JBoss Cache User Guide

A clustered, transactional cache

Release 2.1.0 Alegrias
November 2007

Authors:

ManikSurtani(manik@jboss.org)

BelaBan(bela@jboss.com)

BenWang(ben.wang@jboss.com)

BrianStansberry(brian.stansberry@jboss.com)

GalderZamarreño(galder.zamarreno@jboss.com)

DanielHuang(dhuang@jboss.org)

MirceaMarkus(mircea.markus@jboss.com)

Table of Contents

Preface

This is the official JBoss Cache user guide. Along with its accompanying documents (an FAQ, a tutorial and a whole set of documents on PojoCache), this is freely available on the JBoss Cache documentation site. [http://labs.jboss.com/jbosscache]

When used, JBoss Cache refers to JBoss Cache Core, a tree-structured, clustered, transactional cache. Pojo Cache, also a part of the JBoss Cache distribution, is documented separately. (Pojo Cache is a cache that deals with Plain Old Java Objects, complete with object relationships, with the ability to cluster such pojos while maintaining their relationships. Please see the Pojo Cache documentation for more information about this.)

This book is targeted at both developers wishing to use JBoss Cache as a clustering and caching library in their codebase, as well as people who wish to "OEM" JBoss Cache by building on and extending its features. As such, this book is split into two major sections - one detailing the "User" API and the other going much deeper into specialist topics and the JBoss Cache architecture.

In general, a good knowledge of the Java programming language along with a strong appreciation and understanding of transactions and concurrent threads is necessary. No prior knowledge of JBoss Application Server is expected or required.

For further discussion, use the user forum [http://www.jboss.com/index.html?module=bb&op=viewforum&f=157] linked on the JBoss Cache website. [http://labs.jboss.com/jbosscache] We also provide a mechanism for tracking bug reports and feature requests on the JBoss Cache JIRA issue tracker. [http://jira.jboss.com/jira/browse/JBCACHE] If you are interested in the development of JBoss Cache or in translating this documentation into other languages, we'd love to hear from you. Please post a message on the user forum [http://www.jboss.com/index.html?module=bb&op=viewforum&f=157] or contact us by using the JBoss Cache developer mailing list. [https://lists.jboss.org/mailman/listinfo/jbosscache-dev]

This book is specifically targeted at the JBoss Cache release of the same version number. It may not apply to older or newer releases of JBoss Cache. It is important that you use the documentation appropriate to the version of JBoss Cache you intend to use.

Part I. Introduction to JBoss Cache

This section covers what developers would need to quickly start using JBoss Cache in their projects. It covers an overview of the concepts and API, configuration and deployment information.

1

Overview

1.1. What is JBoss Cache?

JBoss Cache is a tree-structured, clustered, transactional cache. It is the backbone for many fundamental JBoss Application Server clustering services, including - in certain versions - clustering JNDI, HTTP and EJB sessions.

JBoss Cache can also be used as a standalone transactional and clustered caching library or even an object oriented data store. It can even be embedded in other enterprise Java frameworks and application servers such as BEA WebLogic or IBM WebSphere, Tomcat, Spring, Hibernate, and many others. It is also very commonly used directly by standalone Java applications that do not run from within an application server, to maintain clustered state.

1.1.1. And what is Pojo Cache?

Pojo Cache is an extension of the core JBoss Cache API. Pojo Cache offers additional functionality such as:

- maintaining object references even after replication or persistence.
- fine grained replication, where only modified object fields are replicated.
- "API-less" clustering model where pojos are simply annotated as being clustered.

Pojo Cache has a complete and separate set of documentation, including a user guide, FAQ and tutorial and as such, Pojo Cache is not discussed further in this book.

1.2. Summary of Features

JBoss Cache offers a simple and straightforward API, where data (simple Java objects) can be placed in the cache and, based on configuration options selected, this data may be one or all of:

- replicated to some or all cache instances in a cluster.
- persisted to disk and/or a remote cluster ("far-cache").
- garbage collected from memory when memory runs low, and passivated to disk so state isn't lost.

In addition, JBoss Cache offers a rich set of enterprise-class features:

- being able to participate in JTA transactions (works with Java EE compliant TransactionManagers).

- attach to JMX servers and provide runtime statistics on the state of the cache.
- allow client code to attach listeners and receive notifications on cache events.

A cache is organised as a tree, with a single root. Each node in the tree essentially contains a Map, which acts as a store for key/value pairs. The only requirement placed on objects that are cached is that they implement `java.io.Serializable` . Note that this requirement does not exist for Pojo Cache.

JBoss Cache can be either local or replicated. Local trees exist only inside the JVM in which they are created, whereas replicated trees propagate any changes to some or all other trees in the same cluster. A cluster may span different hosts on a network or just different JVMs on a single host.

When a change is made to an object in the cache and that change is done in the context of a transaction, the replication of changes is deferred until the transaction commits successfully. All modifications are kept in a list associated with the transaction for the caller. When the transaction commits, we replicate the changes. Otherwise, (on a rollback) we simply undo the changes locally resulting in zero network traffic and overhead. For example, if a caller makes 100 modifications and then rolls back the transaction, we will not replicate anything, resulting in no network traffic.

If a caller has no transaction associated with it (and isolation level is not NONE - more about this later), we will replicate right after each modification, e.g. in the above case we would send 100 messages, plus an additional message for the rollback. In this sense, running without a transaction can be thought of as analogous as running with auto-commit switched on in JDBC terminology, where each operation is committed automatically.

JBoss Cache works out of the box with most popular transaction managers, and even provides an API where custom transaction manager lookups can be written.

The cache is also completely thread-safe. It uses a pessimistic locking scheme for nodes in the tree by default, with an optimistic locking scheme as a configurable option. With pessimistic locking, the degree of concurrency can be tuned using a number of isolation levels, corresponding to database-style transaction isolation levels, i.e., SERIALIZABLE, REPEATABLE_READ, READ_COMMITTED, READ_UNCOMMITTED and NONE. Concurrency, locking and isolation levels will be discussed later.

1.3. Requirements

JBoss Cache requires Java 5.0 (or newer).

However, there is a way to build JBoss Cache as a Java 1.4.x compatible binary using JBossRetro [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossRetro] to retroweave the Java 5.0 binaries. However, Red Hat Inc. does not offer professional support around the retroweaved binary at this time and the Java 1.4.x compatible binary is not in the binary distribution. See this wiki [http://wiki.jboss.org/wiki/Wiki.jsp?page=JBossCacheHabaneroJava1.4] page for details on building the retroweaved binary for yourself.

In addition to Java 5.0, at a minimum, JBoss Cache has dependencies on JGroups [http://www.jgroups.org] , and Apache's commons-logging [http://jakarta.apache.org/commons/logging/] . JBoss Cache ships with all dependent libraries necessary to run out of the box.

1.4. License

JBoss Cache is an open source product, using the business and OEM-friendly OSI-approved [http://www.opensource.org/] LGPL license. [http://www.gnu.org/copyleft/lesser.html] Commercial development support, production support and training for JBoss Cache is available through JBoss, a division of Red Hat Inc. [http://www.jboss.com] JBoss Cache is a part of JBoss Professional Open Source JEMS [http://www.jboss.comindex] (JBoss Enterprise Middleware Suite).

2

User API

2.1. API Classes

The `Cache` interface is the primary mechanism for interacting with JBoss Cache. It is constructed and optionally started using the `CacheFactory` . The `CacheFactory` allows you to create a `Cache` either from a `Configuration` object or an XML file. Once you have a reference to a `Cache` , you can use it to look up `Node` objects in the tree structure, and store data in the tree.

Reviewing the javadoc for the above interfaces is the best way to learn the API. Below we cover some of the main points.

2.2. Instantiating and Starting the Cache

An instance of the `Cache` interface can only be created via a `CacheFactory` . (This is unlike JBoss Cache 1.x, where an instance of the old `TreeCache` class could be directly instantiated.)

`CacheFactory` provides a number of overloaded methods for creating a `Cache` , but they all do the same thing:

- Gain access to a `Configuration` , either by having one passed in as a method parameter, or by parsing XML content and constructing one. The XML content can come from a provided input stream or from a classpath or filesystem location. See the chapter on Configuration for more on obtaining a `Configuration` .
- Instantiate the `Cache` and provide it with a reference to the `Configuration` .
- Optionally invoke the cache's `create()` and `start()` methods.

An example of the simplest mechanism for creating and starting a cache, using the default configuration values:

```
CacheFactory factory = DefaultCacheFactory.getInstance();
Cache cache = factory.createCache();
```

Here we tell the `CacheFactory` to find and parse a configuration file on the classpath:

```
CacheFactory factory = DefaultCacheFactory.getInstance();
Cache cache = factory.createCache("cache-configuration.xml");
```

Here we configure the cache from a file, but want to programatically change a configuration element. So, we tell the factory not to start the cache, and instead do it ourselves:

```
CacheFactory factory = DefaultCacheFactory.getInstance();
Cache cache = factory.createCache("cache-configuration.xml", false);
Configuration config = cache.getConfiguration();
config.setClusterName(this.getClusterName());

// Have to create and start cache before using it
cache.create();
cache.start();
```

2.3. Caching and Retrieving Data

Next, let's use the `Cache` API to access a `Node` in the cache and then do some simple reads and writes to that node.

```
// Let's get ahold of the root node.
Node rootNode = cache.getRoot();

// Remember, JBoss Cache stores data in a tree structure.
// All nodes in the tree structure are identified by Fqn objects.
Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

// Create a new Node
Node peterGriffin = rootNode.addChild(peterGriffinFqn);

// let's store some data in the node
peterGriffin.put("isCartoonCharacter", Boolean.TRUE);
peterGriffin.put("favouriteDrink", new Beer());

// some tests (just assume this code is in a JUnit test case)
assertTrue(peterGriffin.get("isCartoonCharacter"));
assertEquals(peterGriffinFqn, peterGriffin.getFqn());
assertTrue(rootNode.hasChild(peterGriffinFqn));

Set keys = new HashSet();
keys.add("isCartoonCharacter");
keys.add("favouriteDrink");

assertEquals(keys, peterGriffin.getKeys());

// let's remove some data from the node
peterGriffin.remove("favouriteDrink");

assertNull(peterGriffin.get("favouriteDrink");

// let's remove the node altogether
rootNode.removeChild(peterGriffinFqn);

assertFalse(rootNode.hasChild(peterGriffinFqn));
```

The `Cache` interface also exposes put/get/remove operations that take an Fqn as an argument:

```
Fqn peterGriffinFqn = Fqn.fromString("/griffin/peter");

cache.put(peterGriffinFqn, "isCartoonCharacter", Boolean.TRUE);
cache.put(peterGriffinFqn, "favouriteDrink", new Beer());

assertTrue(peterGriffin.get(peterGriffinFqn, "isCartoonCharacter"));
assertTrue(cache.getRootNode().hasChild(peterGriffinFqn));

cache.remove(peterGriffinFqn, "favouriteDrink");

assertNull(cache.get(peterGriffinFqn, "favouriteDrink");

cache.removeNode(peterGriffinFqn);

assertFalse(cache.getRootNode().hasChild(peterGriffinFqn));
```

2.4. The `Fqn` Class

The previous section used the `Fqn` class in its examples; now let's learn a bit more about that class.

A Fully Qualified Name (Fqn) encapsulates a list of names which represent a path to a particular location in the cache's tree structure. The elements in the list are typically `String` s but can be any `Object` or a mix of different types.

This path can be absolute (i.e., relative to the root node), or relative to any node in the cache. Reading the documentation on each API call that makes use of `Fqn` will tell you whether the API expects a relative or absolute `Fqn` .

The `Fqn` class provides are variety of constructors; see the javadoc for all the possibilities. The following illustrates the most commonly used approaches to creating an Fqn:

```
// Create an Fqn pointing to node 'Joe' under parent node 'Smith'
// under the 'people' section of the tree

// Parse it from a String
Fqn<String> abc = Fqn.fromString("/people/Smith/Joe/");

// Build it directly. A bit more efficient to construct than parsing
String[] strings = new String[] { "people", "Smith", "Joe" };
Fqn<String> abc = new Fqn<String>(strings);

// Here we want to use types other than String
Object[] objs = new Object[]{ "accounts", "NY", new Integer(12345) };
Fqn<Object> acctFqn = new Fqn<Object>(objs);
```

Note that

```
Fqn<String> f = new Fqn<String>("/a/b/c");
```

is *not* the same as

```
Fqn<String> f = Fqn.fromString("/a/b/c");
```

The former will result in an Fqn with a single element, called "/a/b/c" which hangs directly under the cache root. The latter will result in a 3 element Fqn, where "c" idicates a child of "b", which is a child of "a", and "a" hangs off the cache root. Another way to look at it is that the "/" separarator is only parsed when it forms part of a String passed in to `Fqn.fromString()` and not otherwise.

The JBoss Cache API in the 1.x releases included many overloaded convenience methods that took a string in the "/a/b/c" format in place of an `Fqn` . In the interests of API simplicity, no such convenience methods are available in the JBC 2.x API.

2.5. Stopping and Destroying the Cache

It is good practice to stop and destroy your cache when you are done using it, particularly if it is a clustered cache and has thus used a JGroups channel. Stopping and destroying a cache ensures resources like the JGroups channel are properly cleaned up.

```
cache.stop();
cache.destroy();
```

Not also that a cache that has had `stop()` invoked on it can be started again with a new call to `start()`. Similarly, a cache that has had `destroy()` invoked on it can be created again with a new call to `create()` (and then started again with a `start()` call).

2.6. Cache Modes

Although technically not part of the API, the *mode* in which the cache is configured to operate affects the cluster-wide behavior of any `put` or `remove` operation, so we'll briefly mention the various modes here.

JBoss Cache modes are denoted by the `org.jboss.cache.config.Configuration.CacheMode` enumeration. They consist of:

- *LOCAL* - local, non-clustered cache. Local caches don't join a cluster and don't communicate with other caches in a cluster. Therefore their contents don't need to be Serializable; however, we recommend making them Serializable, allowing you the flexibility to change the cache mode at any time.
- *REPL_SYNC* - synchronous replication. Replicated caches replicate all changes to the other caches in the cluster. Synchronous replication means that changes are replicated and the caller blocks until replication acknowledgements are received.
- *REPL_ASYNC* - asynchronous replication. Similar to REPL_SYNC above, replicated caches replicate all changes to the other caches in the cluster. Being asynchronous, the caller does not block until replication acknowledgements are received.
- *INVALIDATION_SYNC* - if a cache is configured for invalidation rather than replication, every time data is changed in a cache other caches in the cluster receive a message informing them that their data is now stale and should be evicted from memory. This reduces replication overhead while still being able to invalidate stale data on remote caches.
- *INVALIDATION_ASYNC* - as above, except this invalidation mode causes invalidation messages to be broadcast asynchronously.

See the chapter on Clustering for more details on how the cache's mode affects behavior. See the chapter on Configuration for info on how to configure things like the cache's mode.

2.7. Adding a CacheListener

The `@org.jboss.cache.notifications.annotation.CacheListener` annotation is a convenient mechanism for receiving notifications from a cache about events that happen in the cache.

Classes annotated with `@CacheListener` need to be public classes. In addition, the class needs to have one or more methods annotated with one of the method-level annotations (in the `org.jboss.cache.notifications.annotation` package). Methods annotated as such need to be public, have a void return type, and accept a single parameter of type `org.jboss.cache.notifications.event.Event` or one of it's subtypes.

- `@CacheStarted` - methods annotated such receive a notification when the cache is started. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.CacheStartedEvent`.
- `@CacheStopped` - methods annotated such receive a notification when the cache is stopped. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.CacheStoppedEvent`.
- `@NodeCreated` - methods annotated such receive a notification when a node is created. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeCreatedEvent`.
- `@NodeRemoved` - methods annotated such receive a notification when a node is removed. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeRemovedEvent`.
- `@NodeModified` - methods annotated such receive a notification when a node is modified. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeModifiedEvent`.
- `@NodeMoved` - methods annotated such receive a notification when a node is moved. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeMovedEvent`.
- `@NodeVisited` - methods annotated such receive a notification when a node is started. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeVisitedEvent`.
- `@NodeLoaded` - methods annotated such receive a notification when a node is loaded from a `CacheLoader`. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeLoadedEvent`.
- `@NodeEvicted` - methods annotated such receive a notification when a node is evicted from memory. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeEvictedEvent`.
- `@NodeActivated` - methods annotated such receive a notification when a node is activated. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodeActivatedEvent`.
- `@NodePassivated` - methods annotated such receive a notification when a node is passivated. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.NodePassivatedEvent`.

- `@TransactionRegistered` - methods annotated such receive a notification when the cache registers a `javax.transaction.Synchronization` with a registered transaction manager. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.TransactionRegisteredEvent`.
- `@TransactionCompleted` - methods annotated such receive a notification when the cache receives a commit or rollback call from a registered transaction manager. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.TransactionCompletedEvent`.
- `@ViewChanged` - methods annotated such receive a notification when the group structure of the cluster changes. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.ViewChangedEvent`.
- `@CacheBlocked` - methods annotated such receive a notification when the cluster requests that cache operations are blocked for a state transfer event. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.CacheBlockedEvent`.
- `@CacheUnblocked` - methods annotated such receive a notification when the cluster requests that cache operations are unblocked after a state transfer event. Methods need to accept a parameter type which is assignable from `org.jboss.cache.notifications.event.CacheUnblockedEvent`.

Refer to the javadocs on the annotations as well as the `Event` subtypes for details of what is passed in to your method, and when.

Example:

```
@CacheListener
public class MyListener
{

   @CacheStarted
   @CacheStopped
   public void cacheStartStopEvent(Event e)
   {
      switch (e.getType())
      {
         case Event.Type.CACHE_STARTED:
            System.out.println("Cache has started");
            break;
         case Event.Type.CACHE_STOPPED:
            System.out.println("Cache has stopped");
            break;
      }
   }

   @NodeCreated
   @NodeRemoved
   @NodeVisited
   @NodeModified
   @NodeMoved
   public void logNodeEvent(NodeEvent ne)
```

```
    {
       log("An event on node " + ne.getFqn() + " has occured");
    }
 }
```

2.8. Using Cache Loaders

Cache loaders are an important part of JBoss Cache. They allow persistence of nodes to disk or to remote cache clusters, and allow for passivation when caches run out of memory. In addition, cache loaders allow JBoss Cache to perform 'warm starts', where in-memory state can be preloaded from persistent storage. JBoss Cache ships with a number of cache loader implementations.

- `org.jboss.cache.loader.FileCacheLoader` - a basic, filesystem based cache loader that persists data to disk. Non-transactional and not very performant, but a very simple solution. Used mainly for testing and not recommended for production use.
- `org.jboss.cache.loader.JDBCCacheLoader` - uses a JDBC connection to store data. Connections could be created and maintained in an internal pool (uses the c3p0 pooling library) or from a configured DataSource. The database this CacheLoader connects to could be local or remotely located.
- `org.jboss.cache.loader.BdbjeCacheLoader` - uses Oracle's BerkeleyDB file-based transactional database to persist data. Transactional and very performant, but potentially restrictive license.
- `org.jboss.cache.loader.JdbmCacheLoader` - an upcoming open source alternative to the BerkeleyDB.
- `org.jboss.cache.loader.tcp.TcpCacheLoader` - uses a TCP socket to "persist" data to a remote cluster, using a "far cache" pattern. [1]
- `org.jboss.cache.loader.ClusteredCacheLoader` - used as a "read-only" CacheLoader, where other nodes in the cluster are queried for state.

These CacheLoaders, along with advanced aspects and tuning issues, are discussed in the chapter dedicated to CacheLoaders .

2.9. Using Eviction Policies

Eviction policies are the counterpart to CacheLoaders. They are necessary to make sure the cache does not run out of memory and when the cache starts to fill, the eviction algorithm running in a separate thread offloads in-memory state to the CacheLoader and frees up memory. Eviction policies can be configured on a per-region basis, so different subtrees in the cache could have different eviction preferences. JBoss Cache ships with several eviction policies:

- `org.jboss.cache.eviction.LRUPolicy` - an eviction policy that evicts the least recently used nodes when thresholds are hit.
- `org.jboss.cache.eviction.LFUPolicy` - an eviction policy that evicts the least frequently used nodes when thresholds are hit.

- `org.jboss.cache.eviction.MRUPolicy` - an eviction policy that evicts the most recently used nodes when thresholds are hit.
- `org.jboss.cache.eviction.FIFOPolicy` - an eviction policy that creates a first-in-first-out queue and evicts the oldest nodes when thresholds are hit.
- `org.jboss.cache.eviction.ExpirationPolicy` - an eviction policy that selects nodes for eviction based on an expiry time each node is configured with.
- `org.jboss.cache.eviction.ElementSizePolicy` - an eviction policy that selects nodes for eviction based on the number of key/value pairs held in the node.

Detailed configuration and implementing custom eviction policies are discussed in the chapter dedicated to eviction policies. .

3

Configuration

3.1. Configuration Overview

The `org.jboss.cache.config.Configuration` class (along with its component parts) is a Java Bean that encapsulates the configuration of the `Cache` and all of its architectural elements (cache loaders, evictions policies, etc.)

The `Configuration` exposes numerous properties which are summarized in the configuration reference section of this book and many of which are discussed in later chapters. Any time you see a configuration option discussed in this book, you can assume that the `Configuration` class or one of its component parts exposes a simple property setter/getter for that configuration option.

3.2. Creating a `Configuration`

As discussed in the User API section , before a `Cache` can be created, the `CacheFactory` must be provided with a `Configuration` object or with a file name or input stream to use to parse a `Configuration` from XML. The following sections describe how to accomplish this.

3.2.1. Parsing an XML-based Configuration File

The most convenient way to configure JBoss Cache is via an XML file. The JBoss Cache distribution ships with a number of configuration files for common use cases. It is recommended that these files be used as a starting point, and tweaked to meet specific needs.

