class<? extends List>> getTypeClasses() {
  return Util.<Class<? extends List>>asSet(
         Util.loadClass("java.util.Collections$SingletonList"));
}
```

Externalizer Identifier

Secondly, in order to save the maximum amount of space possible in the payloads generated, advanced externalizers require externalizer implementations to provide a positive identified via getId() implementations or via XML/programmatic configuration that identifies the externalizer when unmarshalling a payload. In order for this to work however, advanced externalizers require externalizers to be registered on cache manager creation time via XML or programmatic configuration which will be explained in next section. On the contrary, externalizers based on Externalizer and SerializeWith require no pre-registration whatsoever. Internally, {brandname} uses this advanced externalizer mechanism in order to marshall/unmarshall internal classes.

So, getId() should return a positive integer that allows the externalizer to be identified at read time to figure out which Externalizer should read the contents of the incoming buffer, or it can return null. If getId() returns null, it is indicating that the id of this advanced externalizer will be defined via XML/programmatic configuration, which will be explained in next section.

Regardless of the source of the the id, using a positive integer allows for very efficient variable length encoding of numbers, and it's much more efficient than shipping externalizer

implementation class information or class name around. {brandname} users can use any positive integer as long as it does not clash with any other identifier in the system. It's important to understand that a user defined externalizer can even use the same numbers as the externalizers in the {brandname} Core project because the internal {brandname} Core externalizers are special and they use a different number space to the user defined externalizers. On the contrary, users should avoid using numbers that are within the pre-assigned identifier ranges which can be found at the end of this article. {brandname} checks for id duplicates on startup, and if any are found, startup is halted with an error.

When it comes to maintaining which ids are in use, it's highly recommended that this is done in a centralized way. For example, getId() implementations could reference a set of statically defined identifiers in a separate class or interface. Such class/interface would give a global view of the identifiers in use and so can make it easier to assign new ids.

Registering Advanced Externalizers

The following example shows the type of configuration required to register an advanced externalizer implementation for Person object shown earlier stored as a static inner class within it:

infinispan.xml

```
<infinispan>
  <cache-container>
    <serialization>
      <advanced-externalizer class="Person$PersonExternalizer"/>
    </serialization>
  </cache-container>
  ...
</infinispan>
```

Programmatically:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer());
```

As mentioned earlier, when listing these externalizer implementations, users can optionally provide the identifier of the externalizer via XML or programmatically instead of via getId() implementation. Again, this offers a centralized way to maintain the identifiers but it's important that the rules are clear: An AdvancedExternalizer implementation, either via XML/programmatic configuration or via annotation, needs to be associated with an identifier. If it isn't, {brandname} will throw an error and abort startup. If a particular AdvancedExternalizer implementation defines an id both via XML/programmatic configuration and annotation, the value defined via XML/programmatically is the one that will be used. Here's an example of an externalizer whose id is defined at registration time:

infinispan.xml

```
<infinispan>
  <cache-container>
    <serialization>
      <advanced-externalizer id="123"
                             class="Person$PersonExternalizer"/>
    </serialization>
  </cache-container>
  ...
</infinispan>
```

Programmatically:

```
GlobalConfigurationBuilder builder = ...
builder.serialization()
   .addAdvancedExternalizer(123, new Person.PersonExternalizer());
```

Finally, a couple of notes about the programmatic configuration. GlobalConfiguration.addExternalizer() takes varargs, so it means that it is possible to register multiple externalizers in just one go, assuming that their ids have already been defined via @Marshalls annotation. For example:

```
builder.serialization()
   .addAdvancedExternalizer(new Person.PersonExternalizer(),
                            new Address.AddressExternalizer());
```

Preassigned Externalizer Id Ranges

This is the list of Externalizer identifiers that are used by {brandname} based modules or frameworks. {brandname} users should avoid using ids within these ranges.

{brandname} Tree Module:	1000 - 1099
{brandname} Server Modules:	1100 - 1199
Hibernate {brandname} Second Level Cache:	1200 - 1299
{brandname} Lucene Directory:	1300 - 1399
Hibernate OGM:	1400 - 1499
Hibernate Search:	1500 - 1599
{brandname} Query Module:	1600 - 1699
{brandname} Remote Query Module:	1700 - 1799
{brandname} Scripting Module:	1800 - 1849
{brandname} Server Event Logger Module:	1850 - 1899
{brandname} Remote Store:	1900 - 1999

{brandname} Counters:	2000 - 2049
{brandname} Multimap:	2050 - 2099
{brandname} Locks:	2100 - 2149

Chapter 10. Grid File System

{brandname}'s GridFileSystem is an experimental API that exposes an {brandname}-backed data grid as a file system.

This is a deprecated API that will be removed in a future version.

Specifically, the API works as an extension to the JDK's File , InputStream and OutputStream classes: specifically, GridFile, GridInputStream and GridOutputStream. A helper class, GridFilesystem, is also included.

Essentially, the GridFilesystem is backed by 2 {brandname} caches - one for metadata (typically replicated) and one for the actual data (typically distributed). The former is replicated so that each node has metadata information locally and would not need to make RPC calls to list files, etc. The latter is distributed since this is where the bulk of storage space is used up, and a scalable mechanism is needed here. Files themselves are chunked and each chunk is stored as a cache entry, as a byte array.

Here is a quick code snippet demonstrating usage:

```
Cache<String,byte[]> data = cacheManager.getCache("distributed");
Cache<String,GridFile.Metadata> metadata = cacheManager.getCache("replicated");
GridFilesystem fs = new GridFilesystem(data, metadata);

// Create directories
File file=fs.getFile("/tmp/testfile/stuff");
fs.mkdirs(); // creates directories /tmp/testfile/stuff

// List all files and directories under "/usr/local"
file=fs.getFile("/usr/local");
File[] files=file.listFiles();

// Create a new file
file=fs.getFile("/tmp/testfile/stuff/README.txt");
file.createNewFile();
```

Copying stuff to the grid file system:

```
InputStream in=new FileInputStream("/tmp/my-movies/dvd-image.iso");
OutputStream out=fs.getOutput("/grid-movies/dvd-image.iso");
byte[] buffer=new byte[20000];
int len;
while((len=in.read(buffer, 0, buffer.length)) != -1) out.write(buffer, 0, len);
in.close();
out.close();
```

Reading stuff from the grid:

```
InputStream in=in.getInput("/grid-movies/dvd-image.iso");
OutputStream out=new FileOutputStream("/tmp/my-movies/dvd-image.iso");
byte[] buffer=new byte[200000];
int len;
while((len=in.read(buffer, 0, buffer.length)) != -1) out.write(buffer, 0, len);
in.close();
out.close();
```

10.1. WebDAV demo

{brandname} ships with a demo WebDAV application that makes use of the grid file system APIs. This demo app is packaged as a WAR file which can be deployed in a servlet container, such as JBoss AS or Tomcat, and exposes the grid as a file system over WebDAV. This could then be mounted as a remote drive on your operating system.

Chapter 11. CDI Support

{brandname} includes integration with Contexts and Dependency Injection (better known as CDI) via {brandname}'s `infinispan-cdi-embedded` or `infinispan-cdi-remote` module. CDI is part of Java EE specification and aims for managing beans' lifecycle inside the container. The integration allows to inject Cache interface and bridge Cache and CacheManager events. JCache annotations (JSR-107) are supported by `infinispan-jcache` and `infinispan-jcache-remote` artifacts. For more information have a look at Chapter 11 of the JCACHE specification.

11.1. Maven Dependencies

To include CDI support for {brandname} in your project, use one of the following dependencies:

pom.xml for Embedded mode

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-embedded</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

pom.xml for Remote mode

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-cdi-remote</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

Which version of {brandname} should I use?

We recommend using the latest final version {brandname}.