Here is a simple example configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ===================================================================== -->
<!--                                                                       -->
<!--  Sample JBoss Cache Service Configuration                             -->
<!--                                                                       -->
<!-- ===================================================================== -->

<server>

   <mbean code="org.jboss.cache.jmx.CacheJmxWrapper" name="jboss.cache:service=Cache">

      <!-- Configure the TransactionManager -->
      <attribute name="TransactionManagerLookupClass">
```

```
         org.jboss.cache.transaction.GenericTransactionManagerLookup
      </attribute>

      <!-- Node locking level : SERIALIZABLE
                               REPEATABLE_READ (default)
                               READ_COMMITTED
                               READ_UNCOMMITTED
                               NONE              -->
      <attribute name="IsolationLevel">READ_COMMITTED</attribute>

      <!-- Lock parent before doing node additions/removes -->
      <attribute name="LockParentForChildInsertRemove">true</attribute>

      <!-- Valid modes are LOCAL (default)
                           REPL_ASYNC
                           REPL_SYNC
                           INVALIDATION_ASYNC
                           INVALIDATION_SYNC   -->
      <attribute name="CacheMode">LOCAL</attribute>

      <!-- Max number of milliseconds to wait for a lock acquisition -->
      <attribute name="LockAcquisitionTimeout">15000</attribute>


      <!-- Specific eviction policy configurations. This is LRU -->
      <attribute name="EvictionConfig">
         <config>
            <attribute name="wakeUpIntervalSeconds">5</attribute>
            <attribute name="policyClass">org.jboss.cache.eviction.LRUPolicy</attribute>

            <!-- Cache wide default -->
            <region name="/_default_">
               <attribute name="maxNodes">5000</attribute>
               <attribute name="timeToLiveSeconds">1000</attribute>
            </region>
         </config>
      </attribute>
   </mbean>
</server>
```

Another, more complete, sample XML file is included in the configuration reference section of this book, along with a handy look-up table explaining the various options.

For historical reasons, the format of the JBoss Cache configuraton file follows that of a JBoss AS Service Archive (SAR) deployment descriptor (and still can be used as such inside JBoss AS). Because of this dual usage, you may see elements in some configuration files (such as `depends` or `classpath`) that are not relevant outside JBoss AS. These can safely be ignored.

Here's how you tell the `CacheFactory` to create and start a cache by finding and parsing a configuration file on the classpath:

```
          CacheFactory factory = DefaultCacheFactory.getInstance();
```

```
Cache cache = factory.createCache("cache-configuration.xml");
```

3.2.2. Programmatic Configuration

In addition to the XML-based configuration above, the `Configuration` can be built up programatically, using the simple property mutators exposed by `Configuration` and its components. When constructed, the `Configuration` object is preset with JBoss Cache defaults and can even be used as-is for a quick start.

Following is an example of programatically creating a `Configuration` configured to match the one produced by the XML example above, and then using it to create a `Cache` :

```
Configuration config = new Configuration();
String tmlc = GenericTransactionManagerLookup.class.getName();
config.setTransactionManagerLookupClass(tmlc);
config.setIsolationLevel(IsolationLevel.READ_COMMITTED);
config.setCacheMode(CacheMode.LOCAL);
config.setLockParentForChildInsertRemove(true);
config.setLockAcquisitionTimeout(15000);

EvictionConfig ec = new EvictionConfig();
ec.setWakeupIntervalSeconds(5);
ec.setDefaultEvictionPolicyClass(LRUPolicy.class.getName());

EvictionRegionConfig erc = new EvictionRegionConfig();
erc.setRegionName("_default_");

LRUConfiguration lru = new LRUConfiguration();
lru.setMaxNodes(5000);
lru.setTimeToLiveSeconds(1000);

erc.setEvictionPolicyConfig(lru);

List<EvictionRegionConfig> ercs = new ArrayList<EvictionRegionConfig>();
ercs.add(erc);
ec.setEvictionRegionConfigs(erc);

config.setEvictionConfig(ec);

CacheFactory factory = DefaultCacheFactory.getInstance();
Cache cache = factory.createCache(config);
```

Even the above fairly simple configuration is pretty tedious programming; hence the preferred use of XML-based configuration. However, if your application requires it, there is no reason not to use XML-based configuration for most of the attributes, and then access the `Configuration` object to programatically change a few items from the defaults, add an eviction region, etc.

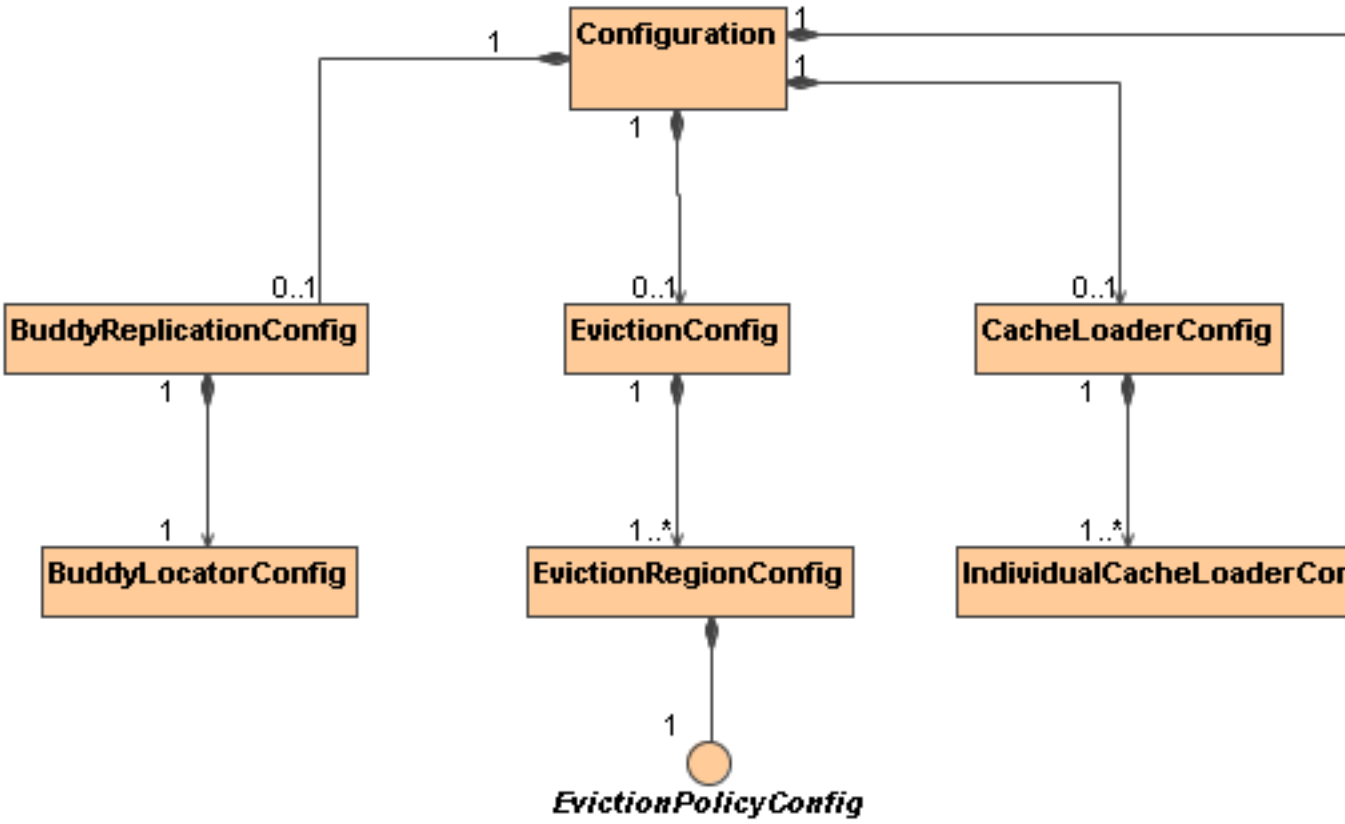
Note that configuration values may not be changed programmatically when a cache is running, except those annotated as `@Dynamic` . Dynamic properties are also marked as such in the configuration reference table. Attempting to change a non-dynamic property will result in a `ConfigurationException` .

3.2.3. Using an IOC Framework

The `Configuration` class and its component parts are all Java Beans that expose all config elements via simple setters and getters. Therefore, any good IOC framework should be able to build up a `Configuration` from an XML file in the framework's own format. See the deployment via the JBoss micrcontainer section for an example of this.

3.3. Composition of a `Configuration` Object

A `Configuration` is composed of a number of subobjects:



Following is a brief overview of the components of a `Configuration` . See the javadoc and the linked chapters in this book for a more complete explanation of the configurations associated with each component.

- `Configuration` : top level object in the hierarchy; exposes the configuration properties listed in the configuration reference section of this book.
- `BuddyReplicationConfig` : only relevant if buddy replication is used. General buddy replication configuration options. Must include a:
- `BuddyLocatorConfig` : implementation-specific configuration object for the `BuddyLocator` implementation being used. What configuration elements are exposed depends on the needs of the `BuddyLocator` implementation.

- `EvictionConfig` : only relevant if eviction is used. General eviction configuration options. Must include at least one:
- `EvictionRegionConfig` : one for each eviction region; names the region, etc. Must include a:
- `EvictionPolicyConfig` : implementation-specific configuration object for the `EvictionPolicy` implementation being used. What configuration elements are exposed depends on the needs of the `EvictionPolicy` implementation.
- `CacheLoaderConfig` : only relevant if a cache loader is used. General cache loader configuration options. Must include at least one:
- `IndividualCacheLoaderConfig` : implementation-specific configuration object for the `CacheLoader` implementation being used. What configuration elements are exposed depends on the needs of the `CacheLoader` implementation.
- `RuntimeConfig` : exposes to cache clients certain information about the cache's runtime environment (e.g. membership in buddy replication groups if buddy replication is used.) Also allows direct injection into the cache of needed external services like a JTA `TransactionManager` or a JGroups `ChannelFactory` .

3.4. Dynamic Reconfiguration

Dynamically changing the configuration of *some* options while the cache is running is supported, by programmatically obtaining the `Configuration` object from the running cache and changing values. E.g.,

```
Configuration liveConfig = cache.getConfiguration();
liveConfig.setLockAcquisitionTimeout(2000);
```

A complete listing of which options may be changed dynamically is in the configuration reference section. An `org.jboss.cache.config.ConfigurationException` will be thrown if you attempt to change a setting that is not dynamic.

3.5. Overriding the Configuration Via the Option API

The Option API allows you to override certain behaviours of the cache on a per invocation basis. This involves creating an instance of `org.jboss.cache.config.Option` , setting the options you wish to override on the `Option` object and passing it in the `InvocationContext` before invoking your method on the cache.

E.g., to override the default node versioning used with optimistic locking:

```
DataVersion v = new MyCustomDataVersion();
cache.getInvocationContext().getOptionOverrides().setDataVersion(v);
Node ch = cache.getRoot().addChild(Fqn.fromString("/a/b/c"));
```

E.g., to suppress replication of a put call in a REPL_SYNC cache:

```
Node node = cache.getChild(Fqn.fromString("/a/b/c"));
cache.getInvocationContext().getOptionOverrides().setLocalOnly(true);
node.put("localCounter", new Integer(2));
```

See the javadocs on the `Option` class for details on the options available.

4

Deploying JBoss Cache

4.1. Standalone Use / Programatic Deployment

When used in a standalone Java program, all that needs to be done is to instantiate the cache using the `CacheFactory` and a `Configuration` instance or an XML file, as discussed in the User API and Configuration chapters.

The same techniques can be used when an application running in an application server wishes to programatically deploy a cache rather than relying on an application server's deployment features. An example of this would be a webapp deploying a cache via a `javax.servlet.ServletContextListener` .

If, after deploying your cache you wish to expose a management interface to it in JMX, see the section on Programatic Registration in JMX .

4.2. JMX-Based Deployment in JBoss AS (JBoss AS 5.x and 4.x)

If JBoss Cache is run in JBoss AS then the cache can be deployed as an MBean simply by copying a standard cache configuration file to the server's `deploy` directory. The standard format of JBoss Cache's standard XML configuration file (as shown in the Configuration Reference) is the same as a JBoss AS MBean deployment descriptor, so the AS's SAR Deployer has no trouble handling it. Also, you don't have to place the configuration file directly in `deploy` ; you can package it along with other services or JEE components in a SAR or EAR.

In AS 5, if you're using a server config based on the standard `all` config, then that's all you need to do; all required jars will be on the classpath. Otherwise, you will need to ensure `jbosscache.jar` and `jgroups-all.jar` are on the classpath. You may need to add other jars if you're using things like `JdbmCacheLoader` . The simplest way to do this is to copy the jars from the JBoss Cache distribution's `lib` directory to the server config's `lib` directory. You could also package the jars with the configuration file in Service Archive (.sar) file or an EAR.

It is possible to deploy a JBoss Cache 2.0 instance in JBoss AS 4.x (at least in 4.2.0.GA; other AS releases are completely untested). However, the significant API changes between the JBoss Cache 2.x and 1.x releases mean none of the standard AS 4.x clustering services (e.g. http session replication) that rely on JBoss Cache will work with JBoss Cache 2.x. Also, be aware that usage of JBoss Cache 2.x in AS 4.x is not something the JBoss Cache developers are making any significant effort to test, so be sure to test your application well (which of course you're doing anyway.)

Note in the example the value of the `mbean` element's `code` attribute: `org.jboss.cache.jmx.CacheJmxWrapper` . This is the class JBoss Cache uses to handle JMX integration;

the `Cache` itself does not expose an MBean interface. See the JBoss Cache MBeans section for more on the `CacheJmxWrapper` .

Once your cache is deployed, in order to use it with an in-VM client such as a servlet, a JMX proxy can be used to get a reference to the cache:

```
MBeanServer server = MBeanServerLocator.locateJBoss();
ObjectName on = new ObjectName("jboss.cache:service=Cache");
CacheJmxWrapperMBean cacheWrapper =
  (CacheJmxWrapperMBean) MBeanServerInvocationHandler.newProxyInstance(server, on,
                                        CacheJmxWrapperMBean.class, false);
Cache cache = cacheWrapper.getCache();
Node root = cache.getRoot(); // etc etc
```

The MBeanServerLocator class is a helper to find the (only) JBoss MBean server inside the current JVM. The `javax.management.MBeanServerInvocationHandler` class' `newProxyInstance` method creates a dynamic proxy implementing the given interface and uses JMX to dynamically dispatch methods invoked against the generated interface to the MBean. The name used to look up the MBean is the same as defined in the cache's configuration file.

Once the proxy to the `CacheJmxWrapper` is obtained, the `getCache()` will return a reference to the `Cache` itself.

4.3. Via JBoss Microcontainer (JBoss AS 5.x)

Beginning with AS 5, JBoss AS also supports deployment of POJO services via deployment of a file whose name ends with `-beans.xml` . A POJO service is one whose implementation is via a "Plain Old Java Object", meaning a simple java bean that isn't required to implement any special interfaces or extend any particular superclass. A `Cache` is a POJO service, and all the components in a `Configuration` are also POJOS, so deploying a cache in this way is a natural step.

Deployment of the cache is done using the JBoss Microcontainer that forms the core of JBoss AS. JBoss Microcontainer is a sophisticated IOC framework (similar to Spring). A `-beans.xml` file is basically a descriptor that tells the IOC framework how to assemble the various beans that make up a POJO service.

The rules for how to deploy the file, how to package it, how to ensure the required jars are on the classpath, etc. are the same as for a JMX-based deployment .

Following is an example `-beans.xml` file. If you look in the `server/all/deploy` directory of an AS 5 installation, you can find several more examples.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
```

```
<bean name="ExampleCacheConfig"
   class="org.jboss.cache.config.Configuration">

   <!-- Externally injected services -->
   <property name="runtimeConfig">
      <bean name="ExampleCacheRuntimeConfig" class="org.jboss.cache.config.RuntimeConfig">
         <property name="transactionManager">
            <inject bean="jboss:service=TransactionManager"
                    property="TransactionManager"/>
         </property>
         <property name="muxChannelFactory"><inject bean="JChannelFactory"/></property>
      </bean>
   </property>

   <property name="multiplexerStack">udp</property>

   <property name="clusterName">Example-EntityCache</property>

   <!--
           Node locking level : SERIALIZABLE
                                REPEATABLE_READ (default)
                                READ_COMMITTED
                                READ_UNCOMMITTED
                                NONE
   -->
   <property name="isolationLevel">REPEATABLE_READ</property>

   <!--     Valid modes are LOCAL
                            REPL_ASYNC
                            REPL_SYNC
   -->
   <property name="cacheMode">REPL_SYNC</property>

   <!--  The max amount of time (in milliseconds) we wait until the
         initial state (ie. the contents of the cache) are retrieved from
         existing members in a clustered environment
   -->
   <property name="initialStateRetrievalTimeout">15000</property>

   <!--    Number of milliseconds to wait until all responses for a
           synchronous call have been received.
   -->
   <property name="syncReplTimeout">20000</property>

   <!--  Max number of milliseconds to wait for a lock acquisition -->
   <property name="lockAcquisitionTimeout">15000</property>

   <property name="exposeManagementStatistics">true</property>

   <!-- Must be true if any entity deployment uses a scoped classloader -->
   <property name="useRegionBasedMarshalling">true</property>
   <!-- Must match the value of "useRegionBasedMarshalling" -->
   <property name="inactiveOnStartup">true</property>

   <!--  Specific eviction policy configurations. This is LRU -->
   <property name="evictionConfig">
     <bean name="ExampleEvictionConfig"
```

```
            class="org.jboss.cache.config.EvictionConfig">
          <property name="defaultEvictionPolicyClass">
             org.jboss.cache.eviction.LRUPolicy
          </property>
           <property name="wakeupIntervalSeconds">5</property>
           <property name="evictionRegionConfigs">
              <list>
                 <bean name="ExampleDefaultEvictionRegionConfig"
                       class="org.jboss.cache.config.EvictionRegionConfig">
                    <property name="regionName">/_default_</property>
                 <property name="evictionPolicyConfig">
                       <bean name="ExampleDefaultLRUConfig"
                             class="org.jboss.cache.eviction.LRUConfiguration">
                          <property name="maxNodes">5000</property>
                          <property name="timeToLiveSeconds">1000</property>
                       </bean>
                    </property>
                 </bean>
              </list>
           </property>
        </bean>
     </property>

  </bean>

  <!-- Factory to build the Cache. -->
  <bean name="DefaultCacheFactory" class="org.jboss.cache.DefaultCacheFactory">
     <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                  factoryMethod="getInstance"/>
  </bean>

  <!-- The cache itself -->
  <bean name="ExampleCache" class="org.jboss.cache.CacheImpl">

     <constructor factoryMethod="createCache">
         <factory bean="DefaultCacheFactory"/>
         <parameter><inject bean="ExampleCacheConfig"/></parameter>
         <parameter>false</false>
     </constructor>

  </bean>

</deployment>
```

See the JBoss Microcontainer documentation [1] for details on the above syntax. Basically, each `bean` element represents an object; most going to create a `Configuration` and its constituent parts .

An interesting thing to note in the above example is the use of the `RuntimeConfig` object. External resources like a `TransactionManager` and a JGroups `ChannelFactory` that are visible to the microcontainer are dependency injected into the `RuntimeConfig` . The assumption here is that in some other deployment descriptor in the AS, the referenced beans have been described.

[1] http://labs.jboss.com/jbossmc/docs

4.4. Binding to JNDI in JBoss AS

With the 1.x JBoss Cache releases, a proxy to the cache could be bound into JBoss AS's JNDI tree using the AS's `JRMPProxyFactory` service. With JBoss Cache 2.x, this no longer works. An alternative way of doing a similar thing with a POJO (i.e. non-JMX-based) service like a `Cache` is under development by the JBoss AS team [2] . That feature is not available as of the time of this writing, although it will be completed before AS 5.0.0.GA is released. We will add a wiki page describing how to use it once it becomes available.

4.5. Runtime Management Information

JBoss Cache includes JMX MBeans to expose cache functionality and provide statistics that can be used to analyze cache operations. JBoss Cache can also broadcast cache events as MBean notifications for handling via JMX monitoring tools.

4.5.1. JBoss Cache MBeans

JBoss Cache provides an MBean that can be registered with your environments JMX server to allow access to the cache instance via JMX. This MBean is the `org.jboss.cache.jmx.CacheJmxWrapper` . It is a StandardMBean, so it's MBean interface is `org.jboss.cache.jmx.CacheJmxWrapperMBean` . This MBean can be used to:

- Get a reference to the underlying `Cache` .
- Invoke create/start/stop/destroy lifecycle operations on the underlying `Cache` .
- Inspect various details about the cache's current state (number of nodes, lock information, etc.)
- See numerous details about the cache's configuration, and change those configuration items that can be changed when the cache has already been started.

See the `CacheJmxWrapperMBean` javadoc for more details.

It is important to note a significant architectural difference between JBoss Cache 1.x and 2.x. In 1.x, the old `TreeCache` class was itself an MBean, and essentially exposed the cache's entire API via JMX. In 2.x, JMX has been returned to it's fundamental role as a management layer. The `Cache` object itself is completely unaware of JMX; instead JMX functionality is added through a wrapper class designed for that purpose. Furthermore, the interface exposed through JMX has been limited to management functions; the general `Cache` API is no longer exposed through JMX. For example, it is no longer possible to invoke a cache `put` or `get` via the JMX interface.

If a `CacheJmxWrapper` is registered, JBoss Cache also provides MBeans for each interceptor configured in the cache's interceptor stack. These MBeans are used to capture and expose statistics related to cache operations. They are hierarchically associated with the `CacheJmxWrapper` MBean and have service names that reflect this relationship. For example, a replication interceptor MBean for the `jboss.cache:service=TomcatClusteringCache` instance will be accessible through the service named `jboss.cache:service=TomcatClusteringCache,cache-interceptor=ReplicationInterceptor` .

[2] http://jira.jboss.com/jira/browse/JBAS-4456

4.5.2. Registering the CacheJmxWrapper with the MBeanServer

The best way to ensure the `CacheJmxWrapper` is registered in JMX depends on how you are deploying your cache:

4.5.2.1. Programatic Registration

Simplest way to do this is to create your `Cache` and pass it to the `CacheJmxWrapper` constructor.

```
CacheFactory factory = DefaultCacheFactory.getInstance();
// Build but don't start the cache
// (although it would work OK if we started it)
Cache cache = factory.createCache("cache-configuration.xml", false);

CacheJmxWrapperMBean wrapper = new CacheJmxWrapper(cache);
MBeanServer server = getMBeanServer(); // however you do it
ObjectName on = new ObjectName("jboss.cache:service=TreeCache");
server.registerMBean(wrapper, on);

// Invoking lifecycle methods on the wrapper results
// in a call through to the cache
wrapper.create();
wrapper.start();

... use the cache

... on application shutdown

// Invoking lifecycle methods on the wrapper results
// in a call through to the cache
wrapper.stop();
wrapper.destroy();
```

Alternatively, build a `Configuration` object and pass it to the `CacheJmxWrapper` . The wrapper will construct the `Cache` :

```
Configuration config = buildConfiguration(); // whatever it does

CacheJmxWrapperMBean wrapper = new CacheJmxWrapper(config);
MBeanServer server = getMBeanServer(); // however you do it
ObjectName on = new ObjectName("jboss.cache:service=TreeCache");
server.registerMBean(wrapper, on);

// Call to wrapper.create() will build the Cache if one wasn't injected
wrapper.create();
wrapper.start();

// Now that it's built, created and started, get the cache from the wrapper
Cache cache = wrapper.getCache();

... use the cache
```

```
               ... on application shutdown

               wrapper.stop();
               wrapper.destroy();
```

4.5.2.2. JMX-Based Deployment in JBoss AS (JBoss AS 4.x and 5.x)

When you deploy your cache in JBoss AS using a -service.xml file , a `CacheJmxWrapper` is automatically registered. There is no need to do anything further. The `CacheJmxWrapper` is accessible from an MBean server through the service name specified in the cache configuration file's `mbean` element.

4.5.2.3. Via JBoss Microcontainer (JBoss AS 5.x)

`CacheJmxWrapper` is a POJO, so the microcontainer has no problem creating one. The trick is getting it to register your bean in JMX. This can be done by specifying the `org.jboss.aop.microcontainer.aspects.jmx.JMX` annotation on the `CacheJmxWrapper` bean:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      ... build up the Configuration

   </bean>

   <!-- Factory to build the Cache. -->
   <bean name="DefaultCacheFactory" class="org.jboss.cache.DefaultCacheFactory">
      <constructor factoryClass="org.jboss.cache.DefaultCacheFactory"
                   factoryMethod="getInstance"/>
   </bean>

   <!-- The cache itself -->
   <bean name="ExampleCache" class="org.jboss.cache.CacheImpl">

      <constructor factoryMethod="createnewInstance">
          <factory bean="DefaultCacheFactory"/>
          <parameter><inject bean="ExampleCacheConfig"/></parameter>
          <parameter>false</false>
      </constructor>

   </bean>

   <!-- JMX Management -->
   <bean name="ExampleCacheJmxWrapper" class="org.jboss.cache.jmx.CacheJmxWrapper">

      <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.cache:service=Exampl
                        exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
```

```
                        registerDirectly=true)</annotation>

      <constructor>
          <parameter><inject bean="ExampleCache"/></parameter>
      </constructor>

   </bean>

</deployment>
```

As discussed in the Programatic Registration section, `CacheJmxWrapper` can do the work of building, creating and starting the `Cache` if it is provided with a `Configuration` . With the microcontainer, this is the preferred approach, as it saves the boilerplate XML needed to create the `CacheFactory` :

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns="urn:jboss:bean-deployer:2.0">

   <!-- First we create a Configuration object for the cache -->
   <bean name="ExampleCacheConfig"
      class="org.jboss.cache.config.Configuration">

      ... build up the Configuration

   </bean>

   <bean name="ExampleCache" class="org.jboss.cache.jmx.CacheJmxWrapper">

      <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(name="jboss.cache:service=Exampl
                        exposedInterface=org.jboss.cache.jmx.CacheJmxWrapperMBean.class,
                        registerDirectly=true)</annotation>

      <constructor>
          <parameter><inject bean="ExampleCacheConfig"/></parameter>
      </constructor>

   </bean>

</deployment>
```

4.5.3. JBoss Cache Statistics

JBoss Cache captures statistics in its interceptors and exposes the statistics through interceptor MBeans. Gathering of statistics is enabled by default; this can be disabled for a specific cache instance through the `ExposeManagementStatistics` configuration attribute. Note that the majority of the statistics are provided by the `CacheMgmtInterceptor` , so this MBean is the most significant in this regard. If you want to disable all statistics for performance reasons, you set `ExposeManagementStatistics` to `false` as this will prevent the `CacheMgmtInterceptor` from being included in the cache's interceptor stack when the cache is started.

If a `CacheJmxWrapper` is registered with JMX, the wrapper also ensures that an MBean is registered in JMX for each interceptor that exposes statistics [3] . Management tools can then access those MBeans to examine the statistics. See the section in the JMX Reference chapter pertaining to the statistics that are made available via JMX.

The name under which the interceptor MBeans will be registered is derived by taking the `ObjectName` under which the `CacheJmxWrapper` is registered and adding a `cache-interceptor` attribute key whose value is the non-qualified name of the interceptor class. So, for example, if the `CacheJmxWrapper` were registered under `jboss.cache:service=TreeCache` , the name of the `CacheMgmtInterceptor` MBean would be `jboss.cache:service=TreeCache,cache-interceptor=CacheMgmtInterceptor` .

Each interceptor's MBean exposes a `StatisticsEnabled` attribute that can be used to disable maintenance of statistics for that interceptor. In addition, each interceptor MBean provides the following common operations and attributes.

- `dumpStatistics` - returns a `Map` containing the interceptor's attributes and values.
- `resetStatistics` - resets all statistics maintained by the interceptor.
- `setStatisticsEnabled(boolean)` - allows statistics to be disabled for a specific interceptor.

4.5.4. Receiving JMX Notifications

JBoss Cache users can register a listener to receive cache events described earlier in the User API chapter. Users can alternatively utilize the cache's management information infrastructure to receive these events via JMX notifications. Cache events are accessible as notifications by registering a `NotificationListener` for the `CacheJmxWrapper` .

See the section in the JMX Reference chapter pertaining to JMX notifications for a list of notifications that can be received through the `CacheJmxWrapper` .

The following is an example of how to programmatically receive cache notifications when running in a JBoss AS environment. In this example, the client uses a filter to specify which events are of interest.

```
        MyListener listener = new MyListener();
        NotificationFilterSupport filter = null;

        // get reference to MBean server
        Context ic = new InitialContext();
        MBeanServerConnection server = (MBeanServerConnection)ic.lookup("jmx/invoker/RMIAdaptor

        // get reference to CacheMgmtInterceptor MBean
        String cache_service = "jboss.cache:service=TomcatClusteringCache";
        ObjectName mgmt_name = new ObjectName(cache_service);
```

[3] Note that if the `CacheJmxWrapper` is not registered in JMX, the interceptor MBeans will not be registered either. The JBoss Cache 1.4 releases included code that would try to "discover" an `MBeanServer` and automatically register the interceptor MBeans with it. For JBoss Cache 2.x we decided that this sort of "discovery" of the JMX environment was beyond the proper scope of a caching library, so we removed this functionality.

```
// configure a filter to only receive node created and removed events
filter = new NotificationFilterSupport();
filter.disableAllTypes();
filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_CREATED);
filter.enableType(CacheNotificationBroadcaster.NOTIF_NODE_REMOVED);

// register the listener with a filter
// leave the filter null to receive all cache events
server.addNotificationListener(mgmt_name, listener, filter, null);

// ...

// on completion of processing, unregister the listener
server.removeNotificationListener(mgmt_name, listener, filter, null);
```

The following is the simple notification listener implementation used in the previous example.

```
private class MyListener implements NotificationListener, Serializable
{
   public void handleNotification(Notification notification, Object handback)
   {
      String message = notification.getMessage();
      String type = notification.getType();
      Object userData = notification.getUserData();

      System.out.println(type + ": " + message);

      if (userData == null)
      {
         System.out.println("notification data is null");
      }
      else if (userData instanceof String)
      {
         System.out.println("notification data: " + (String) userData);
      }
      else if (userData instanceof Object[])
      {
         Object[] ud = (Object[]) userData;
         for (Object data : ud)
         {
            System.out.println("notification data: " + data.toString());
         }
      }
      else
      {
         System.out.println("notification data class: " + userData.getClass().getName()
      }
   }
}
```

Note that the JBoss Cache management implementation only listens to cache events after a client registers to receive MBean notifications. As soon as no clients are registered for notifications, the MBean will remove itself as a cache listener.

4.5.5. Accessing Cache MBeans in a Standalone Environment

JBoss Cache MBeans are easily accessed when running cache instances in an application server that provides an MBean server interface such as JBoss JMX Console. Refer to your server documentation for instructions on how to access MBeans running in a server's MBean container.

In addition, though, JBoss Cache MBeans are also accessible when running in a non-server environment if the JVM is JDK 5.0 or later. When running a standalone cache in a JDK 5.0 environment, you can access the cache's MBeans as follows.

1. Set the system property `-Dcom.sun.management.jmxremote` when starting the JVM where the cache will run.

2. Once the JVM is running, start the JDK 5.0 `jconsole` utility, located in your JDK's `/bin` directory.

3. When the utility loads, you will be able to select your running JVM and connect to it. The JBoss Cache MBeans will be available on the MBeans panel.

Note that the `jconsole` utility will automatically register as a listener for cache notifications when connected to a JVM running JBoss Cache instances.

The following figure shows cache interceptor MBeans in `jconsole` . Cache statistics are displayed for the `CacheMgmtInterceptor` :

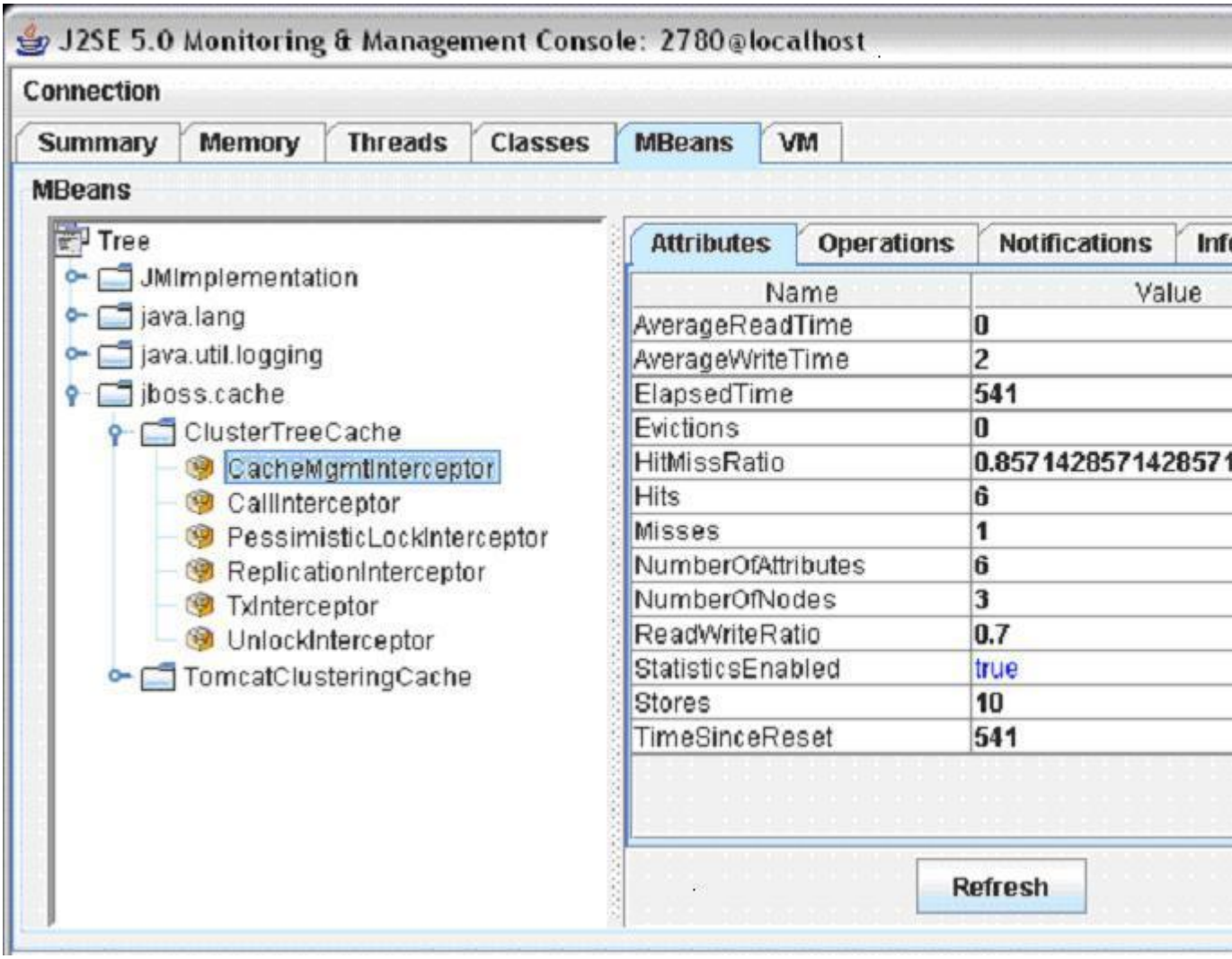


Figure 4.1. CacheMgmtInterceptor MBean in jconsole

5

Version Compatibility and Interoperability

Within a major version, releases of JBoss Cache are meant to be compatible and interoperable. Compatible in the sense that it should be possible to upgrade an application from one version to another by simply replacing the jars. Interoperable in the sense that if two different versions of JBoss Cache are used in the same cluster, they should be able to exchange replication and state transfer messages. Note however that interoperability requires use of the same JGroups version in all nodes in the cluster. In most cases, the version of JGroups used by a version of JBoss Cache can be upgraded.

As such, JBoss Cache 2.x.x is not API or binary compatible with prior 1.x.x versions. However, JBoss Cache 2.1.x will be API and binary compatible with 2.0.x.

A configuration attribute, `ReplicationVersion`, is available and is used to control the wire format of inter-cache communications. They can be wound back from more efficient and newer protocols to "compatible" versions when talking to older releases. This mechanism allows us to improve JBoss Cache by using more efficient wire formats while still providing a means to preserve interoperability.

5.1. Compatibility Matrix

A compatibility matrix [http://labs.jboss.com/portal/jbosscache/compatibility/index.html] is maintained on the JBoss Cache website, which contains information on different versions of JBoss Cache, JGroups and JBoss AS.

Part II. JBoss Cache Architecture

This section digs deeper into the JBoss Cache architecture, and is meant for developers wishing to extend or enhance JBoss Cache, write plugins or are just looking for detailed knowledge of how things work under the hood.

1

Architecture

6.1. Data Structures Within The Cache

A `Cache` consists of a collection of `Node` instances, organised in a tree structure. Each `Node` contains a `Map` which holds the data objects to be cached. It is important to note that the structure is a mathematical tree, and not a graph; each `Node` has one and only one parent, and the root node is denoted by the constant fully qualitied name, `Fqn.ROOT` .

The reason for organising nodes as such is to improve concurrent access to data and make replication and persistence more fine-grained.

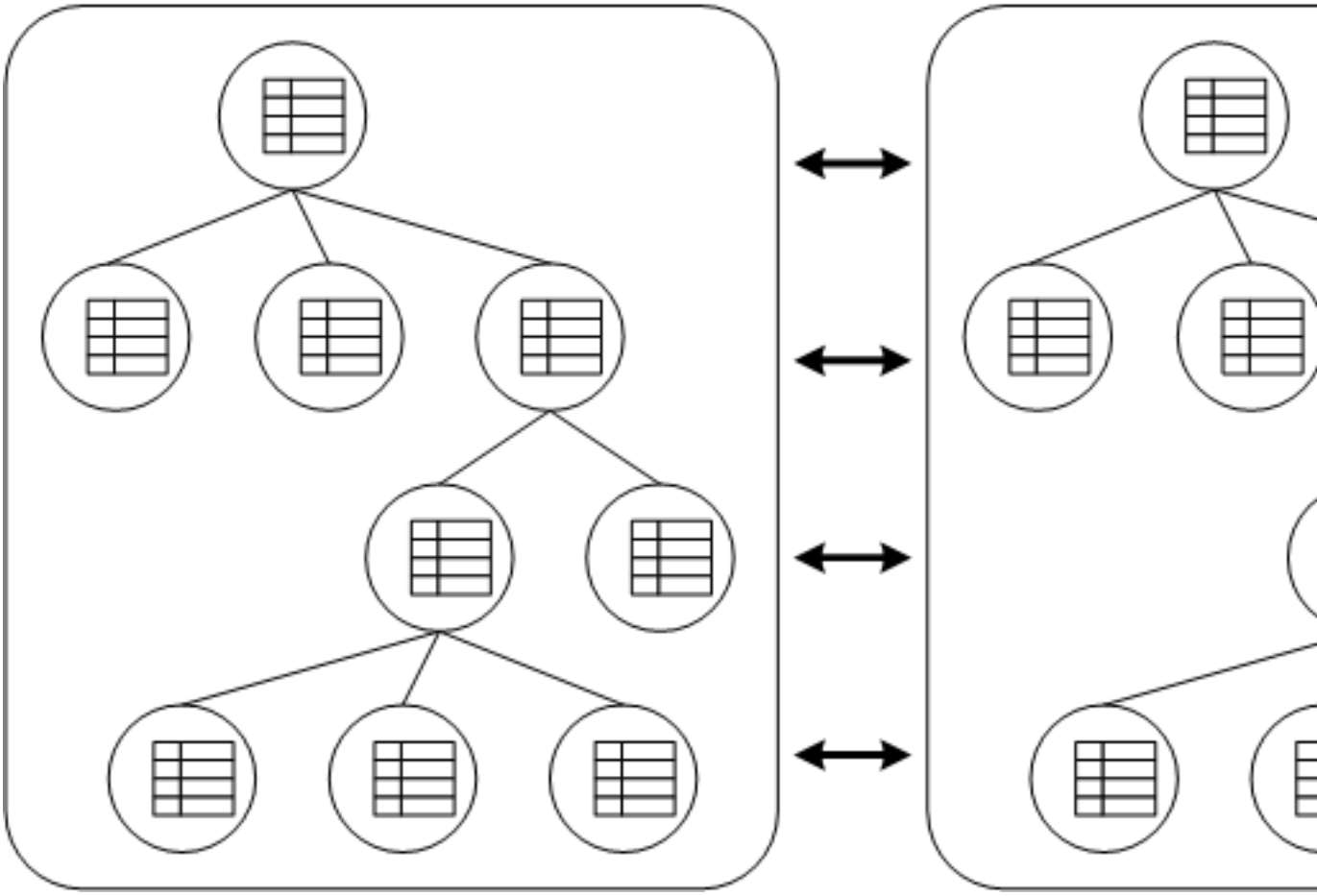


Figure 6.1. Data structured as a tree

In the diagram above, each box represents a JVM. You see 2 caches in separate JVMs, replicating data to each other. These VMs can be located on the same physical machine, or on 2 different machines connected

by a network link. The underlying group communication between networked nodes is done using JGroups [http://www.jgroups.org] .

Any modifications (see API chapter) in one cache instance will be replicated to the other cache. Naturally, you can have more than 2 caches in a cluster. Depending on the transactional settings, this replication will occur either after each modification or at the end of a transaction, at commit time. When a new cache is created, it can optionally acquire the contents from one of the existing caches on startup.

6.2. SPI Interfaces

In addition to `Cache` and `Node` interfaces, JBoss Cache exposes more powerful `CacheSPI` and `NodeSPI` interfaces, which offer more control over the internals of JBoss Cache. These interfaces are not intended for general use, but are designed for people who wish to extend and enhance JBoss Cache, or write custom `Interceptor` or `CacheLoader` instances.

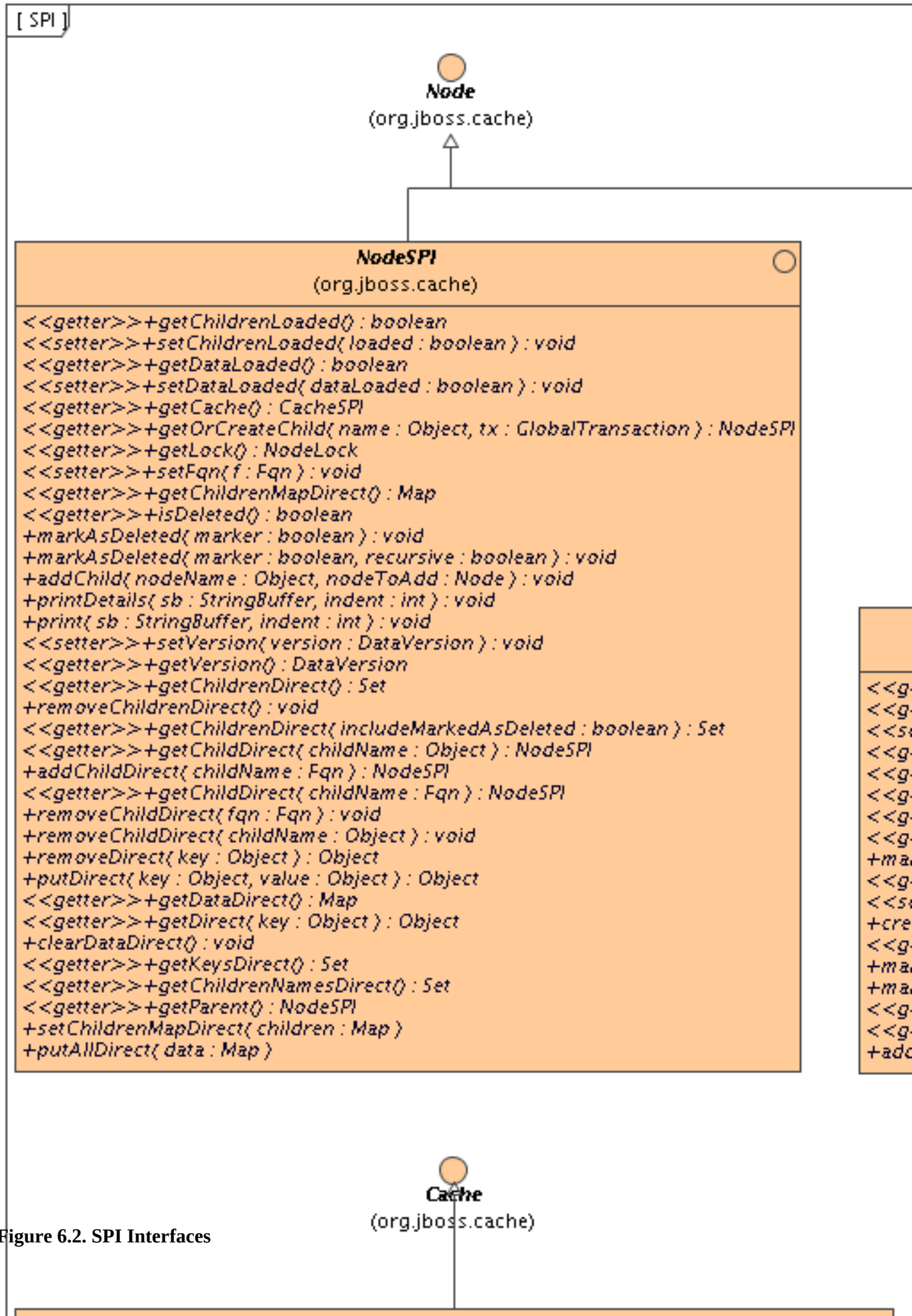


Figure 6.2. SPI Interfaces

The `CacheSPI` interface cannot be created, but is injected into `Interceptor` and `CacheLoader` implementations by the `setCache(CacheSPI cache)` methods on these interfaces. `CacheSPI` extends `Cache` so all the functionality of the basic API is made available.

Similarly, a `NodeSPI` interface cannot be created. Instead, one is obtained by performing operations on `CacheSPI` , obtained as above. For example, `Cache.getRoot() : Node` is overridden as `CacheSPI.getRoot() : NodeSPI` .

It is important to note that directly casting a `Cache` or `Node` to it's SPI counterpart is not recommended and is bad practice, since the inheritace of interfaces it is not a contract that is guaranteed to be upheld moving forward. The exposed public APIs, on the other hand, is guaranteed to be upheld.

6.3. Method Invocations On Nodes

Since the cache is essentially a collection of nodes, aspects such as clustering, persistence, eviction, etc. need to be applied to these nodes when operations are invoked on the cache as a whole or on individual nodes. To achieve this in a clean, modular and extensible manner, an interceptor chain is used. The chain is built up of a series of interceptors, each one adding an aspect or particular functionality. The chain is built when the cache is created, based on the configuration used.

It is important to note that the `NodeSPI` offers some methods (such as the `xxxDirect()` method family) that operate on a node directly without passing through the interceptor stack. Plugin authors should note that using such methods will affect the aspects of the cache that may need to be applied, such as locking, replication, etc. Basically, don't use such methods unless you *really* know what you're doing!

6.3.1. Interceptors

An `Interceptor` is an abstract class, several of which comprise an interceptor chain. It exposes an `invoke()` method, which must be overridden by implementing classes to add behaviour to a call before passing the call down the chain by calling `super.invoke()` .

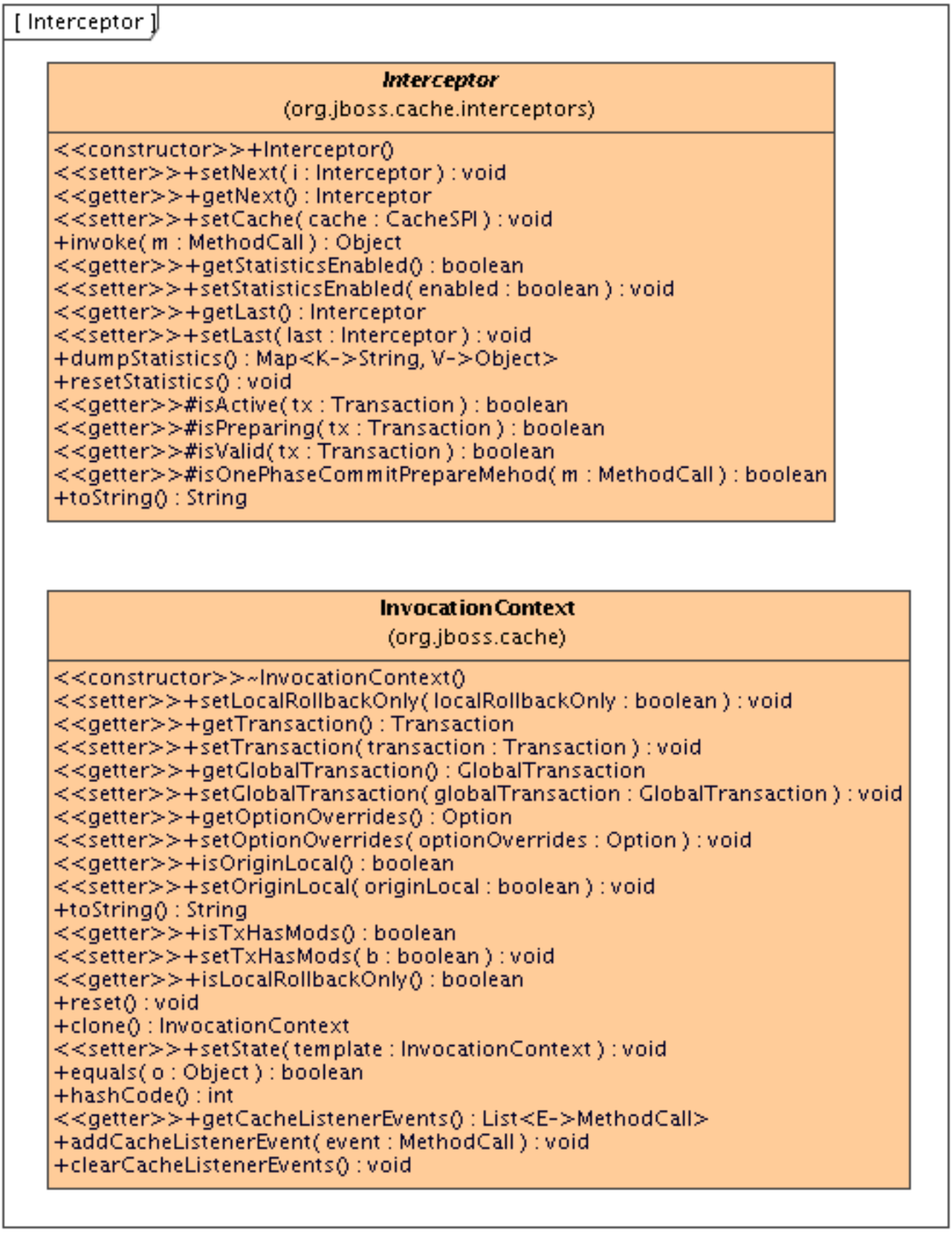


Figure 6.3. SPI Interfaces

JBoss Cache ships with several interceptors, representing different configuration options, some of which are:

- `TxInterceptor` - looks for ongoing transactions and registers with transaction managers to participate in synchronization events
- `ReplicationInterceptor` - replicates state across a cluster using a JGroups channel
- `CacheLoaderInterceptor` - loads data from a persistent store if the data requested is not available in memory

The interceptor chain configured for your cache instance can be obtained and inspected by calling `CacheSPI.getInterceptorChain()`, which returns an ordered `List` of interceptors.

6.3.1.1. Writing Custom Interceptors

Custom interceptors to add specific aspects or features can be written by extending `Interceptor` and overriding `invoke()`. The custom interceptor will need to be added to the interceptor chain by using the `CacheSPI.addInterceptor()` method.

Adding custom interceptors via XML is not supported at this time.

6.3.2. MethodCalls

`org.jboss.cache.marshall.MethodCall` is a class that encapsulates a `java.lang.reflection.Method` and an `Object[]` representing the method's arguments. It is an extension of the `org.jgroups.blocks.MethodCall` class, that adds a mechanism for identifying known methods using magic numbers and method ids, which makes marshalling and unmarshalling more efficient and performant.

This is central to the `Interceptor` architecture, and is the only parameter passed in to `Interceptor.invoke()`.

6.3.3. InvocationContexts

`InvocationContext` holds intermediate state for the duration of a single invocation, and is set up and destroyed by the `InvocationContextInterceptor` which sits at the start of the chain.

`InvocationContext`, as its name implies, holds contextual information associated with a single cache method invocation. Contextual information includes associated `javax.transaction.Transaction` or `org.jboss.cache.transaction.GlobalTransaction`, method invocation origin (`InvocationContext.isOriginLocal()`) as well as `Option` overrides.

The `InvocationContext` can be obtained by calling `Cache.getInvocationContext()`.