11.2. Embedded cache integration

11.2.1. Inject an embedded cache

By default you can inject the default {brandname} cache. Let's look at the following example:

Default cache injection

```
...
import javax.inject.Inject;

public class GreetingService {

    @Inject
    private Cache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

If you want to use a specific cache rather than the default one, you just have to provide your own cache configuration and cache qualifier. See example below:

Qualifier example

```
...
import javax.inject.Qualifier;

@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface GreetingCache {
}
```

Injecting Cache with qualifier

```
...
import org.infinispan.configuration.cache.Configuration;
import org.infinispan.configuration.cache.ConfigurationBuilder;
import org.infinispan.cdi.ConfigureCache;
import javax.enterprise.inject.Produces;

public class Config {

    @ConfigureCache("greeting-cache") // This is the cache name.
    @GreetingCache // This is the cache qualifier.
    @Produces
    public Configuration greetingCacheConfiguration() {
        return new ConfigurationBuilder()
                    .memory()
                        .size(1000)
                    .build();
    }

    // The same example without providing a custom configuration.
    // In this case the default cache configuration will be used.
    @ConfigureCache("greeting-cache")
    @GreetingCache
    @Produces
    public Configuration greetingCacheConfiguration;
}
```

To use this cache in the GreetingService add the `@GeetingCache` qualifier on your cache injection point.

11.2.2. Override the default embedded cache manager and configuration

You can override the default cache configuration used by the default `EmbeddedCacheManager`. For that, you just have to create a `Configuration` producer with default qualifiers as illustrated in the following snippet:

Overriding Configuration

```
public class Config {

    // By default CDI adds the @Default qualifier if no other qualifier is provided.
    @Produces
    public Configuration defaultEmbeddedCacheConfiguration() {
        return new ConfigurationBuilder()
                    .memory()
                        .size(100)
                    .build();
    }
}
```

It's also possible to override the default `EmbeddedCacheManager`. The newly created manager must have default qualifiers and Application scope.

Overriding EmbeddedCacheManager

```
...
import javax.enterprise.context.ApplicationScoped;

public class Config {

    @Produces
    @ApplicationScoped
    public EmbeddedCacheManager defaultEmbeddedCacheManager() {
      return new DefaultCacheManager(new ConfigurationBuilder()
                                          .memory()
                                              .size(100)
                                          .build());
   }
}
```

11.2.3. Configure the transport for clustered use

To use {brandname} in a clustered mode you have to configure the transport with the `GlobalConfiguration`. To achieve that override the default cache manager as explained in the previous section. Look at the following snippet:

Overriding default EmbeddedCacheManager

```
...
package org.infinispan.configuration.global.GlobalConfigurationBuilder;

@Produces
@ApplicationScoped
public EmbeddedCacheManager defaultClusteredCacheManager() {
    return new DefaultCacheManager(
        new GlobalConfigurationBuilder().transport().defaultTransport().build(),
        new ConfigurationBuilder().memory().size(7).build()
    );
}
```

11.3. Remote cache integration

11.3.1. Inject a remote cache

With the CDI integration it's also possible to use a `RemoteCache` as illustrated in the following snippet:

Injecting RemoteCache

```
public class GreetingService {

    @Inject
    private RemoteCache<String, String> cache;

    public String greet(String user) {
        String cachedValue = cache.get(user);
        if (cachedValue == null) {
            cachedValue = "Hello " + user;
            cache.put(user, cachedValue);
        }
        return cachedValue;
    }
}
```

If you want to use another cache, for example the greeting-cache, add the `@Remote` qualifier on the cache injection point which contains the cache name.

Injecting RemoteCache with qualifier

```
public class GreetingService {

    @Inject
    @Remote("greeting-cache")
    private RemoteCache<String, String> cache;

    ...
}
```

Adding the `@Remote` cache qualifier on each injection point might be error prone. That's why the remote cache integration provides another way to achieve the same goal. For that you have to create your own qualifier annotated with `@Remote`:

RemoteCache qualifier

```
@Remote("greeting-cache")
@Qualifier
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface RemoteGreetingCache {
}
```

To use this cache in the GreetingService add the qualifier `@RemoteGreetingCache` qualifier on your cache injection.

11.3.2. Override the default remote cache manager

Like the embedded cache integration, the remote cache integration comes with a default remote cache manager producer. This default `RemoteCacheManager` can be overridden as illustrated in the following snippet:

Overriding default RemoteCacheManager

```
public class Config {

    @Produces
    @ApplicationScoped
    public RemoteCacheManager defaultRemoteCacheManager() {
        return new RemoteCacheManager(localhost, 1544);
    }
}
```

11.4. Use a custom remote/embedded cache manager for one or more cache

It's possible to use a custom cache manager for one or more cache. You just need to annotate the cache manager producer with the cache qualifiers. Look at the following example:

```
public class Config {

   @GreetingCache
   @Produces
   @ApplicationScoped
   public EmbeddedCacheManager specificEmbeddedCacheManager() {
      return new DefaultCacheManager(new ConfigurationBuilder()
                                          .expiration()
                                              .lifespan(60000l)
                                          .build());
   }

   @RemoteGreetingCache
   @Produces
   @ApplicationScoped
   public RemoteCacheManager specificRemoteCacheManager() {
       return new RemoteCacheManager("localhost", 1544);
   }
}
```

With the above code the GreetingCache or the RemoteGreetingCache will be associated with the produced cache manager.

Producer method scope

To work properly the producers must have the scope @ApplicationScoped . Otherwise each injection of cache will be associated to a new instance of cache manager.

11.5. Use JCache caching annotations

There is now a separate module for JSR 107 (JCACHE) integration, including API.

When CDI integration and JCache artifacts are present on the classpath, it is possible to use JCache annotations with CDI managed beans. These annotations provide a simple way to handle common use cases. The following caching annotations are defined in this specification:

- `@CacheResult` - caches the result of a method call
- `@CachePut` - caches a method parameter
- `@CacheRemoveEntry` - removes an entry from a cache

- `@CacheRemoveAll` - removes all entries from a cache

Annotations target type

These annotations must only be used on methods.

To use these annotations, proper interceptors need to be declared in `beans.xml` file:

Interceptors for managed environments such as Application Servers

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.InjectedCacheResultInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCachePutInterceptor</class>
    <class>
org.infinispan.jcache.annotation.InjectedCacheRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.InjectedCacheRemoveAllInterceptor</class>
  </interceptors>
</beans>
```

Interceptors for unmanaged environments such as standalone applications

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/beans_1_1.xsd"
   version="1.2" bean-discovery-mode="annotated">

  <interceptors>
    <class>org.infinispan.jcache.annotation.CacheResultInterceptor</class>
    <class>org.infinispan.jcache.annotation.CachePutInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveEntryInterceptor</class>
    <class>org.infinispan.jcache.annotation.CacheRemoveAllInterceptor</class>
  </interceptors>
</beans>
```

The following snippet of code illustrates the use of `@CacheResult` annotation. As you can see it simplifies the caching of the `Greetingservice#greet` method results.

Using JCache annotations

```
import javax.cache.interceptor.CacheResult;

public class GreetingService {

    @CacheResult
    public String greet(String user) {
        return "Hello" + user;
    }
}
```

The first version of the `GreetingService` and the above version have exactly the same behavior. The only difference is the cache used. By default it's the fully qualified name of the annotated method with its parameter types (e.g. `org.infinispan.example.GreetingService.greet(java.lang.String)`).

Using other cache than default is rather simple. All you need to do is to specify its name with the `cacheName` attribute of the cache annotation. For example:

Specifying cache name for JCache

```
@CacheResult(cacheName = "greeting-cache")
```

11.6. Use Cache events and CDI

It is possible to receive Cache and Cache Manager level events using CDI Events. You can achieve it using `@Observes` annotation as shown in the following snippet:

Event listeners based on CDI

```
import javax.enterprise.event.Observes;
import org.infinispan.notifications.cachemanagerlistener.event.CacheStartedEvent;
import org.infinispan.notifications.cachelistener.event.*;

public class GreetingService {

    // Cache level events
    private void entryRemovedFromCache(@Observes CacheEntryCreatedEvent event) {
        ...
    }

    // Cache Manager level events
    private void cacheStarted(@Observes CacheStartedEvent event) {
        ...
    }
}
```

Chapter 12. JCache (JSR-107) provider

{brandname} provides an implementation of JCache 1.0 API (JSR-107). JCache specifies a standard Java API for caching temporary Java objects in memory. Caching java objects can help get around bottlenecks arising from using data that is expensive to retrieve (i.e. DB or web service), or data that is hard to calculate. Caching these type of objects in memory can help speed up application performance by retrieving the data directly from memory instead of doing an expensive roundtrip or recalculation. This document specifies how to use JCache with {brandname}'s implementation of the specification, and explains key aspects of the API.

12.1. Dependencies

In order to start using {brandname} JCache implementation, a single dependency needs to be added to the Maven pom.xml file:

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-jcache</artifactId>
  <version>${version.infinispan}</version>
  <scope>test</scope>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

12.2. Create a local cache

Creating a local cache, using default configuration options as defined by the JCache API specification, is as simple as doing the following:

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager
CacheManager cacheManager = Caching.getCachingProvider().getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("namedCache",
      new MutableConfiguration<String, String>());
```

By default, the JCache API specifies that data should be stored as `storeByValue`, so that object state mutations outside of operations to the cache, won't have an impact in the objects stored in the cache. {brandname} has so far implemented this using serialization/marshalling to make copies to store in the cache, and that way adhere to the spec. Hence, if using default JCache configuration with {brandname}, data stored must be marshallable.