6.4. Managers For Subsystems

Some aspects and functionality is shared by more than a single interceptor. Some of these have been encapsulated into managers, for use by various interceptors, and are made available by the `CacheSPI` interface.

6.4.1. RpcManager

This class is responsible for calls made via the JGroups channel for all RPC calls to remote caches, and encapsulates the JGroups channel used.

6.4.2. BuddyManager

This class manages buddy groups and invokes group organisation remote calls to organise a cluster of caches into smaller sub-groups.

6.4.3. CacheLoaderManager

Sets up and configures cache loaders. This class wraps individual `CacheLoader` instances in delegating classes, such as `SingletonStoreCacheLoader` or `AsyncCacheLoader` , or may add the `CacheLoader` to a chain using the `ChainingCacheLoader` .

6.5. Marshalling And Wire Formats

Early versions of JBoss Cache simply wrote cached data to the network by writing to an `ObjectOutputStream` during replication. Over various releases in the JBoss Cache 1.x.x series this approach was gradually deprecated in favour of a more mature marshalling framework. In the JBoss Cache 2.x.x series, this is the only officially supported and recommended mechanism for writing objects to datastreams.

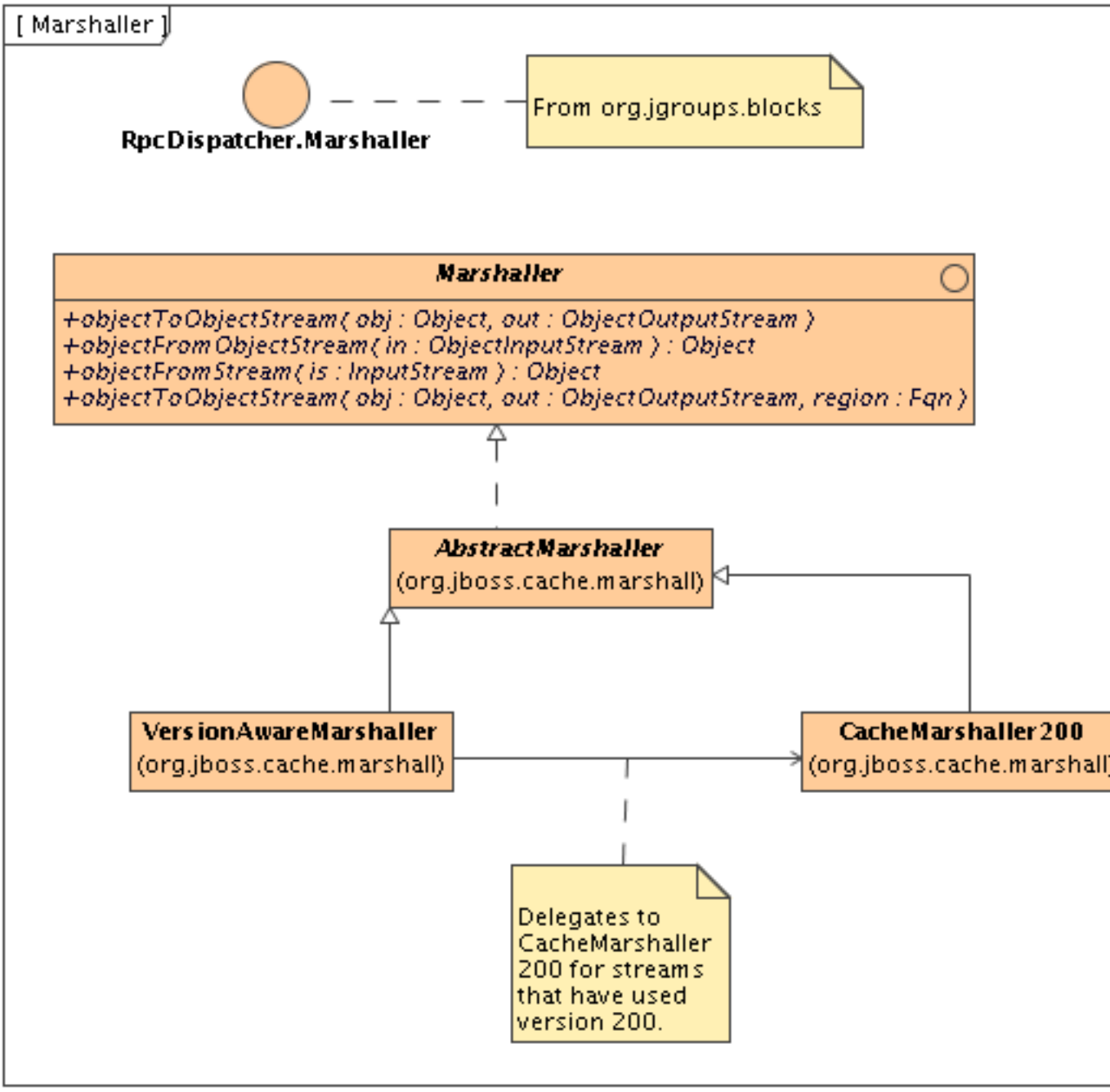


Figure 6.4. The Marshaller interface

6.5.1. The Marshaller Interface

The `Marshaller` interface extends `RpcDispatcher.Marshaller` from JGroups. This interface has two main implementations - a delegating `VersionAwareMarshaller` and a concrete `CacheMarshaller200`.

The marshaller can be obtained by calling `CacheSPI.getMarshaller()`, and defaults to the `VersionAwareMarshaller`. Users may also write their own marshallers by implementing the `Marshaller` interface and adding it to their configuration, by using the `MarshallerClass` configuration attribute.

6.5.2. VersionAwareMarshaller

As the name suggests, this marshaller adds a version `short` to the start of any stream when writing, enabling similar `VersionAwareMarshaller` instances to read the version short and know which specific marshaller implementation to delegate the call to. For example, `CacheMarshaller200` , is the marshaller for JBoss Cache 2.0.x. JBoss Cache 2.1.x, say, may ship with `CacheMarshaller210` with an improved wire protocol. Using a `VersionAwareMarshaller` helps achieve wire protocol compatibility between minor releases but still affords us the flexibility to tweak and improve the wire protocol between minor or micro releases.

6.5.2.1. CacheLoaders

Some of the existing cache loaders, such as the `JDBCCacheLoader` and the `FileCacheLoader` relied on persisting data using `ObjectOutputStream` as well, but now, they are using the `VersionAwareMarshaller` to marshall the persisted data to their cache stores.

6.5.3. CacheMarshaller200

This marshaller treats well-known objects that need marshalling - such as `MethodCall` , `Fqn` , `DataVersion` , and even some JDK objects such as `String` , `List` , `Boolean` and others as types that do not need complete class definitions. Instead, each of these well-known types are represented by a `short` , which is a lot more efficient.

In addition, reference counting is done to reduce duplication of writing certain objects multiple times, to help keep the streams small and efficient.

Also, if `UseRegionBasedMarshalling` is enabled (disabled by default) the marshaller adds region information to the stream before writing any data. This region information is in the form of a `String` representation of an `Fqn` . When unmarshalling, the `RegionManager` can be used to find the relevant `Region` , and use a region-specific `ClassLoader` to unmarshall the stream. This is specifically useful when used to cluster state for application servers, where each deployment has it's own `ClassLoader` . See the section below on regions for more information.

6.6. Class Loading and Regions

When used to cluster state of application servers, applications deployed in the application tend to put instances of objects specific to their application in the cache (or in an `HttpSession` object) which would require replication. It is common for application servers to assign separate `ClassLoader` instances to each application deployed, but have JBoss Cache libraries referenced by the application server's `ClassLoader` .

To enable us to successfully marshall and unmarshall objects from such class loaders, we use a concept called regions. A region is a portion of the cache which share a common class loader (a region also has other uses - see eviction policies).

A region is created by using the `Cache.getRegion(Fqn fqn, boolean createIfNotExists)` method, and returns an implementation of the `Region` interface. Once a region is obtained, a class loader for the region can be set or unset, and the region can be activated/deactivated. By default, regions are active unless the `InactiveOnStartup` configuration attribute is set to `true` .

2

Clustering

This chapter talks about aspects around clustering JBoss Cache.

7.1. Cache Replication Modes

JBoss Cache can be configured to be either local (standalone) or clustered. If in a cluster, the cache can be configured to replicate changes, or to invalidate changes. A detailed discussion on this follows.

7.1.1. Local Mode

Local caches don't join a cluster and don't communicate with other caches in a cluster. Therefore their elements don't need to be serializable - however, we recommend making them serializable, enabling a user to change the cache mode at any time. The dependency on the JGroups library is still there, although a JGroups channel is not started.

7.1.2. Replicated Caches

Replicated caches replicate all changes to some or all of the other cache instances in the cluster. Replication can either happen after each modification (no transactions), or at the end of a transaction (commit time).

Replication can be synchronous or asynchronous . Use of either one of the options is application dependent. Synchronous replication blocks the caller (e.g. on a `put()`) until the modifications have been replicated successfully to all nodes in a cluster. Asynchronous replication performs replication in the background (the `put()` returns immediately). JBoss Cache also offers a replication queue, where modifications are replicated periodically (i.e. interval-based), or when the queue size exceeds a number of elements, or a combination thereof.

Asynchronous replication is faster (no caller blocking), because synchronous replication requires acknowledgments from all nodes in a cluster that they received and applied the modification successfully (round-trip time). However, when a synchronous replication returns successfully, the caller knows for sure that all modifications have been applied to all cache instances, whereas this is not be the case with asynchronous replication. With asynchronous replication, errors are simply written to a log. Even when using transactions, a transaction may succeed but replication may not succeed on all cache instances.

7.1.2.1. Replicated Caches and Transactions

When using transactions, replication only occurs at the transaction boundary - i.e., when a transaction commits. This results in minimising replication traffic since a single modification is broadcast rather than a series of individual modifications, and can be a lot more efficient than not using transactions. Another effect of this is that if a transaction were to roll back, nothing is broadcast across a cluster.

Depending on whether you are running your cluster in asynchronous or synchronous mode, JBoss Cache will use either a single phase or two phase commit [http://en.wikipedia.org/wiki/Two-phase_commit_protocol] protocol, respectively.

7.1.2.1.1. One Phase Commits

Used when your cache mode is REPL_ASYNC. All modifications are replicated in a single call, which instructs remote caches to apply the changes to their local in-memory state and commit locally. Remote errors/rollbacks are never fed back to the originator of the transaction since the communication is asynchronous.

7.1.2.1.2. Two Phase Commits

Used when your cache mode is REPL_SYNC. Upon committing your transaction, JBoss Cache broadcasts a prepare call, which carries all modifications relevant to the transaction. Remote caches then acquire local locks on their in-memory state and apply the modifications. Once all remote caches respond to the prepare call, the originator of the transaction broadcasts a commit. This instructs all remote caches to commit their data. If any of the caches fail to respond to the prepare phase, the originator broadcasts a rollback.

Note that although the prepare phase is synchronous, the commit and rollback phases are asynchronous. This is because Sun's JTA specification [http://java.sun.com/products/jta/] does not specify how transactional resources should deal with failures at this stage of a transaction; and other resources participating in the transaction may have indeterminate state anyway. As such, we do away with the overhead of synchronous communication for this phase of the transaction. That said, they can be forced to be synchronous using the `SyncCommitPhase` and `SyncRollbackPhase` configuration attributes.

7.1.2.2. Buddy Replication

Buddy Replication allows you to suppress replicating your data to all instances in a cluster. Instead, each instance picks one or more 'buddies' in the cluster, and only replicates to these specific buddies. This greatly helps scalability as there is no longer a memory and network traffic impact every time another instance is added to a cluster.

One of the most common use cases of Buddy Replication is when a replicated cache is used by a servlet container to store HTTP session data. One of the pre-requisites to buddy replication working well and being a real benefit is the use of *session affinity* , more casually known as *sticky sessions* in HTTP session replication speak. What this means is that if certain data is frequently accessed, it is desirable that this is always accessed on one instance rather than in a round-robin fashion as this helps the cache cluster optimise how it chooses buddies, where it stores data, and minimises replication traffic.

If this is not possible, Buddy Replication may prove to be more of an overhead than a benefit.

7.1.2.2.1. Selecting Buddies

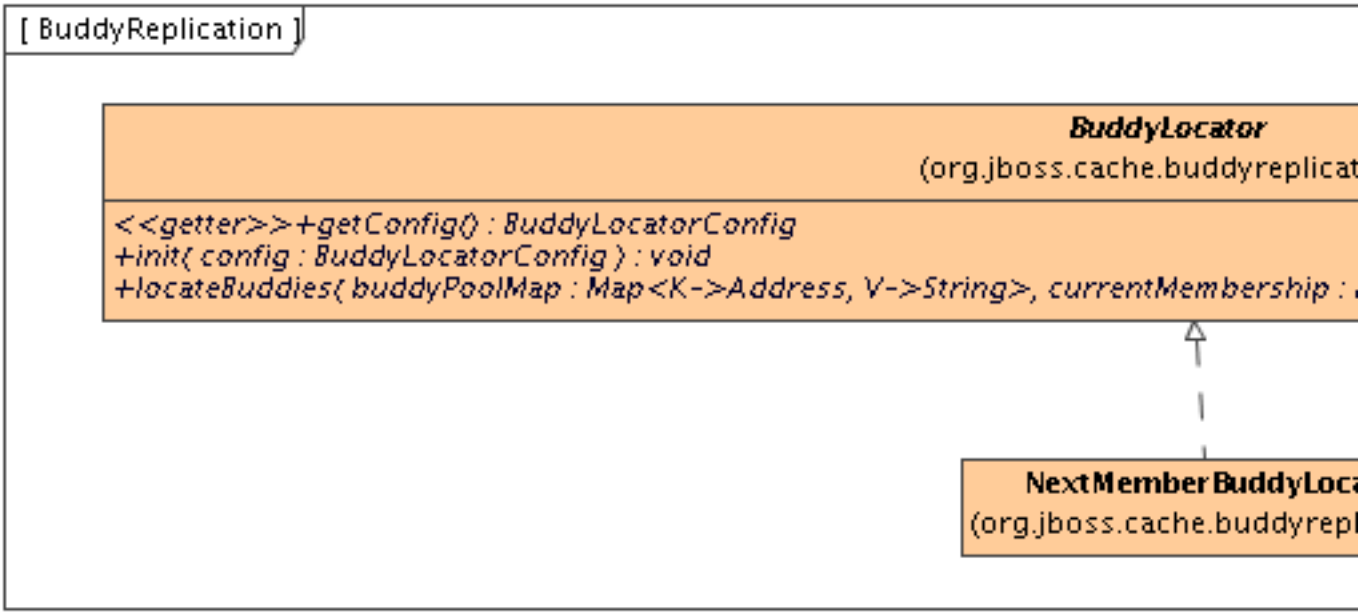


Figure 7.1. BuddyLocator

Buddy Replication uses an instance of a `BuddyLocator` which contains the logic used to select buddies in a network. JBoss Cache currently ships with a single implementation, `NextMemberBuddyLocator` , which is used as a default if no implementation is provided. The `NextMemberBuddyLocator` selects the next member in the cluster, as the name suggests, and guarantees an even spread of buddies for each instance.

The `NextMemberBuddyLocator` takes in 2 parameters, both optional.

- `numBuddies` - specifies how many buddies each instance should pick to back its data onto. This defaults to 1.
- `ignoreColocatedBuddies` - means that each instance will *try* to select a buddy on a different physical host. If not able to do so though, it will fall back to colocated instances. This defaults to `true` .

7.1.2.2.2. BuddyPools

Also known as *replication groups* , a buddy pool is an optional construct where each instance in a cluster may be configured with a buddy pool name. Think of this as an 'exclusive club membership' where when selecting buddies, `BuddyLocator` s that support buddy pools would try and select buddies sharing the same buddy pool name. This allows system administrators a degree of flexibility and control over how buddies are selected. For example, a sysadmin may put two instances on two separate physical servers that may be on two separate physical racks in the same buddy pool. So rather than picking an instance on a different host on the same rack, `BuddyLocator` s would rather pick the instance in the same buddy pool, on a separate rack which may add a degree of redundancy.

7.1.2.2.3. Failover

In the unfortunate event of an instance crashing, it is assumed that the client connecting to the cache (directly or indirectly, via some other service such as HTTP session replication) is able to redirect the

request to any other random cache instance in the cluster. This is where a concept of Data Gravitation comes in.

Data Gravitation is a concept where if a request is made on a cache in the cluster and the cache does not contain this information, it asks other instances in the cluster for the data. In other words, data is lazily transferred, migrating *only* when other nodes ask for it. This strategy prevents a network storm effect where lots of data is pushed around healthy nodes because only one (or a few) of them die.

If the data is not found in the primary section of some node, it would (optionally) ask other instances to check in the backup data they store for other caches. This means that even if a cache containing your session dies, other instances will still be able to access this data by asking the cluster to search through their backups for this data.

Once located, this data is transferred to the instance which requested it and is added to this instance's data tree. The data is then (optionally) removed from all other instances (and backups) so that if session affinity is used, the affinity should now be to this new cache instance which has just *taken ownership* of this data.

Data Gravitation is implemented as an interceptor. The following (all optional) configuration properties pertain to data gravitation.

- `dataGravitationRemoveOnFind` - forces all remote caches that own the data or hold backups for the data to remove that data, thereby making the requesting cache the new data owner. This removal, of course, only happens after the new owner finishes replicating data to its buddy. If set to `false` an evict is broadcast instead of a remove, so any state persisted in cache loaders will remain. This is useful if you have a shared cache loader configured. Defaults to `true` .

- `dataGravitationSearchBackupTrees` - Asks remote instances to search through their backups as well as main data trees. Defaults to `true` . The resulting effect is that if this is `true` then backup nodes can respond to data gravitation requests in addition to data owners.

- `autoDataGravitation` - Whether data gravitation occurs for every cache miss. By default this is set to `false` to prevent unnecessary network calls. Most use cases will know when it may need to gravitate data and will pass in an `Option` to enable data gravitation on a per-invocation basis. If `autoDataGravitation` is `true` this `Option` is unnecessary.

7.1.2.2.4. Configuration

```
<!-- Buddy Replication config -->
<attribute name="BuddyReplicationConfig">
   <config>

      <!-- Enables buddy replication. This is the ONLY mandatory configuration element here. -->
      <buddyReplicationEnabled>true</buddyReplicationEnabled>

      <!-- These are the default values anyway -->
      <buddyLocatorClass>org.jboss.cache.buddyreplication.NextMemberBuddyLocator</buddyLocatorCl

      <!--  numBuddies is the number of backup nodes each node maintains. ignoreColocatedBuddies
```

```
            that each node will *try* to select a buddy on a different physical host. If not abl
            it will fall back to colocated nodes. -->
      <buddyLocatorProperties>
         numBuddies = 1
         ignoreColocatedBuddies = true
      </buddyLocatorProperties>

      <!-- A way to specify a preferred replication group. If specified, we try and pick a buddy
           the same pool name (falling back to other buddies if not available). This allows the
           hint at backup buddies are picked, so for example, nodes may be hinted topick buddies
           physical rack or power supply for added fault tolerance. -->
      <buddyPoolName>myBuddyPoolReplicationGroup</buddyPoolName>

      <!-- Communication timeout for inter-buddy group organisation messages (such as assigning
           removing from groups, defaults to 1000. -->
      <buddyCommunicationTimeout>2000</buddyCommunicationTimeout>

      <!-- Whether data is removed from old owners when gravitated to a new owner. Defaults to t
      <dataGravitationRemoveOnFind>true</dataGravitationRemoveOnFind>

      <!-- Whether backup nodes can respond to data gravitation requests, or only the data owner
           supposed to respond.  Defaults to true. -->
      <dataGravitationSearchBackupTrees>true</dataGravitationSearchBackupTrees>

      <!-- Whether all cache misses result in a data gravitation request. Defaults to false, req
           callers to enable data gravitation on a per-invocation basis using the Options API. -
      <autoDataGravitation>false</autoDataGravitation>

   </config>
</attribute>
```

7.2. Invalidation

If a cache is configured for invalidation rather than replication, every time data is changed in a cache other caches in the cluster receive a message informing them that their data is now stale and should be evicted from memory. Invalidation, when used with a shared cache loader (see chapter on Cache Loaders) would cause remote caches to refer to the shared cache loader to retrieve modified data. The benefit of this is twofold: network traffic is minimised as invalidation messages are very small compared to replicating updated data, and also that other caches in the cluster look up modified data in a lazy manner, only when needed.

Invalidation messages are sent after each modification (no transactions), or at the end of a transaction, upon successful commit. This is usually more efficient as invalidation messages can be optimised for the transaction as a whole rather than on a per-modification basis.

Invalidation too can be synchronous or asynchronous, and just as in the case of replication, synchronous invalidation blocks until all caches in the cluster receive invalidation messages and have evicted stale data while asynchronous invalidation works in a 'fire-and-forget' mode, where invalidation messages are broadcast but doesn't block and wait for responses.

7.3. State Transfer

State Transfer refers to the process by which a JBoss Cache instance prepares itself to begin providing a service by acquiring the current state from another cache instance and integrating that state into its own state.

7.3.1. State Transfer Types

There are three divisions of state transfer types depending on a point of view related to state transfer. First, in the context of particular state transfer implementation, the underlying plumbing, there are two starkly different state transfer types: byte array and streaming based state transfer. Second, state transfer can be full or partial state transfer depending on a subtree being transferred. Entire cache tree transfer represents full transfer while transfer of a particular subtree represents partial state transfer. And finally state transfer can be "in-memory" and "persistent" transfer depending on a particular use of cache.

7.3.2. Byte array and streaming based state transfer

Byte array based transfer was a default and only transfer methodology for cache in all previous releases up to 2.0. Byte array based transfer loads entire state transferred into a byte array and sends it to a state receiving member. Major limitation of this approach is that the state transfer that is very large (>1GB) would likely result in OutOfMemoryException. Streaming state transfer provides an InputStream to a state reader and an OutputStream to a state writer. OutputStream and InputStream abstractions enable state transfer in byte chunks thus resulting in smaller memory requirements. For example, if application state is represented as a tree whose aggregate size is 1GB, rather than having to provide a 1GB byte array streaming state transfer transfers the state in chunks of N bytes where N is user configurable.

Byte array and streaming based state transfer are completely API transparent, interchangeable, and statically configured through a standard cache configuration XML file. Refer to JGroups documentation on how to change from one type of transfer to another.

7.3.3. Full and partial state transfer

If either in-memory or persistent state transfer is enabled, a full or partial state transfer will be done at various times, depending on how the cache is used. "Full" state transfer refers to the transfer of the state related to the entire tree -- i.e. the root node and all nodes below it. A "partial" state transfer is one where just a portion of the tree is transferred -- i.e. a node at a given Fqn and all nodes below it.

If either in-memory or persistent state transfer is enabled, state transfer will occur at the following times:

1. Initial state transfer. This occurs when the cache is first started (as part of the processing of the `start()` method). This is a full state transfer. The state is retrieved from the cache instance that has been operational the longest. [1] If there is any problem receiving or integrating the state, the cache will not start.

   Initial state transfer will occur unless:

   a. The cache's `InactiveOnStartup` property is `true` . This property is used in conjunction with region-based marshalling.

b. Buddy replication is used. See below for more on state transfer with buddy replication.

2. Partial state transfer following region activation. When region-based marshalling is used, the application needs to register a specific class loader with the cache. This class loader is used to unmarshall the state for a specific region (subtree) of the cache.

   After registration, the application calls `cache.getRegion(fqn, true).activate()`, which initiates a partial state transfer of the relevant subtree's state. The request is first made to the oldest cache instance in the cluster. However, if that instance responds with no state, it is then requested from each instance in turn until one either provides state or all instances have been queried.

   Typically when region-based marshalling is used, the cache's `InactiveOnStartup` property is set to `true`. This suppresses initial state transfer, which would fail due to the inability to deserialize the transferred state.

3. Buddy replication. When buddy replication is used, initial state transfer is disabled. Instead, when a cache instance joins the cluster, it becomes the buddy of one or more other instances, and one or more other instances become its buddy. Each time an instance determines it has a new buddy providing backup for it, it pushes it's current state to the new buddy. This "pushing" of state to the new buddy is slightly different from other forms of state transfer, which are based on a "pull" approach (i.e. recipient asks for and receives state). However, the process of preparing and integrating the state is the same.

   This "push" of state upon buddy group formation only occurs if the `InactiveOnStartup` property is set to `false`. If it is `true`, state transfer amongst the buddies only occurs when the application activates the region on the various members of the group.

   Partial state transfer following a region activation call is slightly different in the buddy replication case as well. Instead of requesting the partial state from one cache instance, and trying all instances until one responds, with buddy replication the instance that is activating a region will request partial state from each instance for which it is serving as a backup.

7.3.4. Transient ("in-memory") and persistent state transfer

The state that is acquired and integrated can consist of two basic types:

1. "Transient" or "in-memory" state. This consists of the actual in-memory state of another cache instance - the contents of the various in-memory nodes in the cache that is providing state are serialized and transferred; the recipient deserializes the data, creates corresponding nodes in its own in-memory tree, and populates them with the transferred data.

   "In-memory" state transfer is enabled by setting the cache's `FetchInMemoryState` configuration attribute to `true`.

2. "Persistent" state. Only applicable if a non-shared cache loader is used. The state stored in the state-provider cache's persistent store is deserialized and transferred; the recipient passes the data to its own cache loader, which persists it to the recipient's persistent store.

"Persistent" state transfer is enabled by setting a cache loader's `fetchPersistentState` attribute to `true` . If multiple cache loaders are configured in a chain, only one can have this property set to true; otherwise you will get an exception at startup.

Persistent state transfer with a shared cache loader does not make sense, as the same persistent store that provides the data will just end up receiving it. Therefore, if a shared cache loader is used, the cache will not allow a persistent state transfer even if a cache loader has `fetchPersistentState` set to `true` .

Which of these types of state transfer is appropriate depends on the usage of the cache.

1. If a write-through cache loader is used, the current cache state is fully represented by the persistent state. Data may have been evicted from the in-memory state, but it will still be in the persistent store. In this case, if the cache loader is not shared, persistent state transfer is used to ensure the new cache has the correct state. In-memory state can be transferred as well if the desire is to have a "hot" cache -- one that has all relevant data in memory when the cache begins providing service. (Note that the `<cacheloader><preload>` element in the `CacheLoaderConfig` configuration parameter can be used as well to provide a "warm" or "hot" cache without requiring an in-memory state transfer. This approach somewhat reduces the burden on the cache instance providing state, but increases the load on the persistent store on the recipient side.)

2. If a cache loader is used with passivation, the full representation of the state can only be obtained by combining the in-memory (i.e. non-passivated) and persistent (i.e. passivated) states. Therefore an in-memory state transfer is necessary. A persistent state transfer is necessary if the cache loader is not shared.

3. If no cache loader is used and the cache is solely a write-aside cache (i.e. one that is used to cache data that can also be found in a persistent store, e.g. a database), whether or not in-memory state should be transferred depends on whether or not a "hot" cache is desired.

7.3.5. Configuring State Transfer

To ensure state transfer behaves as expected, it is important that all nodes in the cluster are configured with the same settings for persistent and transient state. This is because byte array based transfers, when requested, rely only on the requester's configuration while stream based transfers rely on both the requester and sender's configuration, and this is expected to be identical.

3

Cache Loaders

JBoss Cache can use a `CacheLoader` to back up the in-memory cache to a backend datastore. If JBoss Cache is configured with a cache loader, then the following features are provided:

- Whenever a cache element is accessed, and that element is not in the cache (e.g. due to eviction or due to server restart), then the cache loader transparently loads the element into the cache if found in the backend store.

- Whenever an element is modified, added or removed, then that modification is persisted in the backend store via the cache loader. If transactions are used, all modifications created within a transaction are persisted. To this end, the `CacheLoader` takes part in the two phase commit protocol run by the transaction manager, although it does not do so explicitly.

8.1. The CacheLoader Interface and Lifecycle

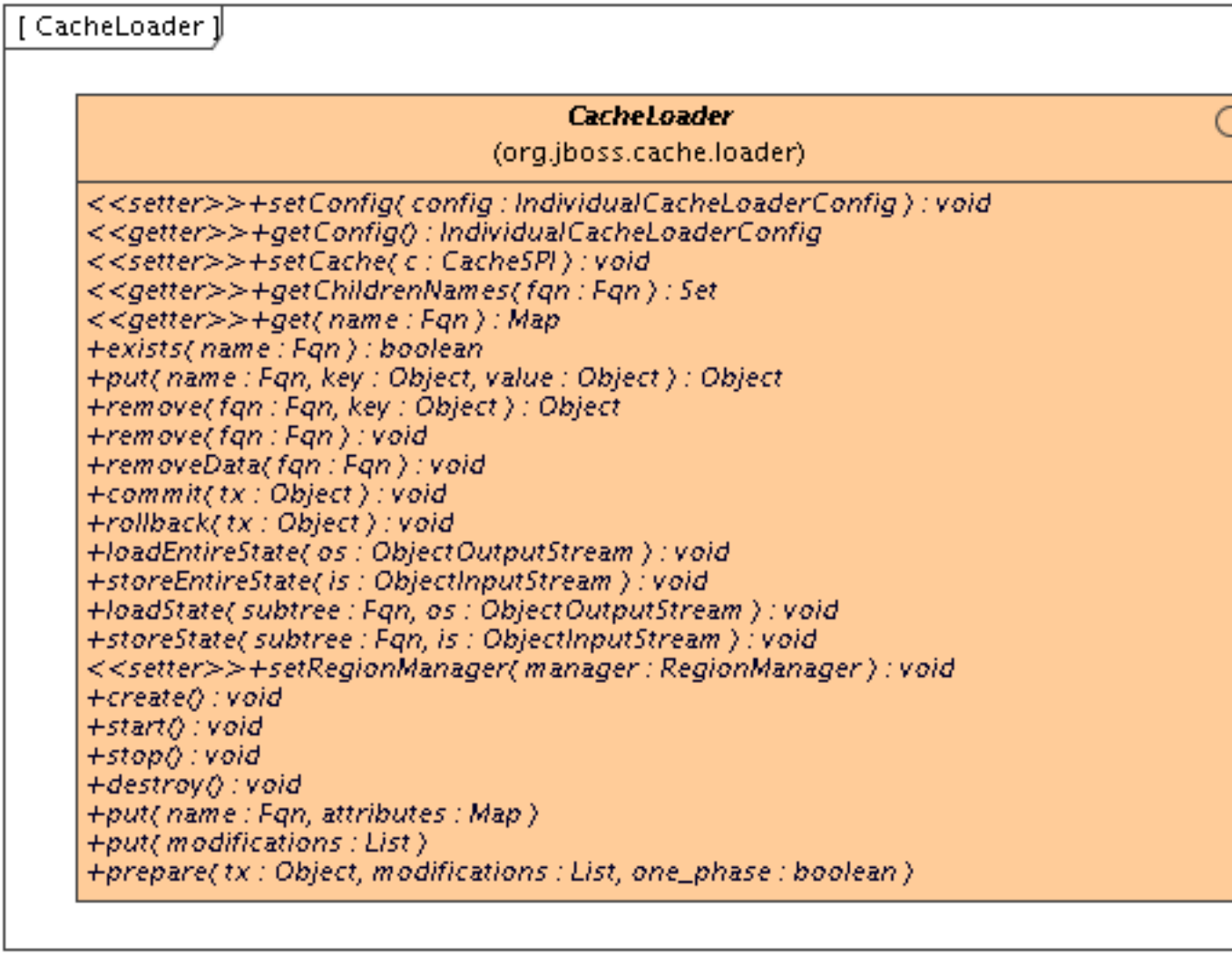


Figure 8.1. The CacheLoader interface

The interaction between JBoss Cache and a `CacheLoader` implementation is as follows. When `CacheLoaderConfiguration` (see below) is non-null, an instance of each configured `CacheLoader` is created when the cache is created, and started when the cache is started.

`CacheLoader.create()` and `CacheLoader.start()` are called when the cache is started. Correspondingly, `stop()` and `destroy()` are called when the cache is stopped.

Next, `setConfig()` and `setCache()` are called. The latter can be used to store a reference to the cache, the former is used to configure this instance of the `CacheLoader` . For example, here a database cache loader could establish a connection to the database.

The `CacheLoader` interface has a set of methods that are called when no transactions are used: `get()` , `put()` , `remove()` and `removeData()` : they get/set/remove the value immediately. These methods are described as javadoc comments in the interface.

Then there are three methods that are used with transactions: `prepare()`, `commit()` and `rollback()`. The `prepare()` method is called when a transaction is to be committed. It has a transaction object and a list of modfications as argument. The transaction object can be used as a key into a hashmap of transactions, where the values are the lists of modifications. Each modification list has a number of `Modification` elements, which represent the changes made to a cache for a given transaction. When `prepare()` returns successfully, then the cache loader *must* be able to commit (or rollback) the transaction successfully.

JBoss Cache takes care of calling prepare(), commit() and rollback() on the cache loaders at the right time.

The `commit()` method tells the cache loader to commit the transaction, and the `rollback()` method tells the cache loader to discard the changes associated with that transaction.

See the javadocs on this interface for a detailed explanation on each method and the contract implementations would need to fulfil.

8.2. Configuration

Cache loaders are configured as follows in the JBoss Cache XML file. Note that you can define several cache loaders, in a chain. The impact is that the cache will look at all of the cache loaders in the order they've been configured, until it finds a valid, non-null element of data. When performing writes, all cache loaders are written to (except if the `ignoreModifications` element has been set to `true` for a specific cache loader. See the configuration section below for details.

```
...

<!-- Cache loader config block -->
<attribute name="CacheLoaderConfiguration">
   <config>
      <!-- if passivation is true, only the first cache loader is used; the rest are ignored -->
      <passivation>false</passivation>
      <!-- comma delimited FQNs to preload -->
      <preload>/</preload>
      <!-- are the cache loaders shared in a cluster? -->
      <shared>false</shared>

      <!-- we can now have multiple cache loaders, which get chained -->
      <!-- the 'cacheloader' element may be repeated -->
      <cacheloader>

         <class>org.jboss.cache.loader.JDBCCacheLoader</class>

         <!-- properties to pass in to the cache loader -->
         <properties>
            cache.jdbc.driver=com.mysql.jdbc.Driver
            cache.jdbc.url=jdbc:mysql://localhost:3306/jbossdb
            cache.jdbc.user=root
            cache.jdbc.password=
            cache.jdbc.sql-concat=concat(1,2)
         </properties>
```

```
         <!-- whether the cache loader writes are asynchronous -->
         <async>false</async>

         <!-- only one cache loader in the chain may set fetchPersistentState to true.
              An exception is thrown if more than one cache loader sets this to true. -->
         <fetchPersistentState>true</fetchPersistentState>

         <!-- determines whether this cache loader ignores writes - defaults to false. -->
         <ignoreModifications>false</ignoreModifications>

         <!-- if set to true, purges the contents of this cache loader when the cache starts up.
              Defaults to false. -->
         <purgeOnStartup>false</purgeOnStartup>

         <!-- defines the cache loader as a singleton store where only the coordinator of the
              cluster will store modifications. -->
         <singletonStore>
            <!-- if true, singleton store functionality is enabled, defaults to false -->
            <enabled>false</enabled>

            <!-- implementation class for singleton store functionality which must extend
                 org.jboss.cache.loader.AbstractDelegatingCacheLoader. Default implementation
                 is org.jboss.cache.loader.SingletonStoreCacheLoader -->
            <class>org.jboss.cache.loader.SingletonStoreCacheLoader</class>

            <!-- properties and default values for the default singleton store functionality
                 implementation -->
            <properties>
               pushStateWhenCoordinator=true
               pushStateWhenCoordinatorTimeout=20000
            </properties>
         </singletonStore>
      </cacheloader>

   </config>
</attribute>
```

The `class` element defines the class of the cache loader implementation. (Note that, because of a bug in the properties editor in JBoss AS, backslashes in variables for Windows filenames might not get expanded correctly, so replace="false" may be necessary). Note that an implementation of cache loader has to have an empty constructor.

The `properties` element defines a configuration specific to the given implementation. The filesystem-based implementation for example defines the root directory to be used, whereas a database implementation might define the database URL, name and password to establish a database connection. This configuration is passed to the cache loader implementation via `CacheLoader.setConfig(Properties)`. Note that backspaces may have to be escaped.

`preload` allows us to define a list of nodes, or even entire subtrees, that are visited by the cache on startup, in order to preload the data associated with those nodes. The default ("/") loads the entire data available in the backend store into the cache, which is probably not a good idea given that the data in the backend store might be large. As an example, `/a, /product/catalogue` loads the subtrees `/a` and `/product/catalogue`

into the cache, but nothing else. Anything else is loaded lazily when accessed. Preloading makes sense when one anticipates using elements under a given subtree frequently. .

`fetchPersistentState` determines whether or not to fetch the persistent state of a cache when joining a cluster. Only one configured cache loader may set this property to true; if more than one cache loader does so, a configuration exception will be thrown when starting your cache service.

`async` determines whether writes to the cache loader block until completed, or are run on a separate thread so writes return immediately. If this is set to true, an instance of `org.jboss.cache.loader.AsyncCacheLoader` is constructed with an instance of the actual cache loader to be used. The `AsyncCacheLoader` then delegates all requests to the underlying cache loader, using a separate thread if necessary. See the Javadocs on `AsyncCacheLoader` for more details. If unspecified, the `async` element defaults to `false` .

Note on using the `async` element: there is always the possibility of dirty reads since all writes are performed asynchronously, and it is thus impossible to guarantee when (and even if) a write succeeds. This needs to be kept in mind when setting the `async` element to true.

`ignoreModifications` determines whether write methods are pushed down to the specific cache loader. Situations may arise where transient application data should only reside in a file based cache loader on the same server as the in-memory cache, for example, with a further shared `JDBCCacheLoader` used by all servers in the network. This feature allows you to write to the 'local' file cache loader but not the shared `JDBCCacheLoader` . This property defaults to `false` , so writes are propagated to all cache loaders configured.

`purgeOnStatup` empties the specified cache loader (if `ignoreModifications` is `false`) when the cache loader starts up.

`shared` indicates that the cache loader is shared among different cache instances, for example where all instances in a cluster use the same JDBC settings t talk to the same remote, shared database. Setting this to `true` prevents repeated and unnecessary writes of the same data to the cache loader by different cache instances. Default value is `false` .

8.2.1. Singleton Store Configuration

`singletonStore` element enables modifications to be stored by only one node in the cluster, the coordinator. Essentially, whenever any data comes in to some node it is always replicated so as to keep the caches' in-memory states in sync; the coordinator, though, has the sole responsibility of pushing that state to disk. This functionality can be activated setting the `enabled` subelement to true in all nodes, but again only the coordinator of the cluster will store the modifications in the underlying cache loader as defined in `cacheloader` element. You cannot define a cache loader as `shared` and with `singletonStore` enabled at the same time. Default value for `enabled` is `false` .

Optionally, within the `singletonStore` element, you can define a `class` element that specifies the implementation class that provides the singleton store functionality. This class must extend `org.jboss.cache.loader.AbstractDelegatingCacheLoader` , and if absent, it defaults to `org.jboss.cache.loader.SingletonStoreCacheLoader` .

The `properties` subelement defines properties that allow changing the behaivour of the class providing the singleton store functionality. By default, `pushStateWhenCoordinator` and `pushStateWhenCoordinatorTimeout` properties have been defined, but more could be added as required by the user-defined class providing singleton store functionality.

`pushStateWhenCoordinator` allows the in-memory state to be pushed to the cache store when a node becomes the coordinator, as a result of the new election of coordinator due to a cluster topology change. This can be very useful in situations where the coordinator crashes and there's a gap in time until the new coordinator is elected. During this time, if this property was set to `false` and the cache was updated, these changes would never be persisted. Setting this property to `true` would ensure that any changes during this process also get stored in the cache loader. You would also want to set this property to `true` if each node's cache loader is configured with a different location. Default value is `true` .

`pushStateWhenCoordinatorTimeout` is only relevant if `pushStateWhenCoordinator` is `true` in which case, sets the maximum number of milliseconds that the process of pushing the in-memory state to the underlying cache loader should take, reporting a `PushStateException` if exceeded. Default value is 20000.

Note on using the `singletonStore` element: setting up a cache loader as a singleton and using cache passivation (via evictions) can lead to undesired effects. If a node is to be passivated as a result of an eviction, while the cluster is in the process of electing a new coordinator, the data will be lost. This is because no coordinator is active at that time and therefore, none of the nodes in the cluster will store the passivated node. A new coordinator is elected in the cluster when either, the coordinator leaves the cluster, the coordinator crashes or stops responding.

8.3. Shipped Implementations

The currently available implementations shipped with JBoss Cache are as follows.

8.3.1. File system based cache loaders

JBoss Cache ships with several cache loaders that utilise the file system as a data store. They all require that the `<cacheloader><properties>` configuration element contains a `location` property, which maps to a directory to be used as a persistent store. (e.g., `location=/tmp/myDataStore`). Used mainly for testing and not recommended for production use.

- `FileCacheLoader` , which is a simple filesystem-based implementation. By default, this cache loader checks for any potential character portability issues in the location or tree node names, for example invalid characters, producing warning messages. These checks can be disabled adding `check.character.portability` property to the `<properties>` element and setting it to `false` (e.g., `check.character.portability=false`).

  The FileCacheLoader has some severe limitations which restrict it's use in a production environment, or if used in such an environment, it should be used with due care and sufficient understanding of these limitations.

  - Due to the way the FileCacheLoader represents a tree structure on disk (directories and files) traversal is inefficient for deep trees.

- Usage on shared filesystems like NFS, Windows shares, etc. should be avoided as these do not implement proper file locking and can cause data corruption.
- Usage with an isolation level of NONE can cause corrupt writes as multiple threads attempt to write to the same file.
- File systems are inherently not transactional, so when attempting to use your cache in a transactional context, failures when writing to the file (which happens during the commit phase) cannot be recovered.

As a rule of thumb, it is recommended that the FileCacheLoader not be used in a highly concurrent, transactional or stressful environment, and it's use is restricted to testing.

- `BdbjeCacheLoader` , which is a cache loader implementation based on the Oracle/Sleepycat's BerkeleyDB Java Edition [http://www.oracle.com/database/berkeley-db/index.html] .
- `JdbmCacheLoader` , which is a cache loader implementation based on the JDBM engine [http://jdbm.sourceforge.net/] , a fast and free alternative to BerkeleyDB.

Note that the BerkeleyDB implementation is much more efficient than the filesystem-based implementation, and provides transactional guarantees, but requires a commercial license if distributed with an application (see http://www.oracle.com/database/berkeley-db/index.html for details).

8.3.2. Cache loaders that delegate to other caches

- `LocalDelegatingCacheLoader` , which enables loading from and storing to another local (same JVM) cache.
- `ClusteredCacheLoader` , which allows querying of other caches in the same cluster for in-memory data via the same clustering protocols used to replicate data. Writes are *not* 'stored' though, as replication would take care of any updates needed. You need to specify a property called `timeout` , a long value telling the cache loader how many milliseconds to wait for responses from the cluster before assuming a null value. For example, `timeout = 3000` would use a timeout value of 3 seconds.

8.3.3. JDBCCacheLoader

JBossCache is distributed with a JDBC-based cache loader implementation that stores/loads nodes' state into a relational database. The implementing class is `org.jboss.cache.loader.JDBCCacheLoader` .

The current implementation uses just one table. Each row in the table represents one node and contains three columns:

- column for `Fqn` (which is also a primary key column)
- column for node contents (attribute/value pairs)
- column for parent `Fqn`

`Fqn` 's are stored as strings. Node content is stored as a BLOB. *WARNING:* JBoss Cache does not impose any limitations on the types of objects used in `Fqn` but this implementation of cache loader requires `Fqn`

to contain only objects of type `java.lang.String`. Another limitation for `Fqn` is its length. Since `Fqn` is a primary key, its default column type is `VARCHAR` which can store text values up to some maximum length determined by the database in use.

See http://wiki.jboss.org/wiki/Wiki.jsp?page=JDBCCacheLoader [http://wiki.jboss.org/wiki/Wiki.jsp?page=JDBCCacheLoader] for configuration tips with specific database systems.

8.3.3.1. JDBCCacheLoader configuration

8.3.3.1.1. Table configuration

Table and column names as well as column types are configurable with the following properties.

- *cache.jdbc.table.name* - the name of the table. Can be prepended with schema name for the given table: <schema_name>.<table_name>. The default value is 'jbosscache'.
- *cache.jdbc.table.primarykey* - the name of the primary key for the table. The default value is 'jbosscache_pk'.
- *cache.jdbc.table.create* - can be true or false. Indicates whether to create the table during startup. If true, the table is created if it doesn't already exist. The default value is true.
- *cache.jdbc.table.drop* - can be true or false. Indicates whether to drop the table during shutdown. The default value is true.
- *cache.jdbc.fqn.column* - FQN column name. The default value is 'fqn'.
- *cache.jdbc.fqn.type* - FQN column type. The default value is 'varchar(255)'.
- *cache.jdbc.node.column* - node contents column name. The default value is 'node'.
- *cache.jdbc.node.type* - node contents column type. The default value is 'blob'. This type must specify a valid binary data type for the database being used.

8.3.3.1.2. DataSource

If you are using JBossCache in a managed environment (e.g., an application server) you can specify the JNDI name of the DataSource you want to use.

- *cache.jdbc.datasource* - JNDI name of the DataSource. The default value is `java:/DefaultDS`.

8.3.3.1.3. JDBC driver

If you are *not* using DataSource you have the following properties to configure database access using a JDBC driver.

- *cache.jdbc.driver* - fully qualified JDBC driver name.
- *cache.jdbc.url* - URL to connect to the database.

- *cache.jdbc.user* - user name to connect to the database.
- *cache.jdbc.password* - password to connect to the database.

8.3.3.1.4. c3p0 connection pooling

JBoss Cache implements JDBC connection pooling when running outside of an application server standalone using the c3p0:JDBC DataSources/Resource Pools library. In order to enable it, just edit the following property:

- *cache.jdbc.connection.factory* - Connection factory class name. If not set, it defaults to standard non-pooled implementation. To enable c3p0 pooling, just set the connection factory class for c3p0. See example below.

You can also set any c3p0 parameters in the same cache loader properties section but don't forget to start the property name with 'c3p0.'. To find a list of available properties, please check the c3p0 documentation for the c3p0 library version distributed in c3p0:JDBC DataSources/Resource Pools [http://sourceforge.net/projects/c3p0] . Also, in order to provide quick and easy way to try out different pooling parameters, any of these properties can be set via a System property overriding any values these properties might have in the JBoss Cache XML configuration file, for example: `-Dc3p0.maxPoolSize=20` . If a c3p0 property is not defined in either the configuration file or as a System property, default value, as indicated in the c3p0 documentation, will apply.

8.3.3.1.5. Configuration example

Below is an example of a JDBCCacheLoader using Oracle as database. The CacheLoaderConfiguration XML element contains an arbitrary set of properties which define the database-related configuration.

```
<attribute name="CacheLoaderConfiguration">
<config>
   <passivation>false</passivation>
   <preload>/some/stuff</preload>
   <cacheloader>
      <class>org.jboss.cache.loader.JDBCCacheLoader</class>

      <properties>
         cache.jdbc.table.name=jbosscache
         cache.jdbc.table.create=true
         cache.jdbc.table.drop=true
         cache.jdbc.table.primarykey=jbosscache_pk
         cache.jdbc.fqn.column=fqn
         cache.jdbc.fqn.type=varchar(255)
         cache.jdbc.node.column=node
         cache.jdbc.node.type=blob
         cache.jdbc.parent.column=parent
         cache.jdbc.driver=oracle.jdbc.OracleDriver
         cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
         cache.jdbc.user=SCOTT
         cache.jdbc.password=TIGER
         cache.jdbc.sql-concat=concat(1,2)
```

```
      </properties>

      <async>false</async>
      <fetchPersistentState>true</fetchPersistentState>
      <ignoreModifications>false</ignoreModifications>
      <purgeOnStartup>false</purgeOnStartup>
   </cacheloader>
</config>
</attribute>
```

As an alternative to configuring the entire JDBC connection, the name of an existing data source can be given:

```
<attribute name="CacheLoaderConfiguration">
<config>
   <passivation>false</passivation>
   <preload>/some/stuff</preload>
   <cacheloader>
      <class>org.jboss.cache.loader.JDBCCacheLoader</class>

      <properties>
         cache.jdbc.datasource=java:/DefaultDS
      </properties>

      <async>false</async>
      <fetchPersistentState>true</fetchPersistentState>
      <ignoreModifications>false</ignoreModifications>
      <purgeOnStartup>false</purgeOnStartup>
   </cacheloader>
</config>
</attribute>
```

Cconfiguration example for a cache loader using c3p0 JDBC connection pooling:

```
<attribute name="CacheLoaderConfiguration">
<config>
   <passivation>false</passivation>
   <preload>/some/stuff</preload>
   <cacheloader>
      <class>org.jboss.cache.loader.JDBCCacheLoader</class>

      <properties>
         cache.jdbc.table.name=jbosscache
         cache.jdbc.table.create=true
         cache.jdbc.table.drop=true
         cache.jdbc.table.primarykey=jbosscache_pk
         cache.jdbc.fqn.column=fqn
         cache.jdbc.fqn.type=varchar(255)
```

```
         cache.jdbc.node.column=node
         cache.jdbc.node.type=blob
         cache.jdbc.parent.column=parent
         cache.jdbc.driver=oracle.jdbc.OracleDriver
         cache.jdbc.url=jdbc:oracle:thin:@localhost:1521:JBOSSDB
         cache.jdbc.user=SCOTT
         cache.jdbc.password=TIGER
         cache.jdbc.sql-concat=concat(1,2)
         cache.jdbc.connection.factory=org.jboss.cache.loader.C3p0ConnectionFactory
         c3p0.maxPoolSize=20
         c3p0.checkoutTimeout=5000
      </properties>

      <async>false</async>
      <fetchPersistentState>true</fetchPersistentState>
      <ignoreModifications>false</ignoreModifications>
      <purgeOnStartup>false</purgeOnStartup>
   </cacheloader>
</config>
</attribute>
```

8.3.4. TcpDelegatingCacheLoader

This cache loader allows to delegate loads and stores to another instance of JBoss Cache, which could reside (a) in the same address space, (b) in a different process on the same host, or (c) in a different process on a different host.