Alternatively, JCache can be configured to store data by reference (just like {brandname} or JDK Collections work). To do that, simply call:

```
Cache<String, String> cache = cacheManager.createCache("namedCache",
      new MutableConfiguration<String, String>().setStoreByValue(false));
```

12.3. Create a remote cache

Creating a remote cache (client-server mode), using default configuration options as defined by the JCache API specification, is as simple as doing the following:

```
import javax.cache.*;
import javax.cache.configuration.*;

// Retrieve the system wide cache manager via
org.infinispan.jcache.remote.JCachingProvider
CacheManager cacheManager = Caching.getCachingProvider(
"org.infinispan.jcache.remote.JCachingProvider").getCacheManager();
// Define a named cache with default JCache configuration
Cache<String, String> cache = cacheManager.createCache("remoteNamedCache",
      new MutableConfiguration<String, String>());
```

In order to use the org.infinispan.jcache.remote.JCachingProvider, infinispan-jcache-remote-<version>.jar and all its transitive dependencies need to be on put your classpath.

12.4. Store and retrieve data

Even though JCache API does not extend neither java.util.Map not java.util.concurrent.ConcurrentMap, it providers a key/value API to store and retrieve data:

```
import javax.cache.*;
import javax.cache.configuration.*;

CacheManager cacheManager = Caching.getCacheManager();
Cache<String, String> cache = cacheManager.createCache("namedCache",
      new MutableConfiguration<String, String>());
cache.put("hello", "world"); // Notice that javax.cache.Cache.put(K) returns void!
String value = cache.get("hello"); // Returns "world"
```

Contrary to standard java.util.Map, javax.cache.Cache comes with two basic put methods called put and getAndPut. The former returns `void` whereas the latter returns the previous value associated with the key. So, the equivalent of java.util.Map.put(K) in JCache is javax.cache.Cache.getAndPut(K).

Even though JCache API only covers standalone caching, it can be plugged with a persistence store, and has been designed with clustering or distribution in mind. The reason why javax.cache.Cache offers two put methods is because standard java.util.Map put call forces implementors to calculate the previous value. When a persistent store is in use, or the cache is distributed, returning the previous value could be an expensive operation, and often users call standard java.util.Map.put(K) without using the return value. Hence, JCache users need to think about whether the return value is relevant to them, in which case they need to call javax.cache.Cache.getAndPut(K) , otherwise they can call java.util.Map.put(K, V) which avoids returning the potentially expensive operation of returning the previous value.

12.5. Comparing java.util.concurrent.ConcurrentMap and javax.cache.Cache APIs

Here's a brief comparison of the data manipulation APIs provided by java.util.concurrent.ConcurrentMap and javax.cache.Cache APIs.

Operation	`java.util.concurrent.ConcurrentMap<K, V>`	`javax.cache.Cache<K, V>`
store and no return	N/A	`void put(K key)`
store and return previous value	`V put(K key)`	`V getAndPut(K key)`
store if not present	`V putIfAbsent(K key, V value)`	`boolean putIfAbsent(K key, V value)`
retrieve	`V get(Object key)`	`V get(K key)`
delete if present	`V remove(Object key)`	`boolean remove(K key)`
delete and return previous value	`V remove(Object key)`	`V getAndRemove(K key)`
delete conditional	`boolean remove(Object key, Object value)`	`boolean remove(K key, V oldValue)`
replace if present	`V replace(K key, V value)`	`boolean replace(K key, V value)`
replace and return previous value	`V replace(K key, V value)`	`V getAndReplace(K key, V value)`
replace conditional	`boolean replace(K key, V oldValue, V newValue)`	`boolean replace(K key, V oldValue, V newValue)`

Comparing the two APIs, it's obvious to see that, where possible, JCache avoids returning the previous value to avoid operations doing expensive network or IO operations. This is an overriding principle in the design of JCache API. In fact, there's a set of operations that are present in java.util.concurrent.ConcurrentMap , but are not present in the javax.cache.Cache because they could be expensive to compute in a distributed cache. The only exception is iterating over the contents of the cache:

Operation	java.util.concurrent.ConcurrentMap<K, V>	javax.cache.Cache<K, V>
calculate size of cache	int size()	N/A
return all keys in the cache	Set<K> keySet()	N/A
return all values in the cache	Collection<V> values()	N/A
return all entries in the cache	Set<Map.Entry<K, V>> entrySet()	N/A
iterate over the cache	use iterator() method on keySet, values or entrySet	Iterator<Cache.Entry<K, V>> iterator()

12.6. Clustering JCache instances

{brandname} JCache implementation goes beyond the specification in order to provide the possibility to cluster caches using the standard API. Given a {brandname} configuration file configured to replicate caches like this:

infinispan.xml

```
<infinispan>
   <cache-container default-cache="namedCache">
      <transport cluster="jcache-cluster" />
      <replicated-cache name="namedCache" />
   </cache-container>
</infinispan>
```

You can create a cluster of caches using this code:

```
import javax.cache.*;
import java.net.URI;

// For multiple cache managers to be constructed with the standard JCache API
// and live in the same JVM, either their names, or their classloaders, must
// be different.
// This example shows how to force their classloaders to be different.
// An alternative method would have been to duplicate the XML file and give
// it a different name, but this results in unnecessary file duplication.
ClassLoader tccl = Thread.currentThread().getContextClassLoader();
CacheManager cacheManager1 = Caching.getCachingProvider().getCacheManager(
      URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));
CacheManager cacheManager2 = Caching.getCachingProvider().getCacheManager(
      URI.create("infinispan-jcache-cluster.xml"), new TestClassLoader(tccl));

Cache<String, String> cache1 = cacheManager1.getCache("namedCache");
Cache<String, String> cache2 = cacheManager2.getCache("namedCache");

cache1.put("hello", "world");
String value = cache2.get("hello"); // Returns "world" if clustering is working

// --

public static class TestClassLoader extends ClassLoader {
  public TestClassLoader(ClassLoader parent) {
     super(parent);
  }
}
```

Chapter 13. Multimap Cache

MutimapCache is a type of {brandname} Cache that maps keys to values in which each key can contain multiple values.

13.1. Installation and configuration

pom.xml

```
<dependency>
  <groupId>org.infinispan</groupId>
  <artifactId>infinispan-multimap</artifactId>
  <version>${version.infinispan}</version>
</dependency>
```

Replace `${version.infinispan}` with the appropriate version of {brandname}.

13.2. MultimapCache API

MultimapCache API exposes several methods to interact with the Multimap Cache. All these methods are non-blocking in most of the cases. See [limitations]

```
public interface MultimapCache<K, V> {

   CompletableFuture<Void> put(K key, V value);

   CompletableFuture<Collection<V>> get(K key);

   CompletableFuture<Boolean> remove(K key);

   CompletableFuture<Boolean> remove(K key, V value);

   CompletableFuture<Void> remove(Predicate<? super V> p);

   CompletableFuture<Boolean> containsKey(K key);

   CompletableFuture<Boolean> containsValue(V value);

   CompletableFuture<Boolean> containsEntry(K key, V value);

   CompletableFuture<Long> size();

   boolean supportsDuplicates();

}
```

13.2.1. CompletableFuture<Void> put(K key, V value)

Puts a key-value pair in the multimap cache.

```
MultimapCache<String, String> multimapCache = ...;

multimapCache.put("girlNames", "marie")
             .thenCompose(r1 -> multimapCache.put("girlNames", "oihana"))
             .thenCompose(r3 -> multimapCache.get("girlNames"))
             .thenAccept(names -> {
                         if(names.contains("marie"))
                             System.out.println("Marie is a girl name");

                          if(names.contains("oihana"))
                             System.out.println("Oihana is a girl name");
                       });
```

The output of this code is :

```
Marie is a girl name
Oihana is a girl name
```

13.2.2. CompletableFuture<Collection<V>> get(K key)

Asynchronous that returns a view collection of the values associated with key in this multimap cache, if any. Any changes to the retrieved collection won't change the values in this multimap cache. When this method returns an empty collection, it means the key was not found.

13.2.3. CompletableFuture<Boolean> remove(K key)

Asynchronous that removes the entry associated with the key from the multimap cache, if such exists.

13.2.4. CompletableFuture<Boolean> remove(K key, V value)

Asynchronous that removes a key-value pair from the multimap cache, if such exists.

13.2.5. CompletableFuture<Void> remove(Predicate<? super V> p)

Asynchronous method. Removes every value that match the given predicate.

13.2.6. CompletableFuture<Boolean> containsKey(K key)

Asynchronous that returns true if this multimap contains the key.

13.2.7. CompletableFuture<Boolean> containsValue(V value)

Asynchronous that returns true if this multimap contains the value in at least one key.

13.2.8. CompletableFuture<Boolean> containsEntry(K key, V value)

Asynchronous that returns true if this multimap contains at least one key-value pair with the value.

13.2.9. CompletableFuture<Long> size()

Asynchronous that returns the number of key-value pairs in the multimap cache. It doesn't return the distinct number of keys.

13.2.10. boolean supportsDuplicates()

Asynchronous that returns true if the multimap cache supports duplicates. This means that the content of the multimap can be 'a' → ['1', '1', '2']. For now this method will always return false, as duplicates are not yet supported. The existence of a given value is determined by 'equals' and `hashcode' method's contract.

13.3. Creating a Multimap Cache

Currently the MultimapCache is configured as a regular cache. This can be done either by code or XML configuration. See how to configure a regular Cache in the section link to [configure a cache].

13.3.1. Embedded mode

```
// create or obtain your EmbeddedCacheManager
EmbeddedCacheManager cm = ... ;

// create or obtain a MultimapCacheManager passing the EmbeddedCacheManager
MultimapCacheManager multimapCacheManager = EmbeddedMultimapCacheManagerFactory.from
(cm);

// define the configuration for the multimap cache
multimapCacheManager.defineConfiguration(multimapCacheName, c.build());

// get the multimap cache
multimapCache = multimapCacheManager.get(multimapCacheName);
```

13.3.2. Server mode

TODO

13.4. Limitations

In almost every case the Multimap Cache will behave as a regular Cache, but some limitations exist

in the current version.

13.4.1. Support for duplicates

Duplicates are not supported yet. This means that the multimap won't contain any duplicate key-value pair. Whenever put method is called, if the key-value pair already exist, this key-value par won't be added. Methods used to check if a key-value pair is already present in the Multimap are the `equals` and `hashcode`.

13.4.2. Eviction

For now, the eviction works per key, and not per key-value pair. This means that whenever a key is evicted, all the values associated with the key will be evicted too. Eviction per key-value could be supported in the future.

13.4.3. Transactions

Implicit transactions are supported through the auto-commit and all the methods are non blocking. Explicit transactions work without blocking in most of the cases. Methods that will block are `size`, `containsEntry` and `remove(Predicate<? super V> p)`

Chapter 14. Transactions

{brandname} can be configured to use and to participate in JTA compliant transactions. Alternatively, if transaction support is disabled, it is equivalent to using autocommit in JDBC calls, where modifications are potentially replicated after every change (if replication is enabled).

On every cache operation {brandname} does the following:

1. Retrieves the current Transaction associated with the thread
2. If not already done, registers XAResource with the transaction manager to be notified when a transaction commits or is rolled back.

In order to do this, the cache has to be provided with a reference to the environment's TransactionManager. This is usually done by configuring the cache with the class name of an implementation of the TransactionManagerLookup interface. When the cache starts, it will create an instance of this class and invoke its `getTransactionManager()` method, which returns a reference to the `TransactionManager`.

{brandname} ships with several transaction manager lookup classes:

Transaction manager lookup implementations

- EmbeddedTransactionManagerLookup: This provides with a basic transaction manager which should only be used for embedded mode when no other implementation is available. This implementation has some severe limitations to do with concurrent transactions and recovery.
- JBossStandaloneJTAManagerLookup: If you're running {brandname} in a standalone environment, or in JBoss AS 7 and earlier, and WildFly 8, 9, and 10, this should be your default choice for transaction manager. It's a fully fledged transaction manager based on JBoss Transactions which overcomes all the deficiencies of the `EmbeddedTransactionManager`.
- WildflyTransactionManagerLookup: If you're running {brandname} in Wildfly 11 or later, this should be your default choice for transaction manager.
- GenericTransactionManagerLookup: This is a lookup class that locate transaction managers in the most popular Java EE application servers. If no transaction manager can be found, it defaults on the `EmbeddedTransactionManager`.

WARN: `DummyTransactionManagerLookup` has been deprecated in 9.0 and it will be removed in the future. Use `EmbeddedTransactionManagerLookup` instead.

Once initialized, the `TransactionManager` can also be obtained from the `Cache` itself:

```
//the cache must have a transactionManagerLookupClass defined
Cache cache = cacheManager.getCache();

//equivalent with calling TransactionManagerLookup.getTransactionManager();
TransactionManager tm = cache.getAdvancedCache().getTransactionManager();
```

14.1. Configuring transactions

Transactions are configured at cache level. Below is the configuration that affects a transaction behaviour and a small description of each configuration attribute.

```
<locking
   isolation="READ_COMMITTED"
   write-skew="false"/>
<transaction
   locking="OPTIMISTIC"
   auto-commit="true"
   complete-timeout="60000"
   mode="NONE"
   notifications="true"
   protocol="DEFAULT"
   reaper-interval="30000"
   recovery-cache="__recoveryInfoCacheName__"
   stop-timeout="30000"
   transaction-manager-lookup=
"org.infinispan.transaction.lookup.GenericTransactionManagerLookup"/>
<versioning
   scheme="NONE"/>
```

or programmatically:

```
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.locking()
    .isolationLevel(IsolationLevel.READ_COMMITTED)
    .writeSkewCheck(false);
builder.transaction()
    .lockingMode(LockingMode.OPTIMISTIC)
    .autoCommit(true)
    .completedTxTimeout(60000)
    .transactionMode(TransactionMode.NON_TRANSACTIONAL)
    .useSynchronization(false)
    .notifications(true)
    .transactionProtocol(TransactionProtocol.DEFAULT)
    .reaperWakeUpInterval(30000)
    .cacheStopTimeout(30000)
    .transactionManagerLookup(new GenericTransactionManagerLookup())
    .recovery()
    .enabled(false)
    .recoveryInfoCacheName("__recoveryInfoCacheName__");
builder.versioning()
    .enabled(false)
    .scheme(VersioningScheme.NONE);
```

- `isolation` - configures the isolation level. Check section Isolation Levels for more details. Default

is REPEATABLE_READ.

- write-skew - enables write skew checks (deprecated). {brandname} automatically sets this attribute in Library Mode. Default is false for READ_COMMITTED. Default is true for REPEATABLE_READ. See Write Skews for more details.
- locking - configures whether the cache uses optimistic or pessimistic locking. Check section Transaction Locking for more details. Default is OPTIMISTIC.
- auto-commit - if enable, the user does not need to start a transaction manually for a single operation. The transaction is automatically started and committed. Default is true.
- complete-timeout - the duration in milliseconds to keep information about completed transactions. Default is 60000.
- mode - configures whether the cache is transactional or not. Default is NONE. The available options are:
  - NONE - non transactional cache
  - FULL_XA - XA transactional cache with recovery enabled. Check section Transaction recovery for more details about recovery.
  - NON_DURABLE_XA - XA transactional cache with recovery disabled.
  - NON_XA - transactional cache with integration via Synchronization instead of XA. Check section Enlisting Synchronizations for details.
  - BATCH- transactional cache using batch to group operations. Check section Batching for details.
- notifications - enables/disables triggering transactional events in cache listeners. Default is true.
- protocol - configures the protocol uses. Default is DEFAULT. Values available are:
  - DEFAULT - uses the traditional Two-Phase-Commit protocol. It is described below.
  - TOTAL_ORDER - uses total order ensured by the Transport to commit transactions. Check section Total Order based commit protocol for details.
- reaper-interval - the time interval in millisecond at which the thread that cleans up transaction completion information kicks in. Defaults is 30000.
- recovery-cache - configures the cache name to store the recovery information. Check section Transaction recovery for more details about recovery. Default is recoveryInfoCacheName.
- stop-timeout - the time in millisecond to wait for ongoing transaction when the cache is stopping. Default is 30000.
- transaction-manager-lookup - configures the fully qualified class name of a class that looks up a reference to a javax.transaction.TransactionManager. Default is org.infinispan.transaction.lookup.GenericTransactionManagerLookup.
- Versioning scheme - configure the version scheme to use when write skew is enabled with optimistic or total order transactions. Check section Write Skews for more details. Default is NONE.

For more details on how Two-Phase-Commit (2PC) is implemented in {brandname} and how locks are being acquired see the section below. More details about the configuration settings are

available in Configuration reference.

14.2. Isolation levels

{brandname} offers two isolation levels - READ_COMMITTED and REPEATABLE_READ.

These isolation levels determine when readers see a concurrent write, and are internally implemented using different subclasses of `MVCCEntry`, which have different behaviour in how state is committed back to the data container.

Here's a more detailed example that should help understand the difference between `READ_COMMITTED` and `REPEATABLE_READ` in the context of {brandname}. With `READ_COMMITTED`, if between two consecutive read calls on the same key, the key has been updated by another transaction, the second read may return the new updated value:

```
Thread1: tx1.begin()
Thread1: cache.get(k) // returns v
Thread2:                                    tx2.begin()
Thread2:                                    cache.get(k) // returns v
Thread2:                                    cache.put(k, v2)
Thread2:                                    tx2.commit()
Thread1: cache.get(k) // returns v2!
Thread1: tx1.commit()
```

With `REPEATABLE_READ`, the final get will still return `v`. So, if you're going to retrieve the same key multiple times within a transaction, you should use `REPEATABLE_READ`.