A TcpDelegatingCacheLoader talks to a remote `org.jboss.cache.loader.tcp.TcpCacheServer` , which can be a standalone process started on the command line, or embedded as an MBean inside JBoss AS. The `TcpCacheServer` has a reference to another JBoss Cache instance, which it can create itself, or which is given to it (e.g. by JBoss, using dependency injection).

The TcpDelegatingCacheLoader is configured with the host and port of the remote TcpCacheServer, and uses this to communicate to it.

The configuration looks as follows:

```
<attribute name="CacheLoaderConfiguration">
<config>
   <cacheloader>
      <class>org.jboss.cache.loader.TcpDelegatingCacheLoader</class>
      <properties>
         host=myRemoteServer
         port=7500
      </properties>
   </cacheloader>
</config>
</attribute>
```

This means this instance of JBoss Cache will delegate all load and store requests to the remote TcpCacheServer running on `myRemoteServer:7500`.

A typical use case could be multiple replicated instances of JBoss Cache in the same cluster, all delegating to the same TcpCacheServer instance. The TcpCacheServer might itself delegate to a database via JDBCCacheLoader, but the point here is that - if we have 5 nodes all accessing the same dataset - they will load the data from the TcpCacheServer, which has do execute one SQL statement per unloaded data set. If the nodes went directly to the database, then we'd have the same SQL executed multiple times. So TcpCacheServer serves as a natural cache in front of the DB (assuming that a network round trip is faster than a DB access (which usually also include a network round trip)).

To alleviate single point of failure, we could configure several cache loaders. The first cache loader is a ClusteredCacheLoader, the second a TcpDelegatingCacheLoader, and the last a JDBCacheLoader, effectively defining our cost of access to a cache in increasing order.

8.3.5. Transforming Cache Loaders

The way cached data is written to `FileCacheLoader` and `JDBCCacheLoader` based cache stores has changed in JBoss Cache 2.0 in such way that these cache loaders now write and read data using the same marhalling framework used to replicate data accross the network. Such change is trivial for replication purpouses as it just requires the rest of the nodes to understand this format. However, changing the format of the data in cache stores brings up a new problem: how do users, which have their data stored in JBoss Cache 1.x.x format, migrate their stores to JBoss Cache 2.0 format?

With this in mind, JBoss Cache 2.0 comes with two cache loader implementations called `org.jboss.cache.loader.TransformingFileCacheLoader` and `org.jboss.cache.loader.TransformingJDBCCacheLoader` located within the optional jbosscache-cacheloader-migration.jar file. These are one-off cache loaders that read data from the cache store in JBoss Cache 1.x.x format and write data to cache stores in JBoss Cache 2.0 format.

The idea is for users to modify their existing cache configuration file(s) momentarily to use these cache loaders and for them to create a small Java application that creates an instance of this cache, recursively reads the entire cache and writes the data read back into the cache. Once the data is transformed, users can revert back to their original cache configuration file(s). In order to help the users with this task, a cache loader migration example has been constructed which can be located under the `examples/cacheloader-migration` directory within the JBoss Cache distribution. This example, called `examples.TransformStore`, is independent of the actual data stored in the cache as it writes back whatever it was read recursively. It is highly recommended that anyone interested in porting their data run this example first, which contains a `readme.txt` file with detailed information about the example itself, and also use it as base for their own application.

8.4. Cache Passivation

A cache loader can be used to enforce node passivation and activation on eviction in a cache.

Cache Passivation is the process of removing an object from in-memory cache and writing it to a secondary data store (e.g., file system, database) on eviction. *Cache Activation* is the process of restoring an object

from the data store into the in-memory cache when it's needed to be used. In both cases, the configured cache loader will be used to read from the data store and write to the data store.

When an eviction policy in effect evicts a node from the cache, if passivation is enabled, a notification that the node is being passivated will be emitted to the cache listeners and the node and its children will be stored in the cache loader store. When a user attempts to retrieve a node that was evicted earlier, the node is loaded (lazy loaded) from the cache loader store into memory. When the node and its children have been loaded, they're removed from the cache loader and a notification is emitted to the cache listeners that the node has been activated.

To enable cache passivation/activation, you can set `passivation` to true. The default is `false` . When passivation is used, only the first cache loader configured is used and all others are ignored.

8.4.1. Cache Loader Behavior with Passivation Disabled vs. Enabled

When passivation is disabled, whenever an element is modified, added or removed, then that modification is persisted in the backend store via the cache loader. There is no direct relationship between eviction and cache loading. If you don't use eviction, what's in the persistent store is basically a copy of what's in memory. If you do use eviction, what's in the persistent store is basically a superset of what's in memory (i.e. it includes nodes that have been evicted from memory).

When passivation is enabled, there is a direct relationship between eviction and the cache loader. Writes to the persistent store via the cache loader only occur as part of the eviction process. Data is deleted from the persistent store when the application reads it back into memory. In this case, what's in memory and what's in the persistent store are two subsets of the total information set, with no intersection between the subsets.

Following is a simple example, showing what state is in RAM and in the persistent store after each step of a 6 step process:

1. Insert /A
2. Insert /B
3. Eviction thread runs, evicts /A
4. Read /A
5. Eviction thread runs, evicts /B
6. Remove /B

When passivation is disabled:

```
1) RAM: /A     Disk: /A
2) RAM: /A, /B Disk: /A, /B
3) RAM: /B     Disk: /A, /B
4) RAM: /A, /B Disk: /A, /B
5) RAM: /A     Disk: /A, /B
6) RAM: /A     Disk: /A
```

When passivation is enabled:

```
1) RAM: /A      Disk:
2) RAM: /A, /B  Disk:
3) RAM: /B      Disk: /A
4) RAM: /A, /B  Disk:
5) RAM: /A      Disk: /B
6) RAM: /A      Disk:
```

8.5. Strategies

This section discusses different patterns of combining different cache loader types and configuration options to achieve specific outcomes.

8.5.1. Local Cache With Store

This is the simplest case. We have a JBoss Cache instance, whose cache mode is `LOCAL` , therefore no replication is going on. The cache loader simply loads non-existing elements from the store and stores modifications back to the store. When the cache is started, depending on the `preload` element, certain data can be preloaded, so that the cache is partly warmed up.

8.5.2. Replicated Caches With All Caches Sharing The Same Store

The following figure shows 2 JBoss Cache instances sharing the same backend store:

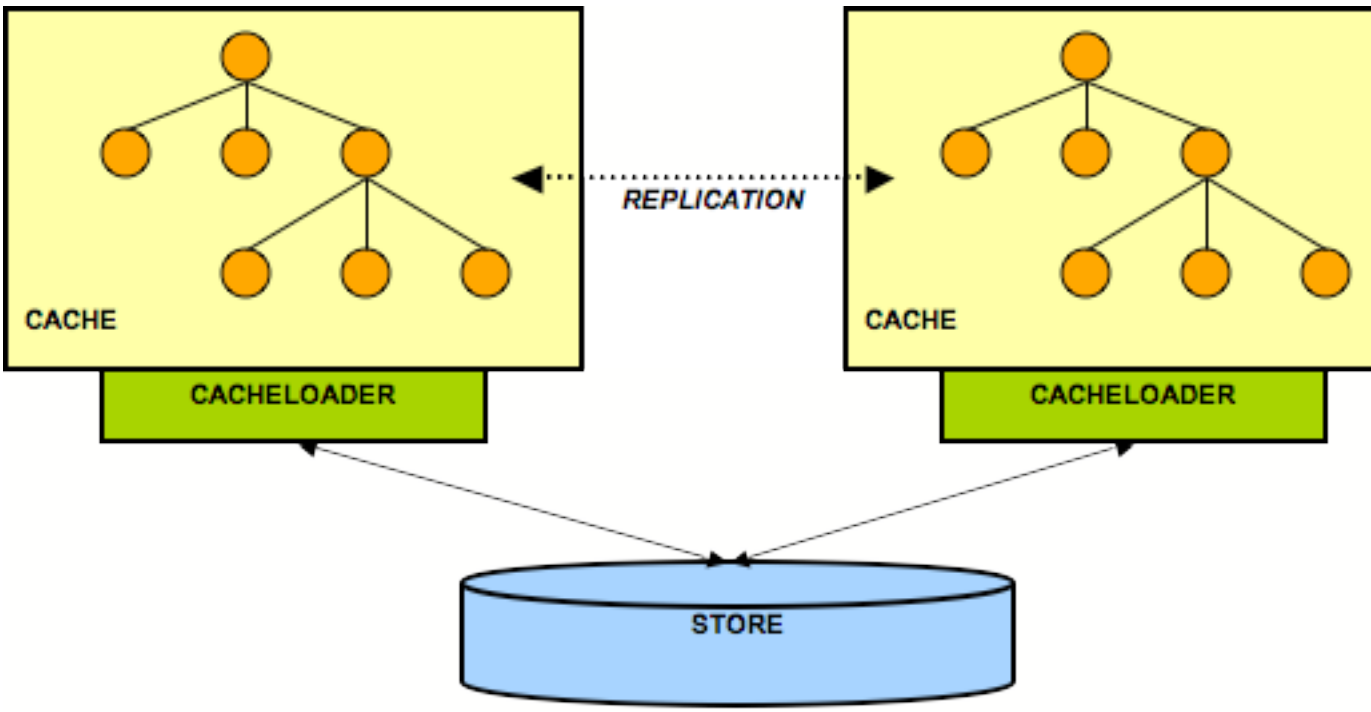


Figure 8.2. 2 nodes sharing a backend store

Both nodes have a cache loader that accesses a common shared backend store. This could for example be a shared filesystem (using the FileCacheLoader), or a shared database. Because both nodes access

the same store, they don't necessarily need state transfer on startup. [1] Rather, the `FetchInMemoryState` attribute could be set to false, resulting in a 'cold' cache, that gradually warms up as elements are accessed and loaded for the first time. This would mean that individual caches in a cluster might have different in-memory state at any given time (largely depending on their preloading and eviction strategies).

When storing a value, the writer takes care of storing the change in the backend store. For example, if node1 made change C1 and node2 C2, then node1 would tell its cache loader to store C1, and node2 would tell its cache loader to store C2.

8.5.3. Replicated Caches With Only One Cache Having A Store

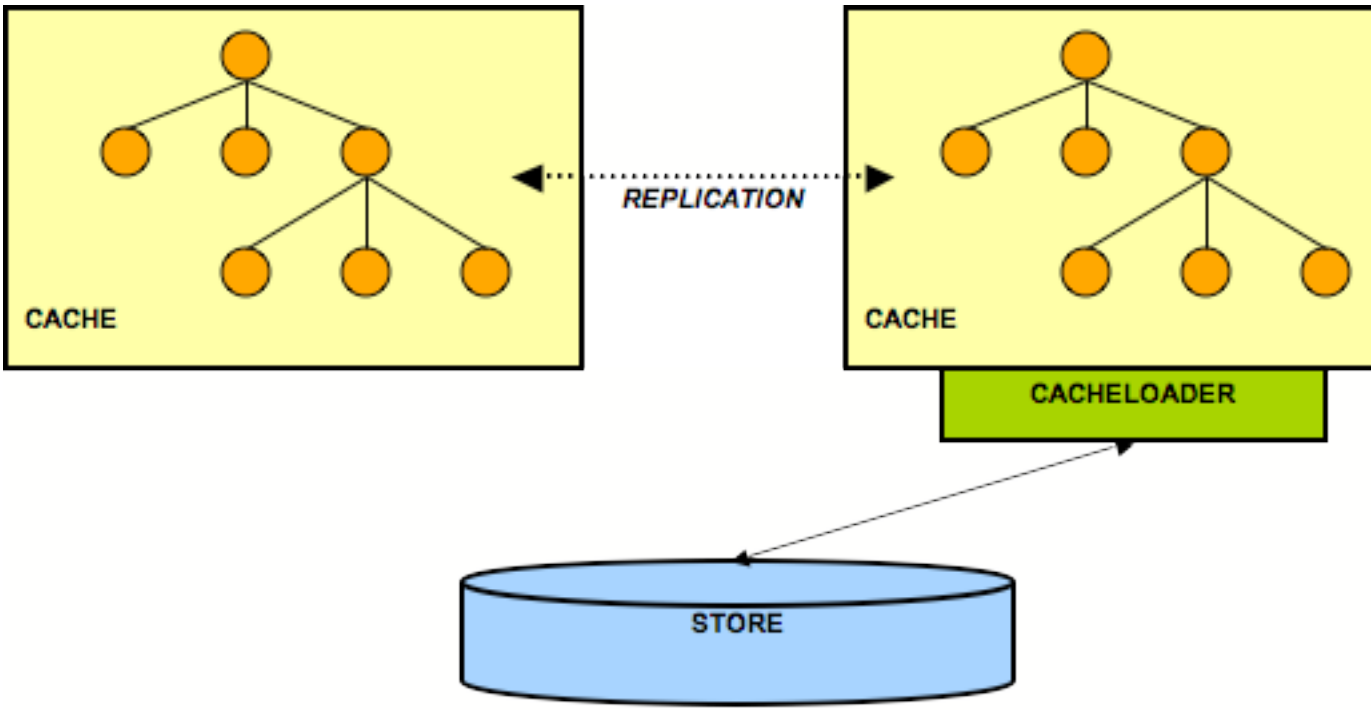


Figure 8.3. 2 nodes but only one accesses the backend store

This is a similar case to the previous one, but here only one node in the cluster interacts with a backend store via its cache loader. All other nodes perform in-memory replication. The idea here is all application state is kept in memory in each node, with the existence of multiple caches making the data highly available. (This assumes that a client that needs the data is able to somehow fail over from one cache to another.) The single persistent backend store then provides a backup copy of the data in case all caches in the cluster fail or need to be restarted.

Note that here it may make sense for the cache loader to store changes asynchronously, that is *not* on the caller's thread, in order not to slow down the cluster by accessing (for example) a database. This is a non-issue when using asynchronous replication.

A weakness with this architecture is that the cache with access to the cache loader becomes a single point of failure. Furthermore, if the cluster is restarted, the cache with the cache loader must be started first

[1] Of course they can enable state transfer, if they want to have a warm or hot cache after startup.

(easy to forget). A solution to the first problem is to configure a cache loader on each node, but set the `singletonStore` configuration to `true`. With this kind of setup, one but only one node will always be writing to a persistent store. However, this complicates the restart problem, as before restarting you need to determine which cache was writing before the shutdown/failure and then start that cache first.

8.5.4. Replicated Caches With Each Cache Having Its Own Store

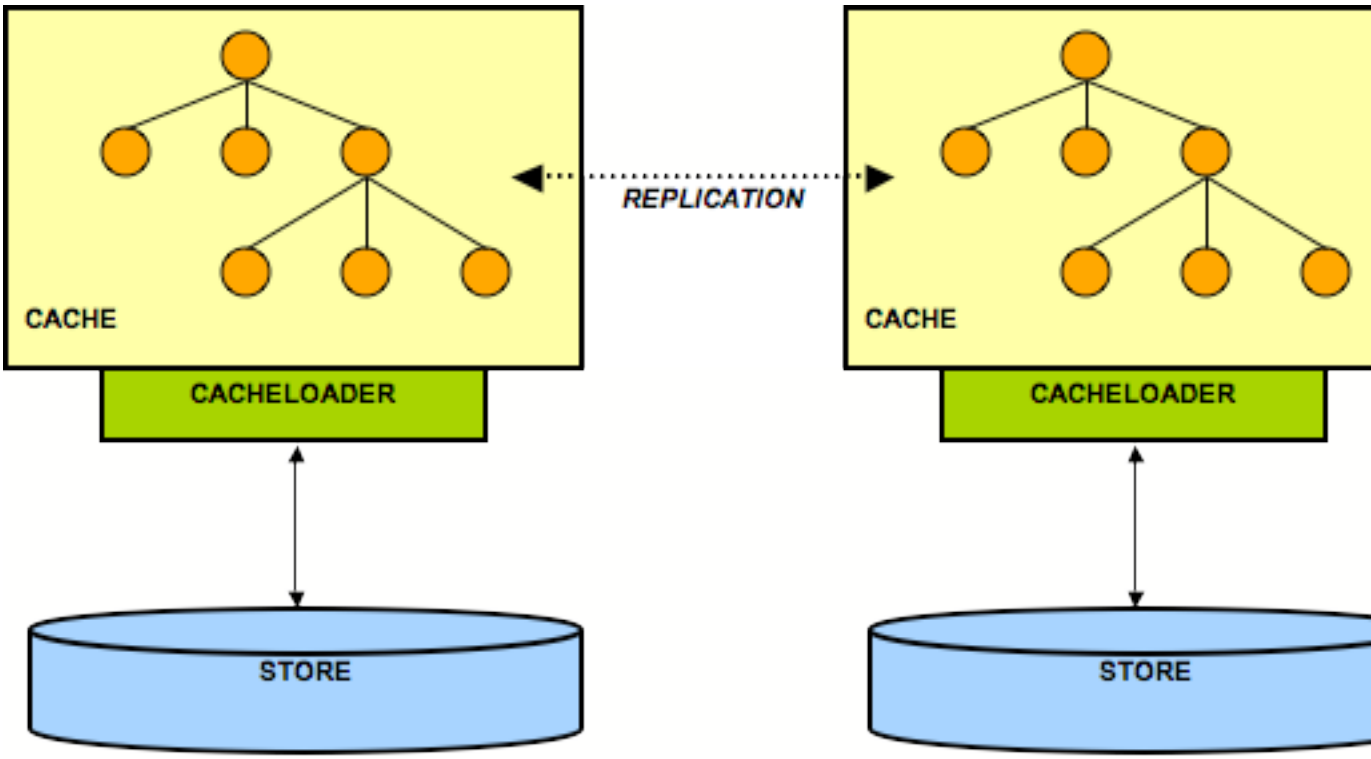


Figure 8.4. 2 nodes each having its own backend store

Here, each node has its own datastore. Modifications to the cache are (a) replicated across the cluster and (b) persisted using the cache loader. This means that all datastores have exactly the same state. When replicating changes synchronously and in a transaction, the two phase commit protocol takes care that all modifications are replicated and persisted in each datastore, or none is replicated and persisted (atomic updates).

Note that JBoss Cache is *not* an XA Resource, that means it doesn't implement recovery. When used with a transaction manager that supports recovery, this functionality is not available.

The challenge here is state transfer: when a new node starts it needs to do the following:

1. Tell the coordinator (oldest node in a cluster) to send it the state. This is always a full state transfer, overwriting any state that may already be present.

2. The coordinator then needs to wait until all in-flight transactions have completed. During this time, it will not allow for new transactions to be started.

3. Then the coordinator asks its cache loader for the entire state using `loadEntireState()`. It then sends back that state to the new node.

4. The new node then tells its cache loader to store that state in its store, overwriting the old state. This is the `CacheLoader.storeEntireState()` method

5. As an option, the transient (in-memory) state can be transferred as well during the state transfer.

6. The new node now has the same state in its backend store as everyone else in the cluster, and modifications received from other nodes will now be persisted using the local cache loader.

8.5.5. Hierarchical Caches

If you need to set up a hierarchy within a single JVM, you can use the `LocalDelegatingCacheLoader`. This type of hierarchy can currently only be set up programmatically.

Hierarchical caches could also be set up spanning more than one JVM or server, using the `TcpDelegatingCacheLoader`.

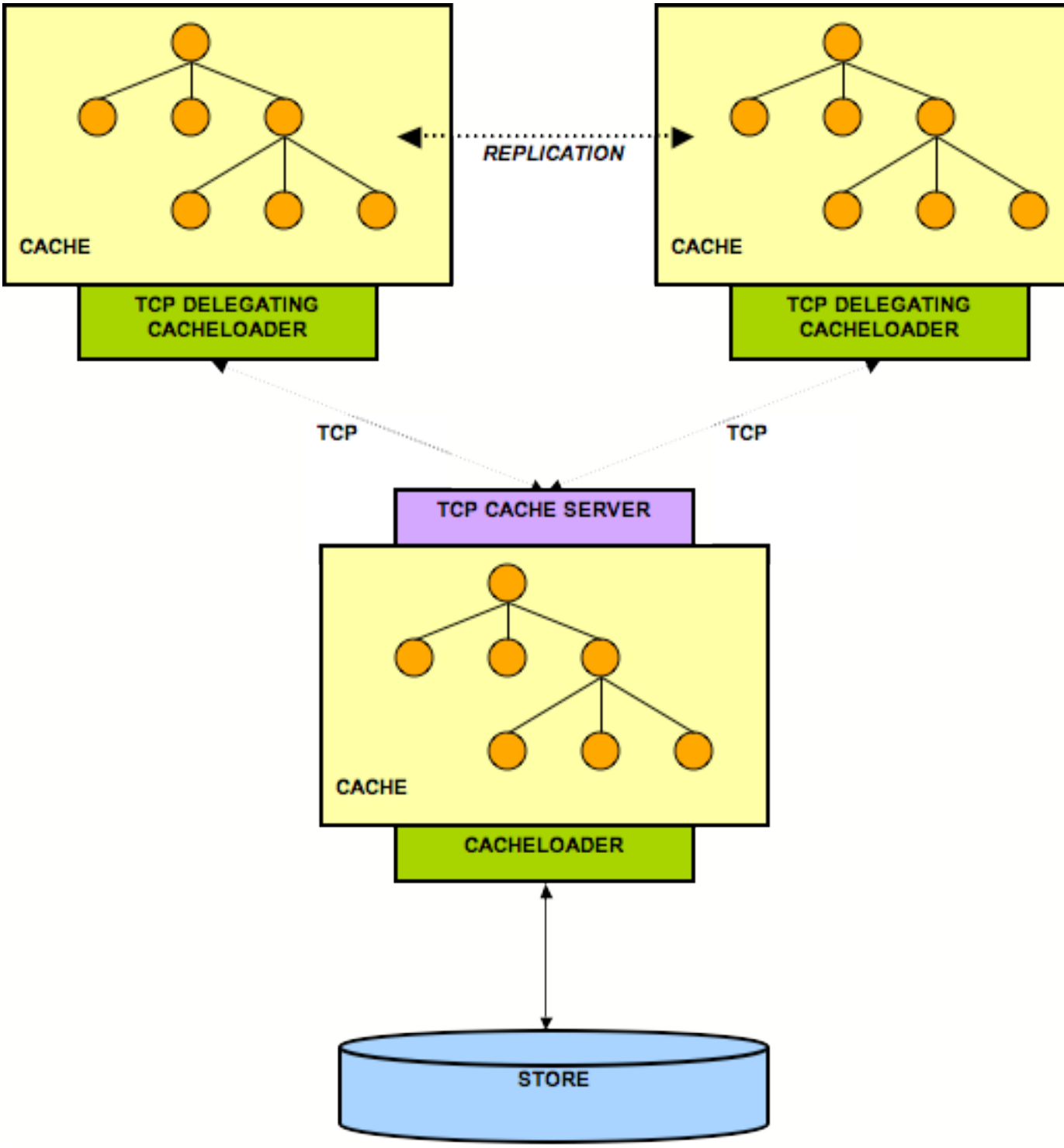


Figure 8.5. TCP delegating cache loader

8.5.6. Multiple Cache Loaders

You can set up more than one cache loader in a chain. Internally, a delegating `ChainingCacheLoader` is used, with references to each cache loader you have configured. Use cases vary depending on the type of cache loaders used in the chain. One example is using a filesystem based cache loader, colocated on the same host as the JVM, used as an overflow for memory. This ensures data is available relatively easily

and with low cost. An additional remote cache loader, such as a `TcpDelegatingCacheLoader` provides resilience between server restarts.