However, as read-locks are not acquired even for `REPEATABLE_READ`, this phenomena can occur:

```
cache.get("A") // returns 1
cache.get("B") // returns 1

Thread1: tx1.begin()
Thread1: cache.put("A", 2)
Thread1: cache.put("B", 2)
Thread2:                                    tx2.begin()
Thread2:                                    cache.get("A") // returns 1
Thread1: tx1.commit()
Thread2:                                    cache.get("B") // returns 2
Thread2:                                    tx2.commit()
```

14.3. Transaction locking

14.3.1. Pessimistic transactional cache

From a lock acquisition perspective, pessimistic transactions obtain locks on keys at the time the key is written.

1. A lock request is sent to the primary owner (can be an explicit lock request or an operation)
2. The primary owner tries to acquire the lock:
    a. If it succeed, it sends back a positive reply;
    b. Otherwise, a negative reply is sent and the transaction is rollback.

As an example:

```
transactionManager.begin();
cache.put(k1,v1); //k1 is locked.
cache.remove(k2); //k2 is locked when this returns
transactionManager.commit();
```

When `cache.put(k1,v1)` returns, `k1` is locked and no other transaction running anywhere in the cluster can write to it. Reading `k1` is still possible. The lock on `k1` is released when the transaction completes (commits or rollbacks).

For conditional operations, the validation is performed in the originator.

14.3.2. Optimistic transactional cache

With optimistic transactions locks are being acquired at transaction prepare time and are only being held up to the point the transaction commits (or rollbacks). This is different from the 5.0 default locking model where local locks are being acquire on writes and cluster locks are being acquired during prepare time.

1. The prepare is sent to all the owners.
2. The primary owners try to acquire the locks needed:
    a. If locking succeeds, it performs the write skew check.
    b. If the write skew check succeeds (or is disabled), send a positive reply.
    c. Otherwise, a negative reply is sent and the transaction is rolled back.

As an example:

```
transactionManager.begin();
cache.put(k1,v1);
cache.remove(k2);
transactionManager.commit(); //at prepare time, K1 and K2 is locked until
committed/rolled back.
```

For conditional commands, the validation still happens on the originator.

14.3.3. What do I need - pessimistic or optimistic transactions?

From a use case perspective, optimistic transactions should be used when there is *not* a lot of

contention between multiple transactions running at the same time. That is because the optimistic transactions rollback if data has changed between the time it was read and the time it was committed (with write skew check enabled).

On the other hand, pessimistic transactions might be a better fit when there is high contention on the keys and transaction rollbacks are less desirable. Pessimistic transactions are more costly by their nature: each write operation potentially involves a RPC for lock acquisition.

14.4. Write Skews

Write skews occur when two transactions independently and simultaneously read and write to the same key. The result of a write skew is that both transactions successfully commit updates to the same key but with different values.

In Library Mode, {brandname} automatically performs write skew checks to ensure data consistency for `REPEATABLE_READ` isolation levels in optimistic transactions. This allows {brandname} to detect and roll back one of the transactions.

The `write-skew` attribute is deprecated for Library Mode. In Remote Client/Server Mode, this attribute is not a valid declaration.

When operating in `LOCAL` mode, write skew checks rely on Java object references to compare differences, which provides a reliable technique for checking for write skews.

In clustered environments, you should configure data versioning to ensure reliable write skew checks. {brandname} provides an implementation of the `EntryVersion` interface called `SIMPLE` versioning, which is backed by a long that is incremented each time the entry is updated.

```
<versioning scheme="SIMPLE|NONE" />
```

Or

```
new ConfigurationBuilder().versioning().scheme(SIMPLE);
```

14.4.1. Forcing write locks on keys in pessimitic transactions

To avoid write-skews with pessimistic transactions, lock keys at read-time with `Flag.FORCE_WRITE_LOCK`.

- In non-transactional caches, `Flag.FORCE_WRITE_LOCK` does not work. The `get()` call reads the key value but does not acquire locks remotely.
- You should use `Flag.FORCE_WRITE_LOCK` with transactions in which the entity is updated later in the same transaction.

Compare the following code snippets for an example of `Flag.FORCE_WRITE_LOCK`:

```
// begin the transaction
if (!cache.getAdvancedCache().lock(key)) {
   // abort the transaction because the key was not locked
} else {
   cache.get(key);
   cache.put(key, value);
   // commit the transaction
}
```

```
// begin the transaction
try {
   // throws an exception if the key is not locked.
   cache.getAdvancedCache().withFlags(Flag.FORCE_WRITE_LOCK).get(key);
   cache.put(key, value);
} catch (CacheException e) {
   // mark the transaction rollback-only
}
// commit or rollback the transaction
```

14.5. Dealing with exceptions

If a CacheException (or a subclass of it) is thrown by a cache method within the scope of a JTA transaction, then the transaction is automatically marked for rollback.

14.6. Enlisting Synchronizations

By default {brandname} registers itself as a first class participant in distributed transactions through XAResource. There are situations where {brandname} is not required to be a participant in the transaction, but only to be notified by its lifecycle (prepare, complete): e.g. in the case {brandname} is used as a 2nd level cache in Hibernate.

{brandname} allows transaction enlistment through Synchronization. To enable it just use `NON_XA` transaction mode.

`Synchronization`s have the advantage that they allow `TransactionManager` to optimize 2PC with a 1PC where only one other resource is enlisted with that transaction (last resource commit optimization). E.g. Hibernate second level cache: if {brandname} registers itself with the `TransactionManager` as a `XAResource` than at commit time, the `TransactionManager` sees two `XAResource` (cache and database) and does not make this optimization. Having to coordinate between two resources it needs to write the tx log to disk. On the other hand, registering {brandname} as a `Synchronisation` makes the `TransactionManager` skip writing the log to the disk (performance improvement).

14.7. Batching

Batching allows atomicity and some characteristics of a transaction, but not full-blown JTA or XA capabilities. Batching is often a lot lighter and cheaper than a full-blown transaction.

Generally speaking, one should use batching API whenever the only participant in the transaction is an {brandname} cluster. On the other hand, JTA transactions (involving `TransactionManager`) should be used whenever the transactions involves multiple systems. E.g. considering the "Hello world!" of transactions: transferring money from one bank account to the other. If both accounts are stored within {brandname}, then batching can be used. If one account is in a database and the other is {brandname}, then distributed transactions are required.

You *do not* have to have a transaction manager defined to use batching.

14.7.1. API

Once you have configured your cache to use batching, you use it by calling `startBatch()` and `endBatch()` on `Cache`. E.g.,

```
Cache cache = cacheManager.getCache();
// not using a batch
cache.put("key", "value"); // will replicate immediately

// using a batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k2", "value");
cache.endBatch(true); // This will now replicate the modifications since the batch was
started.

// a new batch
cache.startBatch();
cache.put("k1", "value");
cache.put("k2", "value");
cache.put("k3", "value");
cache.endBatch(false); // This will "discard" changes made in the batch
```

14.7.2. Batching and JTA

Behind the scenes, the batching functionality starts a JTA transaction, and all the invocations in that scope are associated with it. For this it uses a very simple (e.g. no recovery) internal `TransactionManager` implementation. With batching, you get:

1. Locks you acquire during an invocation are held until the batch completes
2. Changes are all replicated around the cluster in a batch as part of the batch completion process. Reduces replication chatter for each update in the batch.
3. If synchronous replication or invalidation are used, a failure in replication/invalidation will cause the batch to roll back.
4. All the transaction related configurations apply for batching as well.

14.8. Transaction recovery

Recovery is a feature of XA transactions, which deal with the eventuality of a resource or possibly even the transaction manager failing, and recovering accordingly from such a situation.

14.8.1. When to use recovery

Consider a distributed transaction in which money is transferred from an account stored in an external database to an account stored in {brandname}. When `TransactionManager.commit()` is invoked, both resources prepare successfully (1st phase). During the commit (2nd) phase, the database successfully applies the changes whilst {brandname} fails before receiving the commit request from the transaction manager. At this point the system is in an inconsistent state: money is taken from the account in the external database but not visible yet in {brandname} (since locks are only released during 2nd phase of a two-phase commit protocol). Recovery deals with this situation to make sure data in both the database and {brandname} ends up in a consistent state.

14.8.2. How does it work

Recovery is coordinated by the transaction manager. The transaction manager works with {brandname} to determine the list of in-doubt transactions that require manual intervention and informs the system administrator (via email, log alerts, etc). This process is transaction manager specific, but generally requires some configuration on the transaction manager.

Knowing the in-doubt transaction ids, the system administrator can now connect to the {brandname} cluster and replay the commit of transactions or force the rollback. {brandname} provides JMX tooling for this - this is explained extensively in the Transaction recovery and reconciliation section.

14.8.3. Configuring recovery

Recovery is *not* enabled by default in {brandname}. If disabled, the `TransactionManager` won't be able to work with {brandname} to determine the in-doubt transactions. The Transaction configuration section shows how to enable it.

NOTE: `recovery-cache` attribute is not mandatory and it is configured per-cache.

For recovery to work, `mode` must be set to `FULL_XA`, since full-blown XA transactions are needed.

Enable JMX support

In order to be able to use JMX for managing recovery JMX support must be explicitly enabled.

14.8.4. Recovery cache

In order to track in-doubt transactions and be able to reply them, {brandname} caches all transaction state for future use. This state is held only for in-doubt transaction, being removed for successfully completed transactions after when the commit/rollback phase completed.

This in-doubt transaction data is held within a local cache: this allows one to configure swapping this info to disk through cache loader in the case it gets too big. This cache can be specified through the `recovery-cache` configuration attribute. If not specified {brandname} will configure a local cache for you.

It is possible (though not mandated) to share same recovery cache between all the {brandname} caches that have recovery enabled. If the default recovery cache is overridden, then the specified recovery cache must use a TransactionManagerLookup that returns a different transaction manager than the one used by the cache itself.

14.8.5. Integration with the transaction manager

Even though this is transaction manager specific, generally a transaction manager would need a reference to a `XAResource` implementation in order to invoke `XAResource.recover()` on it. In order to obtain a reference to an {brandname} `XAResource` following API can be used:

```
XAResource xar = cache.getAdvancedCache().getXAResource();
```

It is a common practice to run the recovery in a different process from the one running the transaction.

14.8.6. Reconciliation

The transaction manager informs the system administrator on in-doubt transaction in a proprietary way. At this stage it is assumed that the system administrator knows transaction's XID (a byte array).

A normal recovery flow is:

- **STEP 1**: The system administrator connects to an {brandname} server through JMX, and lists the in doubt transactions. The image below demonstrates JConsole connecting to an {brandname} node that has an in doubt transaction.

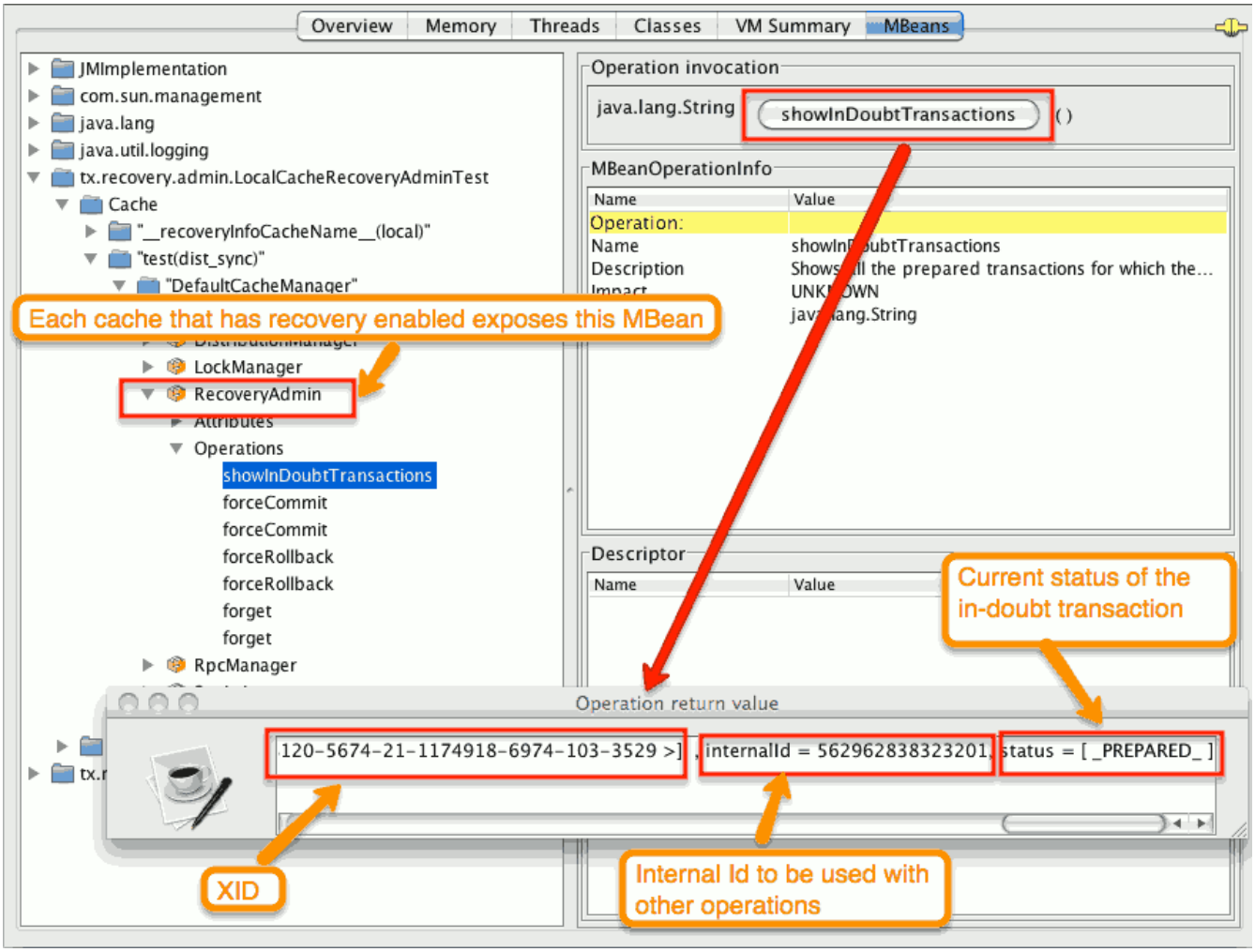

Figure 1. Show in-doubt transactions

The status of each in-doubt transaction is displayed(in this example " *PREPARED* "). There might be multiple elements in the status field, e.g. "PREPARED" and "COMMITTED" in the case the transaction committed on certain nodes but not on all of them.

- **STEP 2**: The system administrator visually maps the XID received from the transaction manager to an {brandname} internal id, represented as a number. This step is needed because the XID, a byte array, cannot conveniently be passed to the JMX tool (e.g. JConsole) and then re-assembled on {brandname}'s side.
- **STEP 3**: The system administrator forces the transaction's commit/rollback through the corresponding jmx operation, based on the internal id. The image below is obtained by forcing the commit of the transaction based on its internal id.

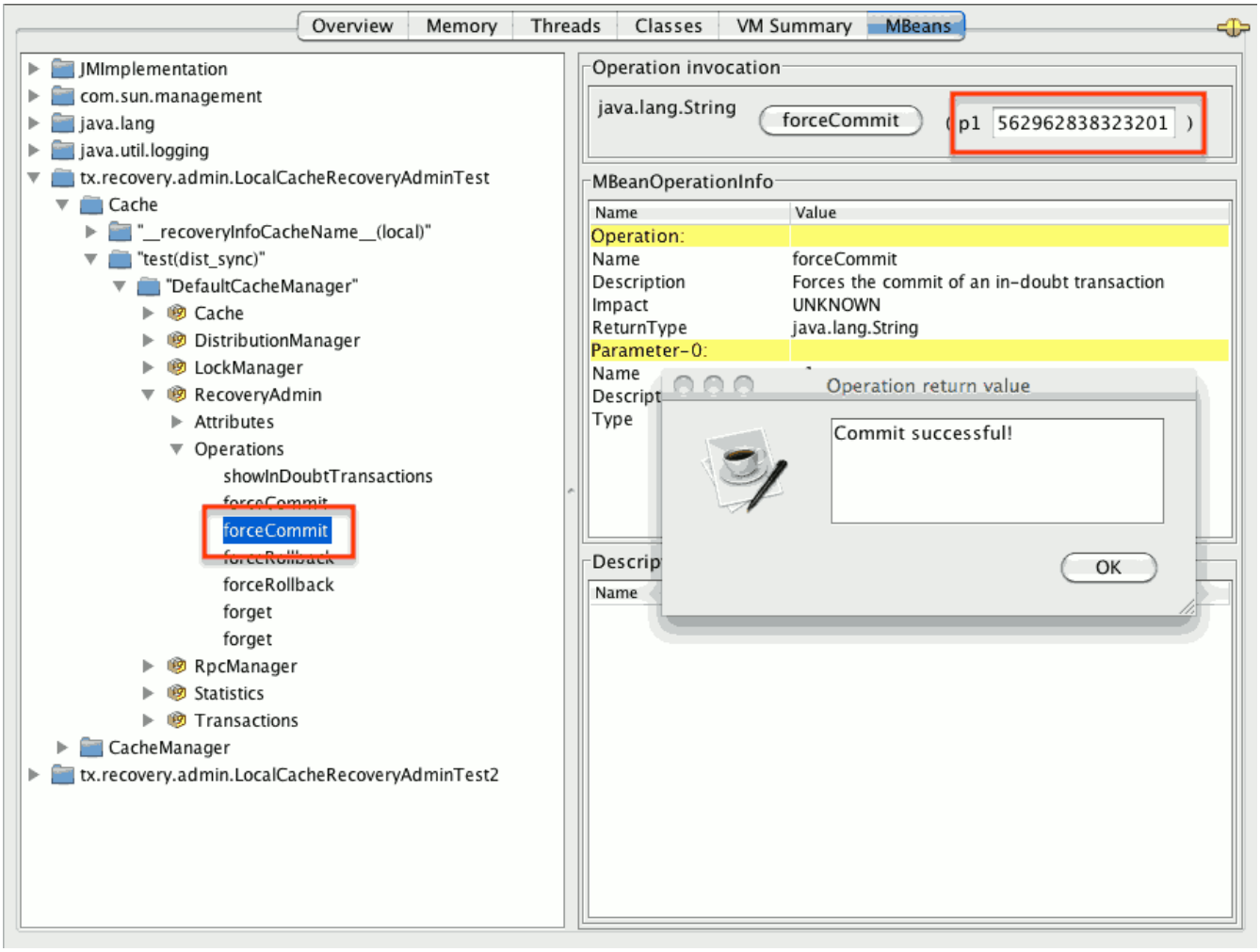

Figure 2. Force commit

All JMX operations described above can be executed on any node, regardless of where the transaction originated.