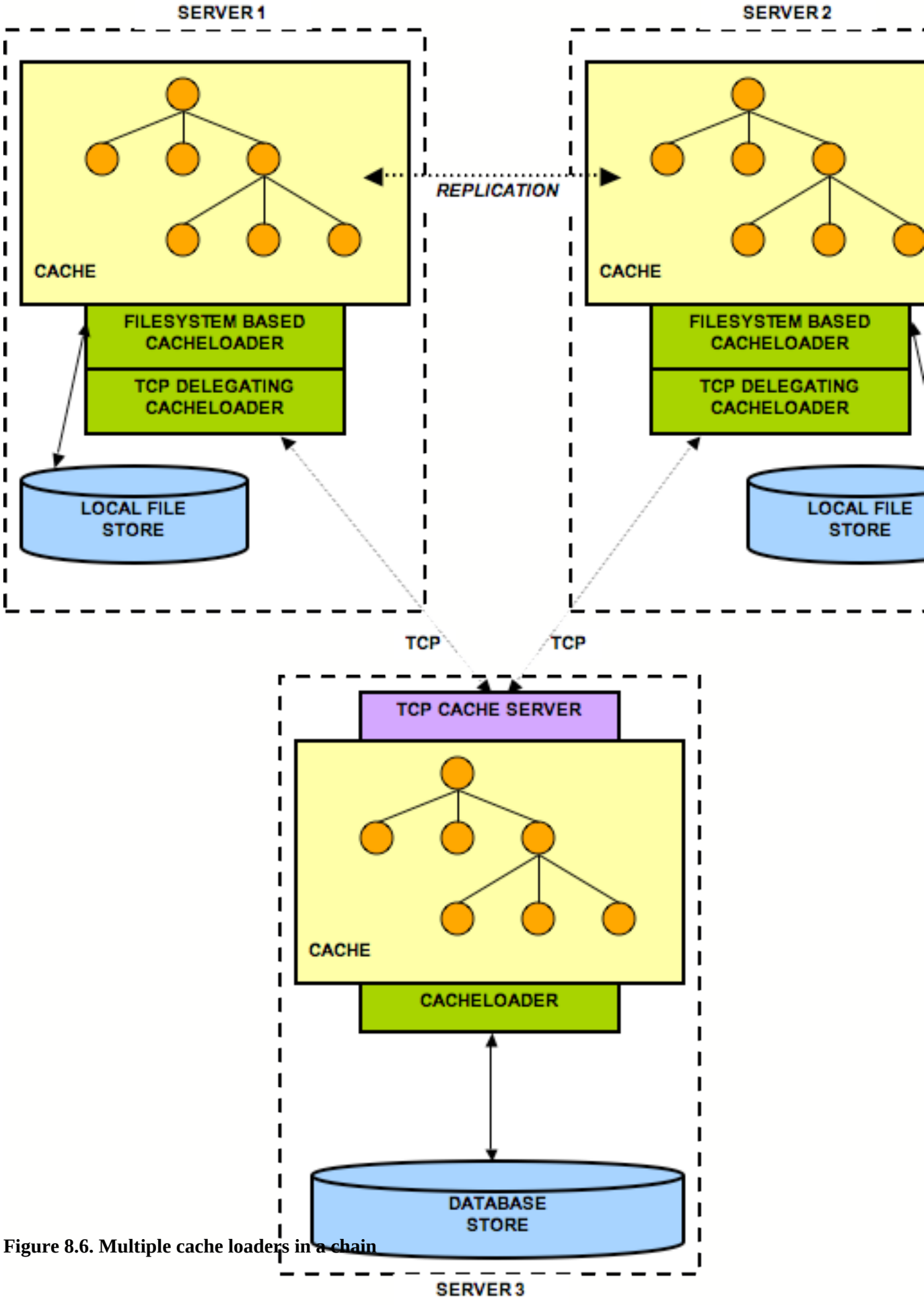


Figure 8.6. Multiple cache loaders in a chain

4

Eviction Policies

Eviction policies control JBoss Cache's memory management by managing how many nodes are allowed to be stored in memory and their life spans. Memory constraints on servers mean cache cannot grow indefinitely, so policies need to be in place to restrict the size of the cache. Eviction policies are most often used alongside cache loaders cache loaders .

9.1. Configuring Eviction Policies

9.1.1. Basic Configuration

The basic eviction policy configuration element looks like:

```
...

<attribute name="EvictionConfig">
   <config>
      <attribute name="wakeUpIntervalSeconds">3</attribute>

      <!-- This defaults to 200000 if not specified -->
      <attribute name="eventQueueSize">100000</attribute>

      <!-- Name of the DEFAULT eviction policy class. -->
      <attribute name="policyClass">org.jboss.cache.eviction.LRUPolicy</attribute>

      <!-- Cache wide default -->
      <region name="/_default_">
         <attribute name="maxNodes">100</attribute>
      </region>

      <!-- override policy used for this region -->
      <region name="/org/jboss/data" policyClass="org.jboss.cache.eviction.LRUPolicy">
         <attribute name="maxNodes">250</attribute>
         <attribute name="minTimeToLiveSeconds">10</attribute>
      </region>

      <!-- We expect a lot of events for this region,
           so override the default event queue size -->
      <region name="/org/jboss/test/data" eventQueueSize="500000">
         <attribute name="maxNodes">60000</attribute>
      </region>
```

```
      </config>
   </attribute>

   ...
```

- `wakeUpIntervalSeconds` - this required parameter defines how often the eviction thread runs
- `eventQueueSize` - this optional parameter defines the size of the queue which holds eviction events. If your eviction thread does not run often enough, you may need to increase this. This can be overridden on a per-region basis.
- `policyClass` - this is required, unless you set individual policyClass attributes on each and every region. This defines the eviction policy to use if one is not defined for a region.

9.1.2. Eviction Regions

The concept of regions and the `Region` class were visited earlier when talking about marshalling. Regions also have another use, in that they are used to define the eviction policy used within the region. In addition to using a region-specific configuration, you can also configure a default, cache-wide eviction policy for nodes that do not fall into predefined regions or if you do not wish to define specific regions. It is important to note that when defining regions using the configuration XML file, all elements of the `Fqn` that defines the region are `java.lang.String` objects.

Looking at the eviction configuration snippet above, we see that a default region, `_default_` , holds attributes which apply to nodes that do not fall into any of the other regions defined.

For each region, you can define parameters which affect how the policy which applies to the region chooses to evict nodes. In the example above, the `LRUPolicy` allows a `maxNodes` parameter which defines how many nodes can exist in the region before it chooses to start evicting nodes. See the javadocs for each policy for a list of allowed parameters. It also defines a `minTimeToLiveSeconds` parameter, which defines a minimum time a node must exist in memory before being considered for eviction.

9.1.2.1. Overlapping Eviction Regions

It's possible to define regions that overlap. In other words, one region can be defined for */a/b/c* , and another defined for */a/b/c/d* (which is just the *d* subtree of the */a/b/c* sub-tree). The algorithm, in order to handle scenarios like this consistently, will always choose the first region it encounters. In this way, if the algorithm needed to decide how to handle */a/b/c/d/e* , it would start from there and work its way up the tree until it hits the first defined region - in this case */a/b/c/d* .

9.1.3. Resident Nodes

Nodes marked as resident (using `Node.setResident()` API) will be ignored by the eviction policies both when checking whether to trigger the eviction and when proceeding with the actual eviction of nodes. E.g. if a region is configured to have a maximum of 10 nodes, resident nodes won't be counted when deciding whether to evict nodes in that region. In addition, resident nodes will not be considered for eviction when the region's eviction threshold is reached.

In order to mark a node as resident the `Node.setResident()` API should be used. By default, the newly created nodes are not resident. The `resident` attribute of a node is neither replicated, persisted nor transaction-aware.

A sample use case for resident nodes would be ensuring "path" nodes don't add "noise" to an eviction policy. E.g.

```
...
          Map lotsOfData = generateData();
          cache.put("/a/b/c", lotsOfData);
          cache.getRoot().getChild("/a").setResident(true);
          cache.getRoot().getChild("/a/b").setResident(true);
...
```

In this example, the nodes `/a` and `/a/b` are paths which exist solely to support the existence of node `/a/b/c` and don't hold any data themselves. As such, they are good candidates for being marked as resident. This would lead to better memory management as no eviction events would be generated when accessing `/a` and `/a/b`.

N.B. when adding attributes to a resident node, e.g. `cache.put("/a", "k", "v")` in the above example, it would make sense to mark the nodes as non-resident again and let them be considered for eviction..

9.1.4. Programmatic Configuration

Configuring eviction using the `Configuration` object entails the use of the `org.jboss.cache.config.EvictionConfig` bean, which is passed into `Configuration.setEvictionConfig()` . See the chapter on Configuration for more on building a `Configuration` programatically.

The use of simple POJO beans to represent all elements in a cache's configuration also makes it fairly easy to programatically add eviction regions after the cache is started . For example, assume we had an existing cache configured via XML with the EvictionConfig element shown above. Now at runtime we wished to add a new eviction region named "/org/jboss/fifo", using `LRUPolicy` but a different number of `maxNodes` :

```
Fqn fqn = Fqn.fromString("/org/jboss/fifo");

// Create a configuration for an LRUPolicy
LRUConfiguration lruc = new LRUConfiguration();
lruc.setMaxNodes(10000);

// Create the region and set the config
Region region = cache.getRegion(fqn, true);
region.setEvictionPolicy(lruc);
```

9.2. Shipped Eviction Policies

9.2.1. LRUPolicy - Least Recently Used

`org.jboss.cache.eviction.LRUPolicy` controls both the node lifetime and age. This policy guarantees a constant order (`O (1)`) for adds, removals and lookups (visits). It has the following configuration parameters:

- `maxNodes` - This is the maximum number of nodes allowed in this region. 0 denotes no limit.
- `timeToLiveSeconds` - The amount of time a node is not written to or read (in seconds) before the node is swept away. 0 denotes no limit.
- `maxAgeSeconds` - Lifespan of a node (in seconds) regardless of idle time before the node is swept away. 0 denotes no limit.
- `minTimeToLiveSeconds` - the minimum amount of time a node must be allowed to live after being accessed before it is allowed to be considered for eviction. 0 denotes that this feature is disabled, which is the default value.

9.2.2. FIFOPolicy - First In, First Out

`org.jboss.cache.eviction.FIFOPolicy` controls the eviction in a proper first in first out order. This policy guarantees a constant order (`O (1)`) for adds, removals and lookups (visits). It has the following configuration parameters:

- `maxNodes` - This is the maximum number of nodes allowed in this region. 0 denotes no limit.
- `minTimeToLiveSeconds` - the minimum amount of time a node must be allowed to live after being accessed before it is allowed to be considered for eviction. 0 denotes that this feature is disabled, which is the default value.

9.2.3. MRUPolicy - Most Recently Used

`org.jboss.cache.eviction.MRUPolicy` controls the eviction in based on most recently used algorithm. The most recently used nodes will be the first to evict with this policy. This policy guarantees a constant order (`O (1)`) for adds, removals and lookups (visits). It has the following configuration parameters:

- `maxNodes` - This is the maximum number of nodes allowed in this region. 0 denotes no limit.
- `minTimeToLiveSeconds` - the minimum amount of time a node must be allowed to live after being accessed before it is allowed to be considered for eviction. 0 denotes that this feature is disabled, which is the default value.

9.2.4. LFUPolicy - Least Frequently Used

`org.jboss.cache.eviction.LFUPolicy` controls the eviction in based on least frequently used algorithm. The least frequently used nodes will be the first to evict with this policy. Node usage starts at 1 when a node is first added. Each time it is visted, the node usage counter increments by 1. This number is used to

determine which nodes are least frequently used. LFU is also a sorted eviction algorithm. The underlying EvictionQueue implementation and algorithm is sorted in ascending order of the node visits counter. This class guarantees a constant order (`O (1)`) for adds, removal and searches. However, when any number of nodes are added/visited to the queue for a given processing pass, a single quasilinear (`O (n * log n)`) operation is used to resort the queue in proper LFU order. Similarly if any nodes are removed or evicted, a single linear (`O (n)`) pruning operation is necessary to clean up the EvictionQueue. LFU has the following configuration parameters:

- `maxNodes` - This is the maximum number of nodes allowed in this region. 0 denotes no limit.
- `minNodes` - This is the minimum number of nodes allowed in this region. This value determines what the eviction queue should prune down to per pass. e.g. If minNodes is 10 and the cache grows to 100 nodes, the cache is pruned down to the 10 most frequently used nodes when the eviction timer makes a pass through the eviction algorithm.
- `minTimeToLiveSeconds` - the minimum amount of time a node must be allowed to live after being accessed before it is allowed to be considered for eviction. 0 denotes that this feature is disabled, which is the default value.

9.2.5. ExpirationPolicy

`org.jboss.cache.eviction.ExpirationPolicy` is a policy that evicts nodes based on an absolute expiration time. The expiration time is indicated using the `org.jboss.cache.Node.put()` method, using a String key `expiration` and the absolute time as a `java.lang.Long` object, with a value indicated as milliseconds past midnight January 1st, 1970 UTC (the same relative time as provided by `java.lang.System.currentTimeMillis()`).

This policy guarantees a constant order (`O (1)`) for adds and removals. Internally, a sorted set (TreeSet) containing the expiration time and Fqn of the nodes is stored, which essentially functions as a heap.

This policy has the following configuration parameters:

- `expirationKeyName` - This is the Node key name used in the eviction algorithm. The configuration default is `expiration` .
- `maxNodes` - This is the maximum number of nodes allowed in this region. 0 denotes no limit.

The following listing shows how the expiration date is indicated and how the policy is applied:

```
Cache cache = DefaultCacheFactory.createCache();
Fqn fqn1 = Fqn.fromString("/node/1");
Long future = new Long(System.currentTimeMillis() + 2000);

// sets the expiry time for a node
cache.getRoot().addChild(fqn1).put(ExpirationConfiguration.EXPIRATION_KEY, future);

assertTrue(cache.getRoot().hasChild(fqn1));
Thread.sleep(5000);
```

```
   // after 5 seconds, expiration completes
   assertFalse(cache.getRoot().hasChild(fqn1));
```

Note that the expiration time of nodes is only checked when the region manager wakes up every `wakeUpIntervalSeconds` , so eviction may happen a few seconds later than indicated.

9.2.6. ElementSizePolicy - Eviction based on number of key/value pairs in a node

`org.jboss.cache.eviction.ElementSizePolicy` controls the eviction in based on the number of key/value pairs in the node. Nodes The most recently used nodes will be the first to evict with this policy. It has the following configuration parameters:

- `maxNodes` - This is the maximum number of nodes allowed in this region. 0 denotes no limit.
- `maxElementsPerNode` - This is the trigger number of attributes per node for the node to be selected for eviction. 0 denotes no limit.
- `minTimeToLiveSeconds` - the minimum amount of time a node must be allowed to live after being accessed before it is allowed to be considered for eviction. 0 denotes that this feature is disabled, which is the default value.

9.3. Writing Your Own Eviction Policies

9.3.1. Eviction Policy Plugin Design

The design of the JBoss Cache eviction policy framework is based on an `EvictionInterceptor` to handle cache events and relay them back to the eviction policies. During the cache start up, an `EvictionInterceptor` will be added to the cache interceptor stack if the eviction policy is specified. Then whenever a node is added, removed, evicted, or visited, the `EvictionInterceptor` will maintain state statistics and information will be relayed to each individual eviction region.

There is a single eviction thread (timer) that will run at a configured interval. This thread will make calls into each of the policy providers and inform it of any aggregated adds, removes and visits (gets) events to the cache during the configured interval. The eviction thread is responsible for kicking off the eviction policy processing (a single pass) for each configured eviction cache region.

9.3.2. Interfaces to implement

In order to implement an eviction policy, the following interfaces must be implemented:

- `org.jboss.cache.eviction.EvictionPolicy`
- `org.jboss.cache.eviction.EvictionAlgorithm`
- `org.jboss.cache.eviction.EvictionQueue`
- `org.jboss.cache.config.EvictionPolicyConfig`

When compounded together, each of these interface implementations define all the underlying mechanics necessary for a complete eviction policy implementation.

Note that:

- The `EvictionPolicyConfig` implementation should maintain getter and setter methods for whatever configuration properties the policy supports (e.g. for `LRUConfiguration` among others there is a `int getMaxNodes()` and a `setMaxNodes(int)`). When the "EvictionConfig" section of an XML configuration is parsed, these properties will be set by reflection.

Alternatively, the implementation of a new eviction policy provider can be simplified by extending `BaseEvictionPolicy` and `BaseEvictionAlgorithm` . Or for properly sorted EvictionAlgorithms (sorted in eviction order - see `LFUAlgorithm`) extending `BaseSortedEvictionAlgorithm` and implementing `SortedEvictionQueue` takes care of most of the common functionality available in a set of eviction policy provider classes

Note that:

- The `BaseEvictionAlgorithm` class maintains a processing structure. It will process the ADD, REMOVE, and VISIT events queued by the region first. It also maintains an collection of items that were not properly evicted during the last go around because of held locks. That list is pruned. Finally, the EvictionQueue itself is pruned for entries that should be evicted based upon the configured eviction rules for the region.

- The `BaseSortedEvictionAlgorithm` class will maintain a boolean through the algorithm processing that will determine if any new nodes were added or visited. This allows the Algorithm to determine whether to resort the eviction queue items (in first to evict order) or to skip the potentially expensive sorting if there have been no changes to the cache in this region.

- The `SortedEvictionQueue` interface defines the contract used by the `BaseSortedEvictionAlgorithm` abstract class that is used to resort the underlying queue. Again, the queue sorting should be sorted in first to evict order. The first entry in the list should evict before the last entry in the queue. The last entry in the queue should be the last entry that will require eviction.

5

Transactions and Concurrency

10.1. Concurrent Access

JBoss Cache is a thread safe caching API, and uses its own efficient mechanisms of controlling concurrent access. It uses a pessimistic locking scheme by default for this purpose. Optimistic locking may alternatively be used, and is discussed later.

10.1.1. Locks

Locking is done internally, on a node-level. For example when we want to access "/a/b/c", a lock will be acquired for nodes "a", "b" and "c". When the same transaction wants to access "/a/b/c/d", since we already hold locks for "a", "b" and "c", we only need to acquire a lock for "d".

Lock owners are either transactions (call is made within the scope of an existing transaction) or threads (no transaction associated with the call). Regardless, a transaction or a thread is internally transformed into an instance of `GlobalTransaction` , which is used as a globally unique identifier for modifications across a cluster. E.g. when we run a two-phase commit protocol across the cluster, the `GlobalTransaction` uniquely identifies a unit of work across a cluster.

Locks can be read or write locks. Write locks serialize read and write access, whereas read-only locks only serialize read access. When a write lock is held, no other write or read locks can be acquired. When a read lock is held, others can acquire read locks. However, to acquire write locks, one has to wait until all read locks have been released. When scheduled concurrently, write locks always have precedence over read locks. Note that (if enabled) read locks can be upgraded to write locks.

Using read-write locks helps in the following scenario: consider a tree with entries "/a/b/n1" and "/a/b/n2". With write-locks, when Tx1 accesses "/a/b/n1", Tx2 cannot access "/a/b/n2" until Tx1 has completed and released its locks. However, with read-write locks this is possible, because Tx1 acquires read-locks for "/a/b" and a read-write lock for "/a/b/n1". Tx2 is then able to acquire read-locks for "/a/b" as well, plus a read-write lock for "/a/b/n2". This allows for more concurrency in accessing the cache.

10.1.2. Pessimistic locking

By default, JBoss Cache uses pessimistic locking. Locking is not exposed directly to user. Instead, a transaction isolation level which provides different locking behaviour is configurable.

10.1.2.1. Isolation levels

JBoss Cache supports the following transaction isolation levels, analogous to database ACID isolation levels. A user can configure an instance-wide isolation level of NONE, READ_UNCOMMITTED,

READ_COMMITTED, REPEATABLE_READ, or SERIALIZABLE. REPEATABLE_READ is the default isolation level used.

1. NONE. No transaction support is needed. There is no locking at this level, e.g., users will have to manage the data integrity. Implementations use no locks.

2. READ_UNCOMMITTED. Data can be read anytime while write operations are exclusive. Note that this level doesn't prevent the so-called "dirty read" where data modified in Tx1 can be read in Tx2 before Tx1 commits. In other words, if you have the following sequence,

   ```
   Tx1     Tx2
    W
             R
   ```

   using this isolation level will not prevent Tx2 read operation. Implementations typically use an exclusive lock for writes while reads don't need to acquire a lock.

3. READ_COMMITTED. Data can be read any time as long as there is no write. This level prevents the dirty read. But it doesn't prevent the so-called 'non-repeatable read' where one thread reads the data twice can produce different results. For example, if you have the following sequence,

   ```
   Tx1     Tx2
    R
             W
    R
   ```

   where the second read in Tx1 thread will produce different result.

   Implementations usually use a read-write lock; reads succeed acquiring the lock when there are only reads, writes have to wait until there are no more readers holding the lock, and readers are blocked acquiring the lock until there are no more writers holding the lock. Reads typically release the read-lock when done, so that a subsequent read to the same data has to re-acquire a read-lock; this leads to nonrepeatable reads, where 2 reads of the same data might return different values. Note that, the write only applies regardless of transaction state (whether it has been committed or not).

4. REPEATABLE_READ. Data can be read while there is no write and vice versa. This level prevents "non-repeatable read" but it does not completely prevent the so-called "phantom read" where new data can be inserted into the tree from another transaction. Implementations typically use a read-write lock. This is the default isolation level used.

5. SERIALIZABLE. Data access is synchronized with exclusive locks. Only 1 writer or reader can have the lock at any given time. Locks are released at the end of the transaction. Regarded as very poor for performance and thread/transaction concurrency.

10.1.2.2. Insertion and Removal of Nodes

By default, before inserting a new node into the tree or removing an existing node from the tree, JBoss Cache will only attempt to acquire a read lock on the new node's parent node. This approach does not treat child nodes as an integral part of a parent node's state. This approach allows greater concurrency if nodes are frequently added or removed, but at a cost of lesser correctness. For use cases where greater correctness is necessary, JBoss Cache provides a configuration option `LockParentForChildInsertRemove`. If this is set to `true`, insertions and removals of child nodes require the acquisition of a *write lock* on the parent node.

In addition to the above, in version 2.1.0 and above, JBoss Cache offers the ability to override this configuration on a per-node basis. See `Node.setLockForChildInsertRemove()` and it's corresponding javadocs for details.

10.1.3. Optimistic Locking

The motivation for optimistic locking is to improve concurrency. When a lot of threads have a lot of contention for access to the data tree, it can be inefficient to lock portions of the tree - for reading or writing - for the entire duration of a transaction as we do in pessimistic locking. Optimistic locking allows for greater concurrency of threads and transactions by using a technique called data versioning, explained here. Note that isolation levels (if configured) are ignored if optimistic locking is enabled.

10.1.3.1. Architecture

Optimistic locking treats all method calls as transactional [1]. Even if you do not invoke a call within the scope of an ongoing transaction, JBoss Cache creates an *implicit transaction* and commits this transaction when the invocation completes. Each transaction maintains a transaction workspace, which contains a copy of the data used within the transaction.

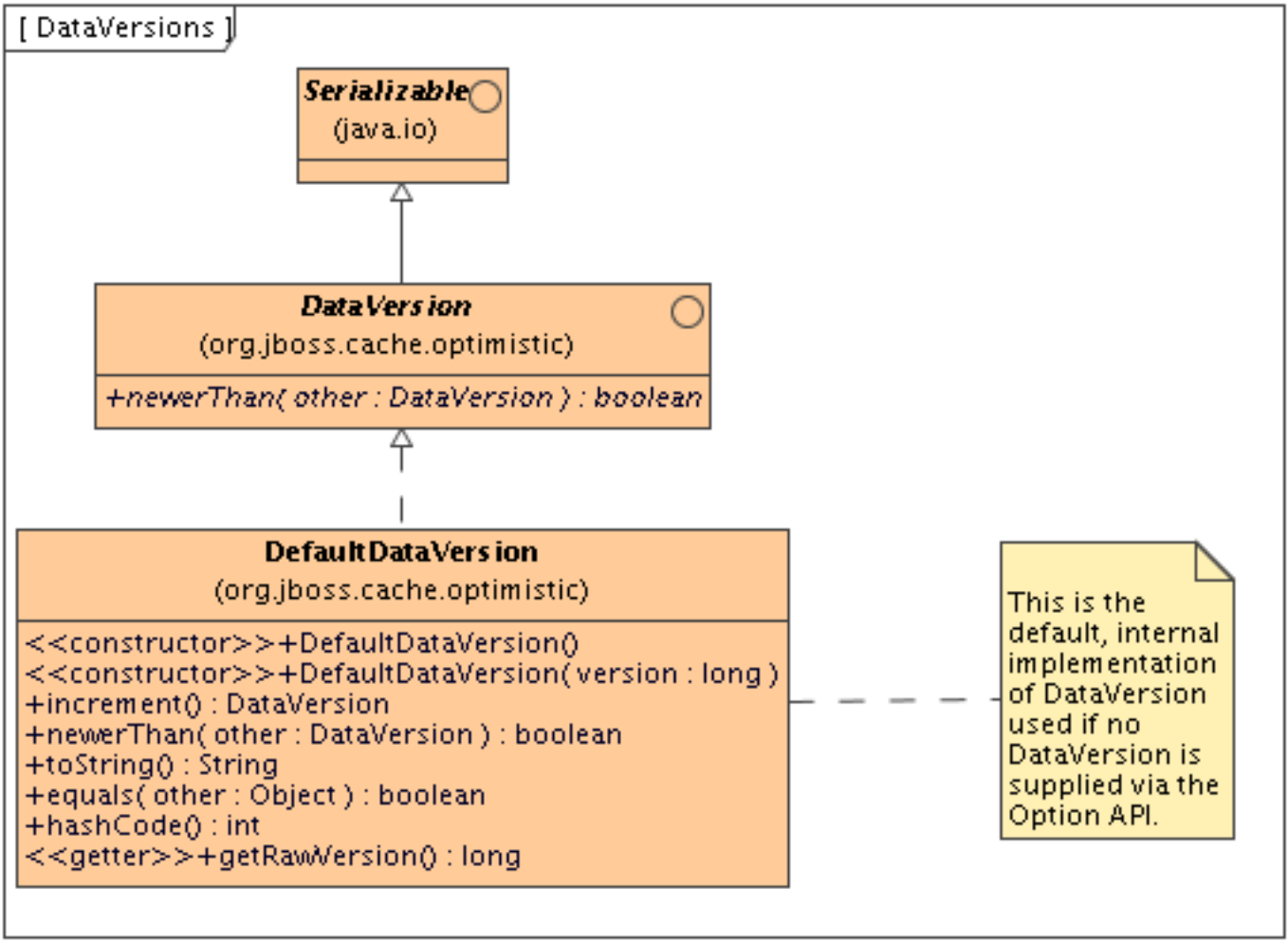
For example, if a transaction calls `cache.getRoot().getChild( Fqn.fromString("/a/b/c") )`, nodes a, b and c are copied from the main data tree and into the workspace. The data is versioned and all calls in the transaction work on the copy of the data rather than the actual data. When the transaction commits, its workspace is merged back into the underlying tree by matching versions. If there is a version mismatch - such as when the actual data tree has a higher version than the workspace, perhaps if another transaction were to access the same data, change it and commit before the first transaction can finish - the transaction throws a `RollbackException` when committing and the commit fails.