Force commit/rollback based on XID

XID-based JMX operations for forcing in-doubt transactions' commit/rollback are available as well: these methods receive byte[] arrays describing the XID instead of the number associated with the transactions (as previously described at step 2). These can be useful e.g. if one wants to set up an automatic completion job for certain in-doubt transactions. This process is plugged into transaction manager's recovery and has access to the transaction manager's XID objects.

14.8.7. Want to know more?

The recovery design document describes in more detail the insides of transaction recovery implementation.

14.9. Total Order based commit protocol

The Total Order based protocol is a multi-master scheme (in this context, multi-master scheme means that all nodes can update all the data) as the (optimistic/pessimist) locking mode implemented in {brandname}. This commit protocol relies on the concept of totally ordered

delivery of messages which, informally, implies that each node which delivers a set of messages, delivers them in the same order.

This protocol comes with this advantages.

1. transactions can be committed in one phase, as they are delivered in the same order by the nodes that receive them.
2. it mitigates distributed deadlocks.

The weaknesses of this approach are the fact that its implementation relies on a single thread per node which delivers the transaction and its modification, and the slightly cost of total ordering the messages in `Transport`.

Thus, this protocol delivers best performance in scenarios of *high contention* , in which it can benefit from the single-phase commit and the deliver thread is not the bottleneck.

Currently, the Total Order based protocol is available only in *transactional* caches for *replicated* and *distributed* modes.

14.9.1. Overview

The Total Order based commit protocol only affects how transactions are committed by {brandname} and the isolation level and write skew affects it behaviour.

When write skew is disabled, the transaction can be committed/rolled back in single phase. The data consistency is guaranteed by the `Transport` that ensures that all owners of a key will deliver the same transactions set by the same order.

On other hand, when write skew is enabled, the protocol adapts and uses one phase commit when it is safe. In `XaResource` enlistment, we can use one phase if the `TransactionManager` request a commit in one phase (last resource commit optimization) and the {brandname} cache is configured in replicated mode. This optimization is not safe in distributed mode because each node performs the write skew check validation in different keys subset. When in `Synchronization` enlistment, the `TransactionManager` does not provide any information if {brandname} is the only resource enlisted (last resource commit optimization), so it is not possible to commit in a single phase.

Commit in one phase

When the transaction ends, {brandname} sends the transaction (and its modification) in total order. This ensures all the transactions are deliver in the same order in all the involved {brandname} nodes. As a result, when a transaction is delivered, it performs a deterministic write skew check over the same state (if enabled), leading to the same outcome (transaction commit or rollback).

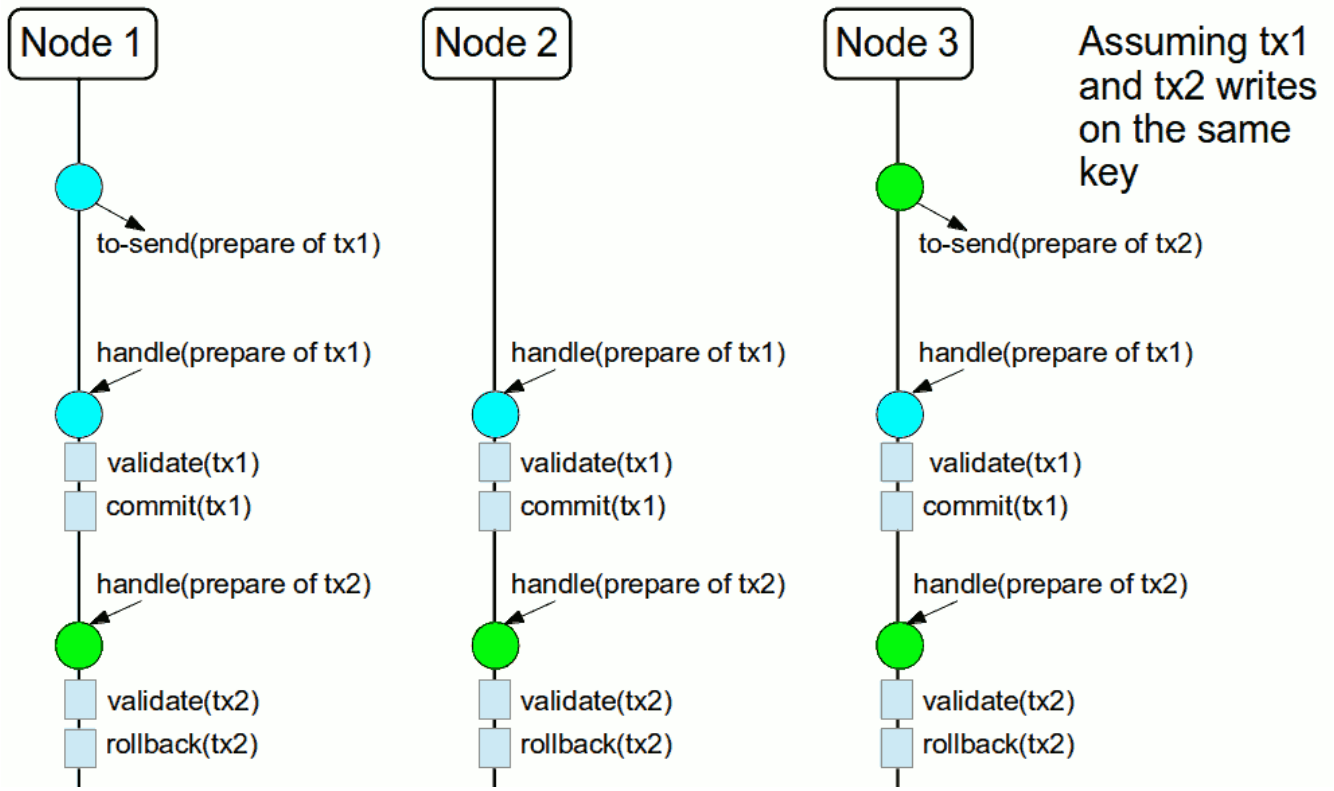

Figure 3. 1-phase commit

The figure above demonstrates a high level example with 3 nodes. `Node1` and `Node3` are running one transaction each and lets assume that both transaction writes on the same key. To make it more interesting, lets assume that both nodes tries to commit at the same time, represented by the first colored circle in the figure. The *blue* circle represents the transaction *tx1* and the *green* the transaction *tx2* . Both nodes do a remote invocation in total order (*to-send*) with the transaction's modifications. At this moment, all the nodes will agree in the same deliver order, for example, *tx1* followed by *tx2* . Then, each node delivers *tx1* , perform the validation and commits the modifications. The same steps are performed for *tx2* but, in this case, the validation will fail and the transaction is rollback in all the involved nodes.

Commit in two phases

In the first phase, it sends the modification in total order and the write skew check is performed. The result of the write skew check is sent back to the originator. As soon as it has the confirmation that all keys are successfully validated, it give a positive response to the `TransactionManager`. On other hand, if it receives a negative reply, it returns a negative response to the `TransactionManager`. Finally, the transaction is committed or aborted in the second phase depending of the `TransactionManager` request.