Optimistic locking uses the same locks we speak of above, but the locks are only held for a very short duration - at the start of a transaction to build a workspace, and when the transaction commits and has to merge data back into the tree.

So while optimistic locking may occasionally fail if version validations fail or may run slightly slower than pessimistic locking due to the inevitable overhead and extra processing of maintaining workspaces, versioned data and validating on commit, it does buy you a near-SERIALIZABLE degree of data integrity while maintaining a very high level of concurrency.

[1] Because of this requirement, you must always have a transaction manager configured when using optimistic locking.

10.1.3.2. Data Versioning



Optimistic locking makes use of the `DataVersion` interface (and an internal and default `DefaultDataVersion` implementation to keep a track of node versioning. In certain cases, where cached data is an in-memory representation of data from an external source such as a database, it makes sense to align the versions used in JBoss Cache with the versions used externally. As such, using the options API , it is possible to set the `DataVersion` you wish to use on a per-invocation basis, allowing you to implement the `DataVersion` interface to hold the versioning information obtained externally before putting your data into the cache.

10.1.3.3. Configuration

Optimistic locking is enabled by using the NodeLockingScheme XML attribute, and setting it to "OPTIMISTIC":

```
...
<!--
Node locking scheme:
OPTIMISTIC
PESSIMISTIC (default)
-->
```

```
<attribute name="NodeLockingScheme">OPTIMISTIC</attribute>
...
```

It is generally advisable that if you have an eviction policy defined along with optimistic locking, you define the eviction policy's `minTimeToLiveSeconds` parameter to be slightly greater than the transaction timeout value set in your transaction manager. This ensures that data versions in the cache are not evicted while transactions are in progress[2].

10.2. Transactional Support

JBoss Cache can be configured to use and participate in JTA compliant transactions. Alternatively, if transaction support is disabled, it is equivalent to setting AutoCommit to on where modifications are potentially [3] replicated after every change (if replication is enabled).

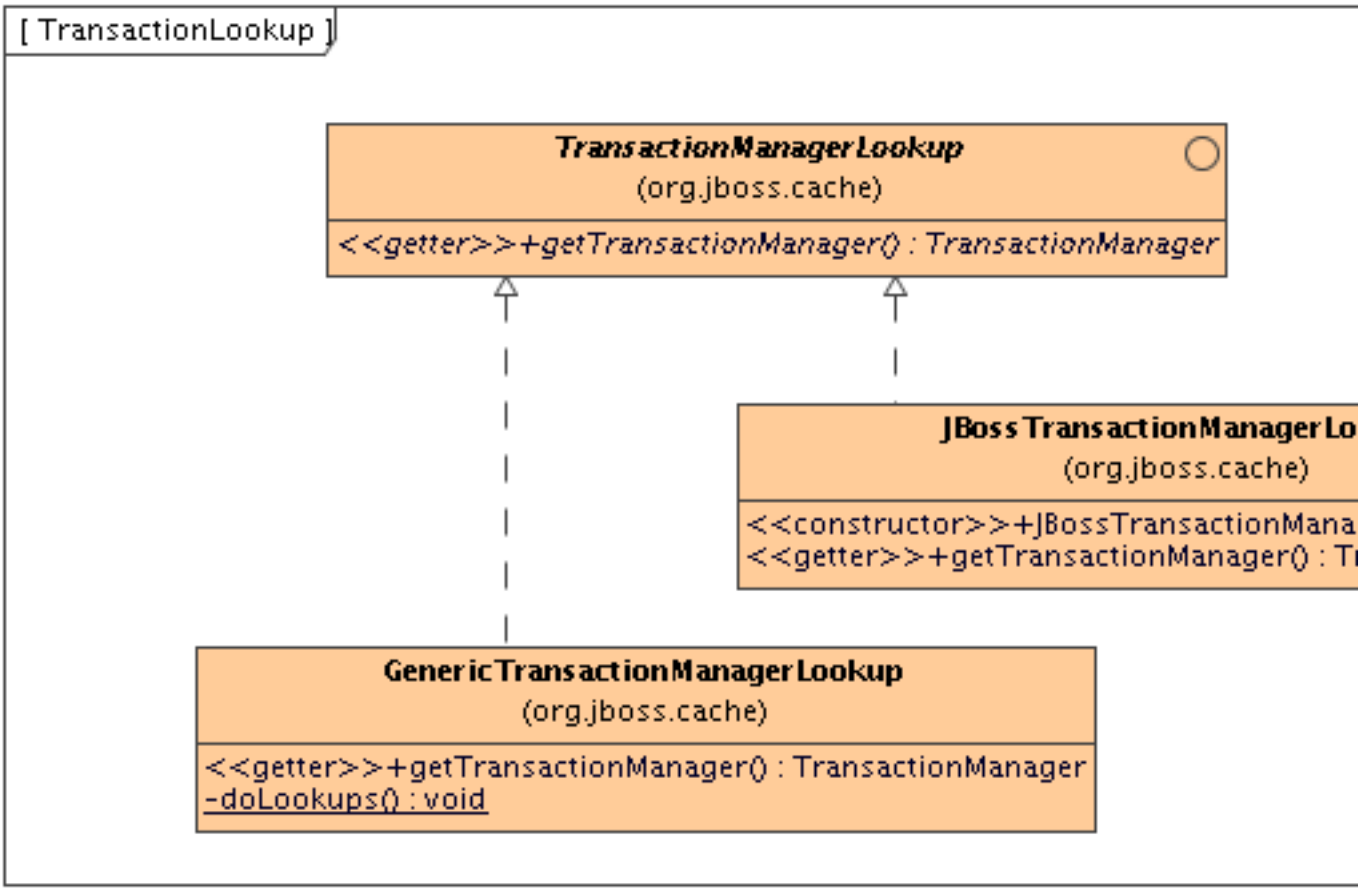
What JBoss Cache does on every incoming call is:

1. Retrieve the current `javax.transaction.Transaction` associated with the thread
2. If not already done, register a `javax.transaction.Synchronization` 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 environment's `javax.transaction.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` .

[2]See JBCACHE-1155 [http://jira.jboss.com/jira/browse/JBCACHE-1155]

[3]Depending on whether interval-based asynchronous replication is used



JBoss Cache ships with `JBossTransactionManagerLookup` and `GenericTransactionManagerLookup`. The `JBossTransactionManagerLookup` is able to bind to a running JBoss AS instance and retrieve a `TransactionManager` while the `GenericTransactionManagerLookup` is able to bind to most popular Java EE application servers and provide the same functionality. A dummy implementation - `DummyTransactionManagerLookup` - is also provided, primarily for unit tests. Being a dummy, this is just for demo and testing purposes and is not recommended for production use.

An alternative to configuring a `TransactionManagerLookup` is to programatically inject a reference to the `TransactionManager` into the `Configuration` object's `RuntimeConfig` element:

```
TransactionManager tm = getTransactionManager(); // magic method
cache.getConfiguration().getRuntimeConfig().setTransactionManager(tm);
```

Injecting the `TransactionManager` is the recommended approach when the `Configuration` is built by some sort of IOC container that already has a reference to the TM.

When the transaction commits, we initiate either a one- two-phase commit protocol. See replicated caches and transactions for details.

Part III. JBoss Cache References

This section contains technical references for easy looking up.

1

Configuration References

11.1. Sample XML Configuration File

This is what a typical XML configuration file looks like. It is recommended that you use one of the configurations shipped with the JBoss Cache distribution and tweak according to your needs rather than write one from scratch.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- ===================================================================== -->
<!--                                                                       -->
<!--  Sample JBoss Cache Service Configuration                             -->
<!--                                                                       -->
<!-- ===================================================================== -->

<server>

   <!-- ===================================================================== -->
   <!-- Defines JBoss Cache configuration                                     -->
   <!-- ===================================================================== -->

   <!-- Note the value of the 'code' attribute has changed since JBC 1.x -->
   <mbean code="org.jboss.cache.jmx.CacheJmxWrapper" name="jboss.cache:service=Cache">

      <!-- Ensure JNDI and the TransactionManager are started before the
           cache.  Only works inside JBoss AS; ignored otherwise -->
      <depends>jboss:service=Naming</depends>
      <depends>jboss:service=TransactionManager</depends>

      <!-- Configure the TransactionManager -->
      <attribute name="TransactionManagerLookupClass">
         org.jboss.cache.transaction.GenericTransactionManagerLookup
      </attribute>

      <!-- Node locking level : SERIALIZABLE
                                REPEATABLE_READ (default)
                                READ_COMMITTED
                                READ_UNCOMMITTED
                                NONE             -->
      <attribute name="IsolationLevel">REPEATABLE_READ</attribute>

      <!-- Lock parent before doing node additions/removes -->
      <attribute name="LockParentForChildInsertRemove">true</attribute>
```

```
<!-- Valid modes are LOCAL (default)
                     REPL_ASYNC
                     REPL_SYNC
                     INVALIDATION_ASYNC
                     INVALIDATION_SYNC   -->
<attribute name="CacheMode">REPL_ASYNC</attribute>

<!-- Name of cluster. Needs to be the same for all JBoss Cache nodes in a
     cluster in order to find each other.
-->
<attribute name="ClusterName">JBossCache-Cluster</attribute>

<!--Uncomment next three statements to use the JGroups multiplexer.
   This configuration is dependent on the JGroups multiplexer being
   registered in an MBean server such as JBossAS.  This type of
   dependency injection only works in the AS; outside it's up to
   your code to inject a ChannelFactory if you want to use one.
-->
<!--
<depends optional-attribute-name="MultiplexerService"
  proxy-type="attribute">jgroups.mux:name=Multiplexer</depends>
<attribute name="MultiplexerStack">tcp</attribute>
-->

<!-- JGroups protocol stack properties.
   ClusterConfig isn't used if the multiplexer is enabled above.
-->
<attribute name="ClusterConfig">
   <config>
      <!-- UDP: if you have a multihomed machine, set the bind_addr
           attribute to the appropriate NIC IP address -->
      <!-- UDP: On Windows machines, because of the media sense feature
           being broken with multicast (even after disabling media sense)
           set the loopback attribute to true -->
      <UDP mcast_addr="228.1.2.3" mcast_port="48866"
           ip_ttl="64" ip_mcast="true"
           mcast_send_buf_size="150000" mcast_recv_buf_size="80000"
           ucast_send_buf_size="150000" ucast_recv_buf_size="80000"
           loopback="false"/>
      <PING timeout="2000" num_initial_members="3"/>
      <MERGE2 min_interval="10000" max_interval="20000"/>
      <FD shun="true"/>
      <FD_SOCK/>
      <VERIFY_SUSPECT timeout="1500"/>
      <pbcast.NAKACK gc_lag="50" retransmit_timeout="600,1200,2400,4800" />
      <UNICAST timeout="600,1200,2400",4800/>
      <pbcast.STABLE desired_avg_gossip="400000"/>
      <FC max_credits="2000000" min_threshold="0.10"/>
      <FRAG2 frag_size="8192"/>
      <pbcast.GMS join_timeout="5000" join_retry_timeout="2000"
                  shun="true" print_local_addr="true"/>
      <pbcast.STATE_TRANSFER/>
   </config>
</attribute>

<!--
```

```
          The max amount of time (in milliseconds) we wait until the
          initial state (ie. the contents of the cache) are retrieved from
          existing members in a clustered environment
      -->
      <attribute name="StateRetrievalTimeout">20000</attribute>

      <!--
          Number of milliseconds to wait until all responses for a
          synchronous call have been received.
      -->
      <attribute name="SyncReplTimeout">20000</attribute>

      <!-- Max number of milliseconds to wait for a lock acquisition -->
      <attribute name="LockAcquisitionTimeout">15000</attribute>


      <!-- Specific eviction policy configurations. This is LRU -->
      <attribute name="EvictionConfig">
         <config>
            <attribute name="wakeUpIntervalSeconds">5</attribute>
            <!-- This defaults to 200000 if not specified -->
            <attribute name="eventQueueSize">200000</attribute>
            <attribute name="policyClass">org.jboss.cache.eviction.LRUPolicy</attribute>

            <!-- Cache wide default -->
            <region name="/_default_">
               <attribute name="maxNodes">5000</attribute>
               <attribute name="timeToLiveSeconds">1000</attribute>
            </region>
            <region name="/org/jboss/data">
               <attribute name="maxNodes">5000</attribute>
               <attribute name="timeToLiveSeconds">1000</attribute>
            </region>
            <region name="/org/jboss/test/data">
               <attribute name="maxNodes">5</attribute>
               <attribute name="timeToLiveSeconds">4</attribute>
            </region>
            <region name="/test">
               <attribute name="maxNodes">10000</attribute>
               <attribute name="timeToLiveSeconds">4</attribute>
            </region>
            <region name="/maxAgeTest">
               <attribute name="maxNodes">10000</attribute>
               <attribute name="timeToLiveSeconds">8</attribute>
               <attribute name="maxAgeSeconds">10</attribute>
            </region>
         </config>
      </attribute>
   </mbean>
</server>
```

11.2. Reference table of XML attributes

A list of definitions of each of the XML attributes used above. If the description of an attribute states that it is *dynamic* , that means it can be changed after the cache is created and started.

Name	Description
BuddyReplicationConfig	An XML element that contains detailed buddy replication configuration. See section on Buddy Replication for details.
CacheLoaderConfig	An XML element that contains detailed cache loader configuration. See chapter on Cache Loaders for details.
CacheLoaderConfiguration	*Deprecated* . Use `CacheLoaderConfig` .
CacheMode	LOCAL, REPL_SYNC, REPL_ASYNC, INVALIDATION_SYNC or INVALIDATION_ASYNC. Defaults to LOCAL. See the chapter on Clustering for details.
ClusterConfig	The configuration of the underlying JGroups stack. Ignored if `MultiplexerService` and `MultiplexerStack` are used. See the various *-service.xml files in the source distribution `etc/META-INF` folder for examples. See the JGroups documentation [http://www.jgroups.org] or the JGroups wiki page [http://wiki.jboss.org/wiki/Wiki.jsp?page=JGroups] for more information.
ClusterName	Name of cluster. Needs to be the same for all nodes in a cluster in order for them to communicate with each other.
EvictionPolicyConfig	Configuration parameter for the specified eviction policy. See chapter on eviction policies for details. This property is *dynamic* .
ExposeManagementStatistics	Specifies whether interceptors that provide statistics should have statistics gathering enabled at startup. Also controls whether a `CacheMgmtInterceptor` (whose sole purpose is gathering statistics) should be added to the interceptor chain. Default value is *true* . See the JBoss Cache Statistics section section for more details.
FetchInMemoryState	Whether or not to acquire the initial in-memory state from existing members. Allows for hot caches when enabled. Also see the `fetchPersistentState` element

	in `CacheLoaderConfig` . Defaults to `true` . This property is *dynamic* .
InactiveOnStartup	Whether or not the entire tree is inactive upon startup, only responding to replication messages after `activateRegion()` is called to activate one or more parts of the tree. If true, property `FetchInMemoryState` is ignored. This property should only be set to true if `UseRegionBasedMarshalling` is also `true` .
StateRetrievalTimeout	Time in milliseconds to wait for state retrieval. This should be longer than `LockAcquisitionTimeout` as the node providing state may need to wait that long to acquire necessary read locks on the cache. This property is *dynamic* .
IsolationLevel	Node locking isolation level : SERIALIZABLE, REPEATABLE_READ (default), READ_COMMITTED, READ_UNCOMMITTED, and NONE. Note that this is ignored if NodeLockingScheme is OPTIMISTIC. Case doesn't matter. See documentation on Transactions and Concurrency for more details.
LockAcquisitionTimeout	Time in milliseconds to wait for a lock to be acquired. If a lock cannot be acquired an exception will be thrown. This property is *dynamic* .
LockParentForChildInsertRemove	Controls whether inserting or removing a node requires a write lock on the node's parent (when pessimistic locking is used) or whether it results in an update of the parent node's version (when optimistic locking is used). The default value is `false` .
MarshallerClass	An instance of `org.jboss.cache.marshall.Marshaller` used to serialize data to byte streams. Defaults to `org.jboss.cache.marshall.VersionAwareMarshaller` if not specified.
MultiplexerService	The JMX object name of the service that defines the JGroups multiplexer. In JBoss AS 5.0 this service is normally defined in the jgroups-multiplexer.sar. This XML attribute can only be handled by the JBoss AS MBean deployment services; if it is included in a file passed to a `CacheFactory` the factory's creation of the cache will fail. Inside JBoss AS, the attribute should be specified using the "depends optional-

	attribute-name" syntax shown in the example above. Inside the AS if this attribute is defined, an instance of `org.jgroups.jmx.JChannelFactoryMBean` will be injected into the `CacheJmxWrapper` which will use it to obtain a multiplexed JGroups channel. The configuration of the channel will be that associated with `MultiplexerStack`. The `ClusterConfig` attribute will be ignored.
MultiplexerStack	The name of the JGroups stack to be used with the cache cluster. Stacks are defined in the configuration of the external `MultiplexerService` discussed above. In JBoss AS 5 this is normally done in the jgroups-multiplexer.sar/META-INF/multiplexer-stacks.xml file. The default stack is `udp`. This attribute is used in conjunction with `MultiplexerService`.
NodeLockingScheme	May be PESSIMISTIC (default) or OPTIMISTIC.
ReplicationVersion	Tells the cache to serialize cluster traffic in a format consistent with that used by the given release of JBoss Cache. Different JBoss Cache versions use different wire formats; setting this attribute tells a cache from a later release to serialize data using the format from an earlier release. This allows caches from different releases to interoperate. For example, a 2.1.0 cache could have this value set to "2.0.0", allowing it to interoperate with a 2.0.0 cache. Valid values are a dot-separated release number, with any final qualifer also separated by a dot, e.g. "2.0.0" or "2.0.0.GA". Values that indicate a 1.x release are not supported in the 2.x series.
ReplQueueInterval	Time in milliseconds for elements from the replication queue to be replicated. Only used if `UseReplQueue` is enabled. This property is *dynamic*.
ReplQueueMaxElements	Max number of elements in the replication queue until replication kicks in. Only used if `UseReplQueue` is enabled. This property is *dynamic*.
SyncCommitPhase	This option is used to control the behaviour of the commit part of a 2-phase commit protocol, when using REPL_SYNC (does not apply to other cache modes). By default this is set to `false`. There is a performance penalty to enabling this, especially when running in a large cluster, but the upsides are greater cluster-wide

	data integrity. See the chapter on clustered caches for more information on this. This property is *dynamic* .
SyncReplTimeout	For synchronous replication: time in milliseconds to wait until replication acks have been received from all nodes in the cluster. It is usually best that this is greater than `LockAcquisitionTimeout` . This property is *dynamic* .
SyncRollbackPhase	This option is used to control the behaviour of the rollback part of a 2-phase commit protocol, when using REPL_SYNC (does not apply to other cache modes). By default this is set to `false` . There is a performance penalty to enabling this, especially when running in a large cluster, but the upsides are greater cluster-wide data integrity. See the chapter on clustered caches for more information on this. This property is *dynamic* .
TransactionManagerLookupClass	The fully qualified name of a class implementing TransactionManagerLookup. Default is JBossTransactionManagerLookup. There is also an option of GenericTransactionManagerLookup for example.
UseInterceptorMbeans	*Deprecated* . Use `ExposeManagementStatistics` .
UseRegionBasedMarshalling	When unmarshalling replicated data, this option specifies whether or not to support use of different classloaders for different cache regions. This defaults to `false` if unspecified.
UseReplQueue	For asynchronous replication: whether or not to use a replication queue. Defaults to `false` .

2

JMX References

12.1. JBoss Cache Statistics

The following table describes the statistics currently available and may be collected via JMX.

Table 12.1. JBoss Cache Management Statistics

CacheLoaderInterceptor	CacheLoaderMisses	long	Number of unsuccessful attempts to load a node through a cache loader.
CacheMgmtInterceptor	Hits	long	Number of successful attribute retrievals.
CacheMgmtInterceptor	Misses	long	Number of unsuccessful attribute retrievals.
CacheMgmtInterceptor	Stores	long	Number of attribute store operations.
CacheMgmtInterceptor	Evictions	long	Number of node evictions.
CacheMgmtInterceptor	NumberOfAttributes	int	Number of attributes currently cached.
CacheMgmtInterceptor	NumberOfNodes	int	Number of nodes currently cached.
CacheMgmtInterceptor	ElapsedTime	long	Number of seconds that the cache has been running.
CacheMgmtInterceptor	TimeSinceReset	long	Number of seconds since the cache statistics have been reset.
CacheMgmtInterceptor	AverageReadTime	long	Average time in milliseconds to retrieve a cache attribute, including unsuccessful attribute retrievals.
CacheMgmtInterceptor	AverageWriteTime	long	Average time in milliseconds to write a cache attribute.
CacheMgmtInterceptor	HitMissRatio	double	Ratio of hits to hits and misses. A hit is a get attribute operation that results in an object being returned to the client. The retrieval may be from a cache loader if the entry isn't in the local cache.
CacheMgmtInterceptor	ReadWriteRatio	double	Ratio of read operations to write operations. This is the ratio of cache hits and misses to cache stores.
CacheStoreInterceptor	CacheLoaderStores	long	Number of nodes written to the cache loader.
InvalidationInterceptor	Invalidations	long	Number of cached nodes that have been invalidated.
PassivationInterceptor	Passivations	long	Number of cached nodes that have been passivated.
TxInterceptor	Prepares	long	Number of transaction prepare operations performed by this interceptor.
TxInterceptor	Commits	long	Number of transaction commit operations performed by this interceptor.
TxInterceptor	Rollbacks	long	Number of transaction rollbacks operations performed by this interceptor.

12.2. JMX MBean Notifications

The following table depicts the JMX notifications available for JBoss Cache as well as the cache events to which they correspond. These are the notifications that can be received through the `CacheJmxWrapper` MBean. Each notification represents a single event published by JBoss Cache and provides user data corresponding to the parameters of the event.

Table 12.2. JBoss Cache MBean Notifications

Notification Type	Notification Data	CacheListener Event
org.jboss.cache.CacheStarted	String : cache service name	cacheStarted
org.jboss.cache.CacheStopped	String : cache service name	cacheStopped
org.jboss.cache.NodeCreated	String : fqn	NodeCreated
org.jboss.cache.NodeEvicted	String : fqn	NodeEvicted
org.jboss.cache.NodeLoaded	String : fqn	NodeLoaded
org.jboss.cache.NodeModifed	String : fqn	NodeModifed
org.jboss.cache.NodeRemoved	String : fqn	NodeRemoved
org.jboss.cache.NodeVisited	String : fqn	NodeVisited
org.jboss.cache.ViewChange	String : view	ViewChange
org.jboss.cache.NodeActivate	Object[0]=String: fqn Object[1]=Boolean: pre	NodeActivate
org.jboss.cache.NodeEvict	Object[0]=String: fqn Object[1]=Boolean: pre	NodeEvict
org.jboss.cache.NodeModify	Object[0]=String: fqn Object[1]=Boolean: pre Object[2]=Boolean: isLocal	NodeModify
org.jboss.cache.NodePassivate	Object[0]=String: fqn Object[1]=Boolean: pre	NodePassivate
org.jboss.cache.NodeRemove	Object[0]=String: fqn Object[1]=Boolean: pre Object[2]=Boolean: isLocal	NodeRemove