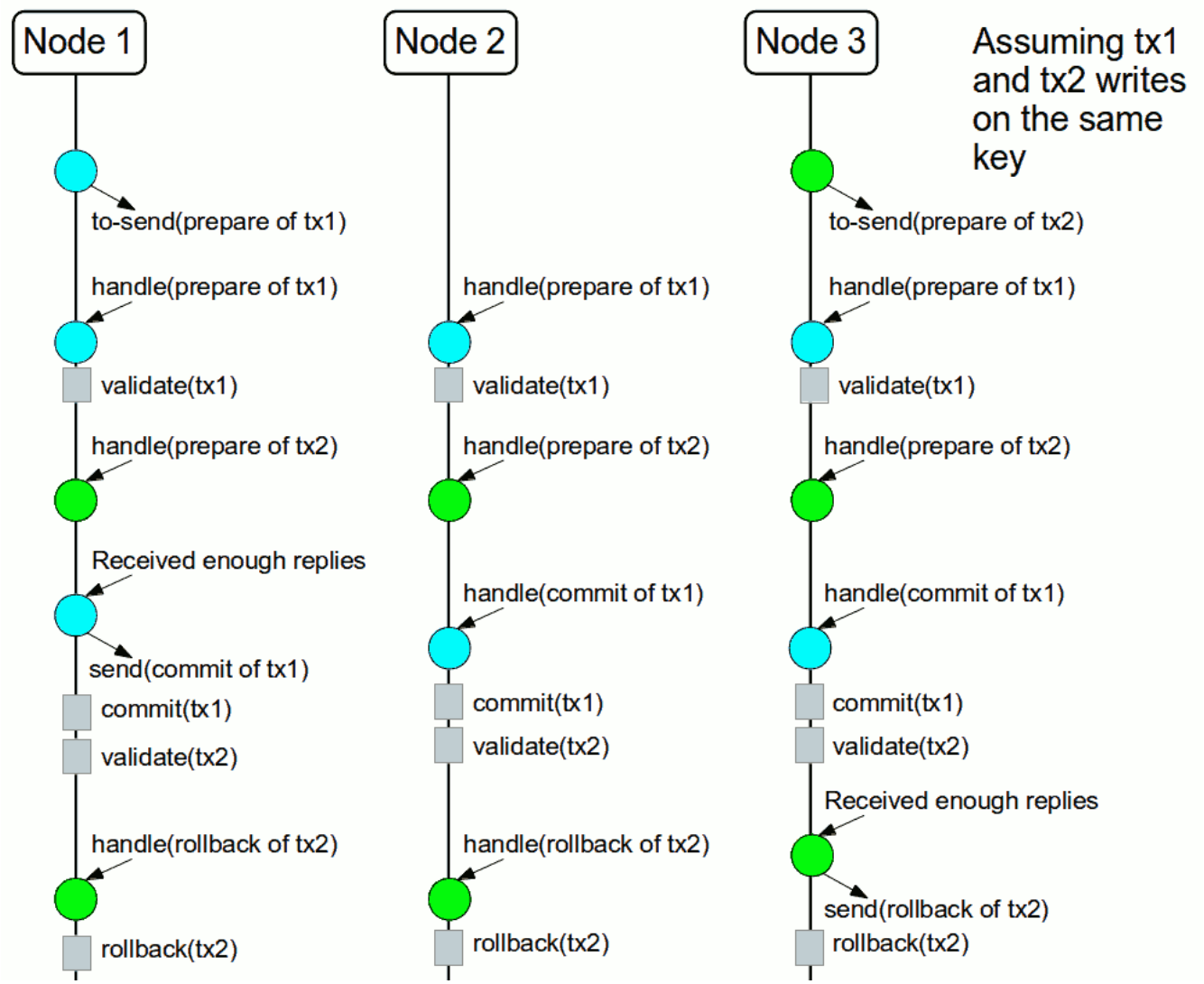

Figure 4. 2-phase commit

The figure above shows the scenario described in the first figure but now committing the transactions using two phases. When *tx1* is deliver, it performs the validation and it replies to the `TransactionManager`. Next, lets assume that *tx2* is deliver before the `TransactionManager` request the second phase for *tx1*. In this case, *tx2* will be enqueued and it will be validated only when *tx1* is completed. Eventually, the `TransactionManager` for *tx1* will request the second phase (the commit) and all the nodes are free to perform the validation of *tx2* .

Transaction Recovery

Transaction recovery is currently not available for Total Order based commit protocol.

State Transfer

For simplicity reasons, the total order based commit protocol uses a blocking version of the current state transfer. The main differences are:

1. enqueue the transaction deliver while the state transfer is in progress;
2. the state transfer control messages (`CacheTopologyControlCommand`) are sent in total order.

This way, it provides a synchronization between the state transfer and the transactions deliver that is the same all the nodes. Although, the transactions caught in the middle of state transfer (i.e. sent

before the state transfer start and deliver after it) needs to be re-sent to find a new total order involving the new joiners.

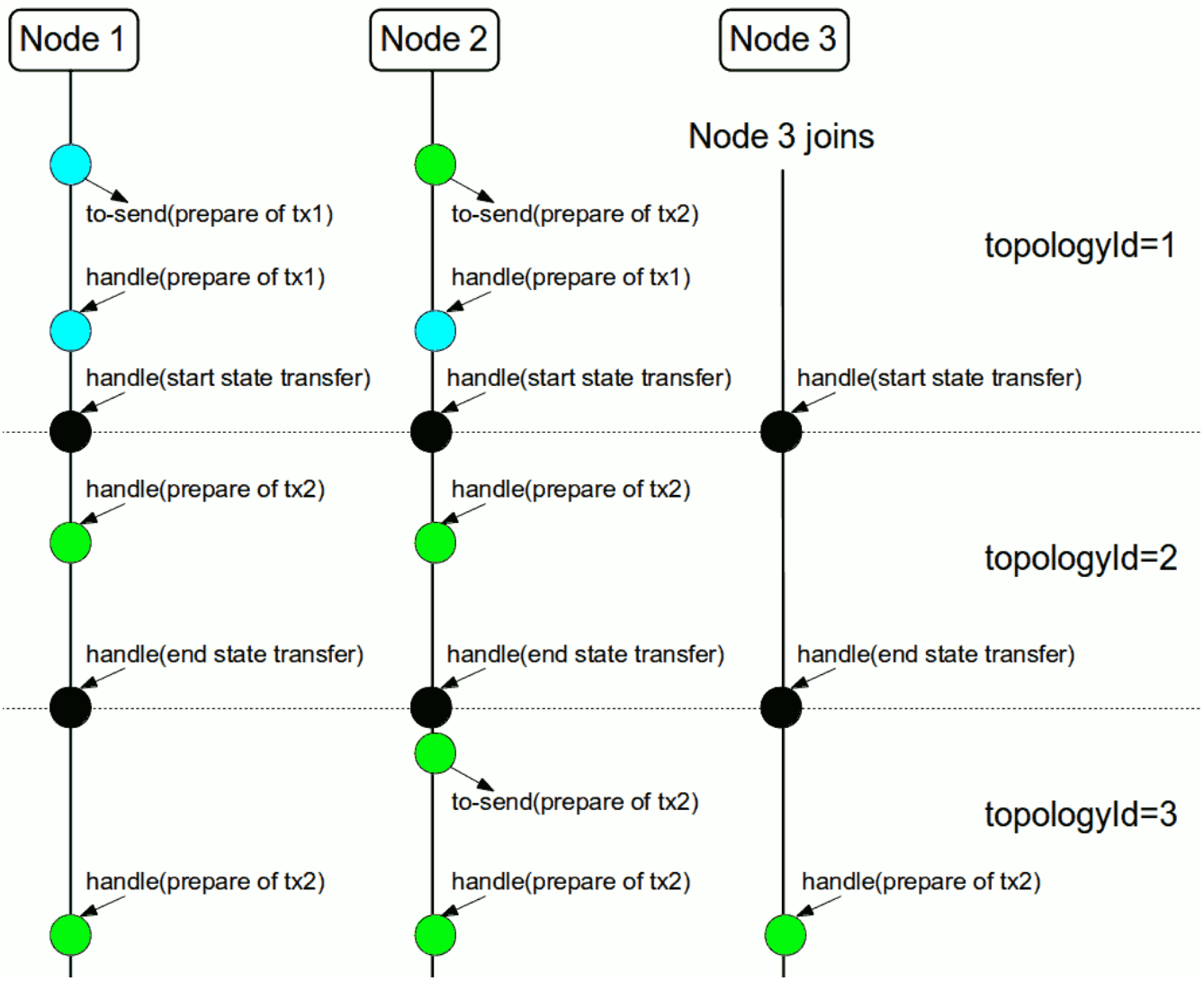

Figure 5. Node joining during transaction

The figure above describes a node joining. In the scenario, the *tx2* is sent in *topologyId=1* but when it is received, it is in *topologyId=2* . So, the transaction is re-sent involving the new nodes.

14.9.2. Configuration

To use total order in your cache, you need to add the `TOA` protocol in your `jgroups.xml` configuration file.

jgroups.xml

```
<tom.TOA />
```

Check the JGroups Manual for more details.

If you are interested in detail how JGroups guarantees total order, check the link::http://jgroups.org/manual/index.html#TOA[TOA manual].

Also, you need to set the `protocol=TOTAL_ORDER` in the `<transaction>` element, as shown in Transaction configuration.

14.9.3. When to use it?

Total order shows benefits when used in write intensive and high contented workloads. It avoids contention in the lock keys.